Universal Messaging Developer Guide

Version 10.5

October 2019

Document ID: NUM-DG-105-20230727

Table of Contents

About this Documentation....5
Online Information and Support....6
Data Protection....6

Universal Messaging Client Development....7

2 Enterprise APIs....9
Overview of the Enterprise Client APIs....10
Running the Sample Applications....11
Enterprise Developer's Guide for Java....13
Enterprise Developer's Guide for C++....101
Enterprise Developer's Guide for C#....146
Enterprise Developer's Guide for VBA....209
Enterprise Developer's Guide for Python....216

3 Mobile Clients....225
Overview of Mobile Clients....226
iPhone Applications....226
Android Applications....226

4 Web Client APIs....229
Overview of Web Client APIs....230
Web Developer's Guide for Javascript....230
Web Developer's Guide for Java....272

5 API Language Comparisons....279

About this Documentation

- Online Information and Support 6
- Data Protection 6

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com. The site requires credentials for Software AG's Product Support site Empower. If you do not have Empower credentials, you must use the TECHcommunity website.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.asp and give us a call.

Software AG TECHcommunity

You can find documentation and other technical information on the Software AG TECHcommunity website at http://techcommunity.softwareag.com. You can:

- Access product documentation, if you have TECHcommunity credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

Universal Messaging Client Development

Client APIs are available for a wide range of languages at the enterprise level. APIs are also available for building applications for Web-based and mobile device scenarios.

We provide the client API documentation under the following main headings:

- "Enterprise Client APIs" on page 10
- "Web Client APIs" on page 230
- "Mobile Client APIs" on page 226

2 Enterprise APIs

- Overview of the Enterprise Client APIs 10
- Running the Sample Applications 11
- Enterprise Developer's Guide for Java 13
- Enterprise Developer's Guide for C++ 101
- Enterprise Developer's Guide for C# 146
- Enterprise Developer's Guide for VBA 209
- Enterprise Developer's Guide for Python 216

Overview of the Enterprise Client APIs

Our Universal Messaging Enterprise APIs allow developers to implement real-time publish/subscribe functionality into enterprise-class applications using a range of languages:

- **Java**

  The *Universal Messaging Java Client API* is our full-featured enterprise-class client API for Java developers:

  - “Enterprise Developer's Guide for Java” on page 13: developing Java applications/systems that will use Universal Messaging
  - API documentation for the Java Client API is available in the *Reference Guide*.

- **C++**

  The *Universal Messaging C++ Client API* is our full-featured enterprise-class client API for C++ developers:

  - “Enterprise Developer's Guide for C++” on page 101: developing C++ applications/systems that will use Universal Messaging
  - API documentation for the C++ Client API is available in the *Reference Guide*.

- **C# .NET**

  The *Universal Messaging C# Client API* is our full-featured enterprise-class client API for C# developers:

  - “Enterprise Developer's Guide for C#” on page 146: developing C# applications/systems that will use Universal Messaging
  - API documentation for the C# .NET client API is available in the *Reference Guide*.

- **Excel VBA**

  Our *VBA API* allows Microsoft Office applications such as Excel to publish and subscribe to Universal Messaging channels, and to asynchronously receive events in realtime:

  - “Enterprise Developer's Guide for VBA” on page 209

- **Python**

  The *Universal Messaging Python Client API* utilizes the C++ API to provide an enterprise-class API for Python developers:

  - “Enterprise Developer's Guide for Python” on page 216

See Universal Messaging's “Language API Comparison Grid” on page 279 for an overview of basic functional differences between the APIs.

Running the Sample Applications

In order to make it easy for you to test the rich functionality provided by Universal Messaging, the Universal Messaging installation includes compiled versions of all the sample applications as well as native launchers that wrap them and use environment variables to minimize the input required by the user.

In preparation for running the sample applications, you need to open a command prompt at the operating system level, and type in appropriate commands to set up the environment for the sample applications, as described below.

Client Command Prompts on Windows

The client command prompt is a console/shell with environment variables set by a client environment script. Examples of such environment variables include RNAME, PATH, CLASSPATH, Certificate stores etc.

On Windows platforms, you can open a client command prompt using a shortcut in the Windows Start menu. The shortcut is **Software AG > Tools > Universal Messaging Clients *n.n* > *<InstanceName>* > Universal Messaging Client Examples Command Prompt for *<InstanceName>* *n.n***, where *<InstanceName>* is the name of the Universal Messaging realm server instance, and *n.n* is the product version number.

If you want to run the Java, .NET or C++ sample applications that are delivered with the product, you need to set up the appropriate environment in the command prompt. To do this, proceed as follows:

Java	Run the command: `<InstallDir>/UniversalMessaging\java\<InstanceName>\bin\env.bat` This will set up all the environment configurations which are required to compile and run the Java samples. **Note:** You can run the Universal Messaging Tools Runner tool ("runUMTool") from the same command prompt as for the Java sample applications.
.NET (C#)	Run the command: `<InstallDir>/UniversalMessaging\java\<InstanceName>\bin\dotnetenv.bat` This will set up all the environment configurations which are required to compile and run the .NET (C#) samples. Additionally, it will redirect the command prompt location to the .NET examples.
C++	Run the command: `<InstallDir>/UniversalMessaging\java\<InstanceName>\bin\cplusenv.bat`

	This will set up all the environment configurations which are required to compile and run the C++ samples. Additionally, it will redirect the command prompt location to the C++ examples. **Note:** There are some further steps required to compile the C++ sample applications before running them. These steps are described in the section "C++ Prerequisites" on page 141.

Client Command Prompts on UNIX-based platforms

On UNIX-based platforms, you open a client command prompt by executing the appropriate softlink in the `links` folder under the install directory:

```
cd ~/<InstallDir>/UniversalMessaging_<version>/links/Client/umserver
```

where *<version>* is the product version number.

If you want to run the Java, .NET or C++ sample applications that are delivered with the product, you need to set up the appropriate environment in the command prompt. To do this, use one of the following commands:

- For Java:

  ```
  ./Java\ Examples\ Command\ Prompt
  ```

- For .NET:

  ```
  ./DotNet\ Examples\ Command\ Prompt
  ```

- For C++:

  ```
  ./C++\ Examples\ Command\ Prompt
  ```

As an alternative to using the above commands in UNIX-based platforms, you can set up the environment by using one of the following commands:

Java	Run the command: `<InstallDir>/UniversalMessaging/java/<InstanceName>/bin/env.sh` This command also redirects the command prompt location to the Java examples.
.NET (C#)	Run the command: `<InstallDir>/UniversalMessaging/java/<InstanceName>/bin/dotnetenv.sh` This command also redirects the command prompt location to the .NET examples.
C++	Run the command: `<InstallDir>/UniversalMessaging/java/<InstanceName>/bin/cplusenv.sh`

This command also redirects the command prompt location to the C++ examples.

Note:
There are some further steps required to compile the C++ sample applications before running them. These steps are described in the section "C++ Prerequisites" on page 141.

Running the Java Sample Applications when Basic Authentication is Enabled

To use the Java sample applications on a Universal Messaging server on which basic authentication is enabled, you must configure one of the following system properties in the .conf file of the sample application:

- `UM_PASSWORD=`*`password`*. Passes the user password to the session used by the application.
- `PASS_PASSWORD_IN_CONSOLE=true|false`. Specifies whether to provide the user password in the console. The default value is false.

Important:
Do not use the two system properties together.

The Java sample applications are located in the *Software AG_directory* \UniversalMessaging\java*instance_name*\bin directory. You configure the system properties in the configuration file of the sample application that you want to use by adding the following line:

```
wrapper.java.additional.n=-DUM_PASSWORD=password
```

or

```
wrapper.java.additional.n=-DPASS_PASSWORD_IN_CONSOLE=true|false
```

where *`n`* is the consecutive number of the system property and *`password`* is the user password.

Keep in mind that the system creates the session used by the sample application with the default username `SampleApplication`. You can change the username by configuring the `UM_USERNAME` property in the .conf file of the application.

Enterprise Developer's Guide for Java

This guide describes how to develop and deploy Enterprise-class Java applications using Universal Messaging, and assumes you already have Universal Messaging installed.

General Features

Creating a Session

To interact with a Universal Messaging Server, the first thing to do is create a Universal Messaging Session nSession object, which is effectively your logical and physical connection to a Universal Messaging Realm. The steps below describe session creation.

1. Create a nSessionAttributes object with the RNAME value of your choice

   ```
   String[] RNAME={"nsp://127.0.0.1:9000"};
   nSessionAttributes nsa=new nSessionAttributes(RNAME);
   ```

2. Call the create method on nSessionFactory to create your session

   ```
   nSession mySession=nSessionFactory.create(nsa)
   ```

   Alternatively, if you require the use of a session reconnect handler (nReconnectHandler) to intercept the automatic reconnection attempts, pass an instance of that class too in the create method:

   ```
   public class myReconnectHandler implements nReconnectHandler{
     //implement tasks associated with reconnection
   }
   myReconnectHandler rhandler=new myReconnectHandler();
   nSession mySession=nSessionFactory.create(nsa, rhandler);
   ```

3. Initialise the session object to open the connection to the Universal Messaging Realm

   ```
       mySession.init();
   ```

   To enable the use of DataGroups and to create an nDataStream , you should pass an instance of nDataStreamListener to the init call.

   ```
   public void SimpleStreamListener implements nDataStreamListener{
     //implement onMessage callback for nDataStreamListener callbacks
   }
   nDataStreamListener myListener = new SimpleStreamListener();
   nDataStream myStream = mySession.init(myListener);
   ```

After initialising your Universal Messaging session, you will be connected to the Universal Messaging Realm. From that point, all functionality is subject to a Realm ACL check. If you call a method that requires a permission your credential does not have, you will receive an nSecurityException.

Events

A Universal Messaging Event (nConsumeEvent) is the object that is published to a Universal Messaging channel or queue. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an Event Dictionary (see "Event Dictionaries" on page 15).

Each nConsumeEvent object has an nEventAttributes object associated with it which contains all available metadata associated with the event

Constructing an Event

In this Java code snippet, we construct our Universal Messaging Event object (`nConsumeEvent`), and, in this example, pass a byte array data into the constructor:

```
nConsumeEvent evt = new nConsumeEvent( "String", "Hello World".getBytes() );
```

Event Dictionaries

Universal Messaging Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within an event (for related information, see "Events" on page 14).

Event Dictionaries are quite similar to a hash table, supporting primitive types, arrays, and nested dictionaries.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's dictionary.

Event dictionaries can facilitate the automated purging of data from channels through the use of Publish Keys.

Constructing an Event

In this code snippet, we assume we want to publish an event containing the definition of a bond, say, with a name of "bond1":

```
nEventProperties props = new nEventProperties();
props.put("bondname", "bond1");
props.put("price", 100.00);
nConsumeEvent evt = new nConsumeEvent( "atag", props );
channel.publish(evt);
```

Note that in this example code, we also create a new Universal Messaging Event object (`nConsumeEvent`, see "Events" on page 14) to make use of our Event Dictionary (`nEventProperties`).

Channel Joins

Joining a channel to another channel or queue allows you to set up content routing so that events published to the source channel will be passed on to the destination channel/queue automatically. Joins also support the use of filters, thus enabling dynamic content routing.

Please note that while channels can be joined to both channels and queues, queues cannot be used as the source of a join.

When creating a join there is one compulsory parameter and two optional ones. The compulsory parameter is the destination channel. The optional parameters are the maximum join hops and a filter to be applied to the join.

Creating Channel Joins

Joins are created as follows:

```
//Obtain a reference to the source channel
nChannel mySrcChannel = mySession.findChannel( nca );
//Obtain a reference to the destination channel
nChannel myDstChannel = mySession.findChannel( dest );
```

```
//create the join
mySrcChannel.joinChannel( myDstChannel, true, jhc, SELECTOR );
```

Channel joins can be created using the `nmakechanjoin` sample application which is provided in the *<InstallDir>*/UniversalMessaging/java/*<InstanceName>*/bin directory.

For details of this sample application please see the section *Java Client: Create a Channel Join* in the online documentation.

Deleting Channel Joins

Channel joins can also be deleted. Please see the sample application *Java Client: Delete a Channel Join* in the online documentation.

Archive Joins

It is possible to archive messages from a given channel by using an archive join. To perform an archive join, the destination must be a queue in which the archived messages will be stored. An example of this can be seen below:

Since this is an archive join, all events matching the optional selector parameter (all events if no selector is specified) will be put into the archive queue, by design this includes all duplicate events published to the source.

```
nChannelAttributes archiveAtr = new nChannelAttributes();
archiveAtr.setName(rchanName);
nQueue archiveQueue = mySession.findQueue(archiveAtr);
mySrcChannel.joinChannelToArchive(archiveQueue);
```

Inter-Cluster Joins

Inter-cluster joins are added and deleted in almost exactly the same way as normal joins. The only differences are that the two clusters must have an inter-cluster connection in place, and that since the clusters do not share a namespace, each channel must be retrieved from nodes in their respective clusters, rather than through the same node. For example :

```
nChannel cluster1chan1 = realmNode1.findChannel(channelattributes1);
nChannel cluster2chan1 = realmNode4.findChannel(channelattributes2);
cluster1chan1.joinChannel(cluster2chan1);
```

Related Links

For a conceptional overview of channel joins, see the section *Data Routing using Channel Joins* in the *Concepts* guide.

For a description of how to set up and manage channel joins, see the section *Creating Channel Joins* in the *Administration Guide*. The description details the usage based on the Enterprise Manager, but the same general principles apply if you are using the API.

Google Protocol Buffers

Overview

Google Protocol Buffers are a way of efficiently serializing structured data. They are language and platform neutral and have been designed to be easily extensible. The structure of your data is defined once, and then specific serialization and deserialization code is produced specifically to handle your data format efficiently.

For general details on using Google protocol buffers, see the section *Google Protocol Buffers* in the *Concepts* guide.

Using Google Protocol Buffers with Universal Messaging

To create an nProtobuf event, simply build your protocol buffer as normal and pass it into the nProtobuf constructor along with the message type used (see the programmatic example below).

```
Example.Builder example = Example.newBuilder();
example.setEmail("example@email.com");
example.setName("Name");
example.setAddress1("Norton Foldgate");
example.setHouseNumber(1);
byte[] buffer = example.build().toByteArray();
nProtobufEvent evt = new nProtobufEvent(buffer,"example");
myChannel.publish(evt);
```

nProtobuf events are received by subscribers in the normal way.

```
public void go(nConsumeEvent evt) {
    if (evt instanceof nProtobufEvent) {
        totalMsgs++;
        // Get the data of the message
        byte[] buffer = evt.getEventData();
        if(((nProtobufEvent) evt).getTypeName().equals("BidOffer")){
            BidOffer bid = null;
            bid = BidOffer.parseFrom(buffer);
            //......//
        }
     }
}
```

The Enterprise Manager can be used to view, edit and republish protocol buffer events, even if the Enterprise Manager is not running on the same machine as the server. The Enterprise Manager is able to parse the protocol buffer message and display the contents, rather than the binary data.

All descriptors will be automatically synced across the cluster if the channel is cluster-wide.

Programmatic example

```
//Create a realm node (this is standard administration API connection)
       realm = new nRealmNode(new nSessionAttributes(testServer.getDefaultAdapter()));
        realm.waitForEntireNameSpace();
//Create a channel with the descriptors.
Path path =Paths.get("../../changeManagement/test/protobuf/SAGTester.fds");
byte[] bytes = Files.readAllBytes(path);
        byte[][] descriptors = new byte[1][bytes.length];
        descriptors[0]=bytes;
```

```
        myAttribs.setProtobufDescriptorSets(descriptors);
myChannel = nsession.createChannel(myAttribs);
```

Then we can publish using the protobuf serialized as usual, along with the "name" of the protobuf message type.

```
nProtobufEvent pbe = new nProtobufEvent(tester.toByteArray(), "SAGTester");
            myChannel.publish(pbe);
```

You can then use Universal Messaging style message filters, as you would for normal events. e.g. "Name='test'".

Updating the Google Protocol Buffer

The protocol buffer definition files associated with a store (i.e. a channel or a queue) can be updated without requiring the store to be deleted and re-created. Once updated, all filtering will be done with the new protobuf definitions.

To update the protocol buffer definitions for a store programmatically, proceed as follows:

Stores have a method `updateProtobufDescriptors(byte[][] descriptors)`. This takes an array of the descriptors to be applied to the channel. After calling this method, the new descriptors will be loaded and will be used for filtering on the channel from that point onwards. The code for performing this can be seen below.

```
nChannel myChannel = session.findChannel(channelAttributes);
myChannel.updateProtobufDefinitions(descriptors2);
```

This API is available via the client API. The Admin API is not required in order to perform these operations.

Instructions for updating the protocol buffer definitions using the Enterprise Manager are available in the section *Editing Channels* of the *Administration Guide*.

Publish / Subscribe Using Channels/Topics

Publish / Subscribe is one of several messaging paradigms available in Universal Messaging. Universal Messaging Channels are a logical rendezvous point for publishers (producers) and subscribers (consumers) or data (events).

Universal Messaging Data Streams and Data Groups provide an alternative style of Publish/Subscribe where user subscriptions can be managed remotely on behalf of clients (see "Publish / Subscribe Using Data Streams and Data Groups" on page 31).

Universal Messaging Channels equate to Topics if you are using the Universal Messaging Provider for JMS.

Under the publish / subscribe paradigm, each event is delivered to each subscriber once and only once per subscription, and is not typically removed from the channel as a result of the message being consumed by an individual client.

This section demonstrates how Universal Messaging pub / sub works in Java.

Creating a Channel

Channels can be created programmatically as detailed below, or they can be created using the Enterprise Manager.

In order to create a channel, first of all you must create an nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using an RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession' we can then begin creating the channel object. Channels have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them.

To create a channel, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setMaxEvents(0);
cattrib.setTTL(0);
cattrib.setType(nChannelAttributes.PERSISTENT_TYPE);
cattrib.setName("mychannel");
nChannel myChannel=mySession.createChannel(cattrib);
```

Now we have a reference to a Universal Messaging channel within the realm.

Note:
The set of permissible characters you can use to name a channel is described in the section *Creating Channels* in the Enterprise Manager section of the *Administration Guide*.

Setting User and Group Permissions during Channel Creation

User and a group permissions can be created using the factory methods defined in the nStorePermission class.

For example, a user permission can be created with

```
nStorePermission.createForUser(<username>, <host>, <permission_mask>)
```

or

```
nStorePermission.createForUser(<subject>, <permission_mask>)
```

where <username> and <host> are String parameters, <subject> is a String in the format "<username>@<host>" and <permission_mask> is a long representing the mask for the corresponding user/group.

The permission mask must be generated using the nStorePermissionGenerator class. The utility provides methods for building channel permissions from an EnumSet. The following enumeration is defined:

- nChannelStorePermission - for all permissions that can be applied on a channel

Here is an example for generating the permission mask:

```
long channelPermission = nStorePermissionGenerator.getChannelPermissions
    (EnumSet.of(nChannelStorePermission.MANAGE, nChannelStorePermission.PUBLISH,
                nChannelStorePermission.PURGE));
```

The Client API contains the following method which is accessible from a session instance:

```
createChannel(nChannelAttributes attributes, long initialEid,
    Collection<nStorePermission> channelPermissions)
```

The method for creating multiple stores is overloaded and accepts collection with permissions that can be applied to the corresponding store. Here is an example:

```
create(nChannelAttributes[] attr, Collection<Collection<nStorePermission>>
permissionsList)
```

The permissions for each store are also a collection, since multiple permissions can be applied on a single store during creation. If the create procedure fails for one of the stores, the others are created successfully. The reason for the failure can be found using the methods defined in the nCreateResult class which is used as a returned value for each store when multiple channels are requested to be created from the client.

If applying the requested permissions fails when attempting to create a single channel, an nSecurityException is thrown containing the name of the subject for which the operation has failed. For example, if the client tries to grant permissions for a group which does not exist, the operation will fail and the channel will not be created. The client is authorized to grant permissions on store creation only for existing groups.

Here is a code sample illustrating the usage for creating a channel:

```
long userPermission = nStorePermissionGenerator.getChannelPermissions(
    EnumSet.of(nChannelStorePermission.MANAGE, nChannelStorePermission.PUBLISH));
long secondUserPermission = nStorePermissionGenerator.getChannelPermissions(
    EnumSet.of(nChannelStorePermission.PUBLISH, nChannelStorePermission.PURGE));
nStorePermission firstPermission = nStorePermission.createForUser(
    "user", "127.0.0.1", userPermission);
nStorePermission secondPermission = nStorePermission.createForUser(
    "username@127.0.0.1", secondUserPermission);
Collection<nStorePermission> channelPermissions =
    Arrays.asList(firstPermission, secondPermission);
nChannelAttributes channelAttributes = new nChannelAttributes("channelToCreate");
session.createChannel(channelAttributes, 0, channelPermissions);
```

Finding a Channel

In order to find a channel programmatically you must create your nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using the correct RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the channel object. Channels have an associated set of attributes, that define their behavior within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them,

To find a channel previously created, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setName("mychannel");
nChannel myChannel=mySession.findChannel(cattrib);
```

This returns a reference to a Universal Messaging channel within the realm.

Behavior if a channel has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "CHANNEL DELETED" and an event data component equal to "The channel has been deleted".

Publishing events to a Channel

There are 2 types of publish available in Universal Messaging for channels:

- "Reliable Publish" on page 22
- "Transactional Publish" on page 22

Reliable Publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional Publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once the session has been established with the Universal Messaging realm server and the channel has been located, an event must be constructed prior to a publish call being made to the channel.

For reliable publish, there are a number of method prototypes on a channel that allow us to publish different types of events onto a channel. Here are examples of some of them. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
myChannel.publish(new nConsumeEvent("TAG", message.getBytes()));
//Publishing a dictionary (nEventProperties)
nEventProperties props = new nEventProperties();
props.put("bondname", "bond1");
props.put("price", 100.00);
nConsumeEvent evt = new nConsumeEvent( "atag", props, "data".toBytes());
myChannel.publish(evt);
// publishing an XML document
InputStream is = new FileInputStream( aFile);
DOMParser p = new DOMParser();
p.parse( new InputSource( is ) );
Document doc = p.getDocument();
myChannel.publish( "XML", doc );
```

Transactional Publish

Transactional publishing provides a means of verifying that the server received the events from the publisher, and therefore provides guaranteed delivery.

There are similar prototypes available to the developer for transactional publishing. Once the session is established and the channel located, we then need to construct the events for the transaction and publish these events to the transaction. Only when the transaction has been committed will the events become available to subscribers on the channel.

Below are some code snippets for transactional publishing:

```
//Publishing a single event in a transaction
nTransactionAttributes tattrib=new nTransactionAttributes(myChannel);
nTransaction myTransaction=nTransactionFactory.create(tattrib);
myTransaction.publish(new nConsumeEvent("TAG", message.getBytes()));
myTransaction.commit();
//Publising multiple events in a transaction
Vector messages=new Vector();
messages.addElement(message1);
nTransactionAttributes tattrib=new nTransactionAttributes(myChannel);
nTransaction myTransaction=nTransactionFactory.create(tattrib);
```

```
myTransaction.publish(messages);
myTransaction.commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify that a transaction has been committed or aborted, a call can be made on the transaction that will determine if the events within the transaction were successfully received by the Universal Messaging Realm Server. This call can be made regardless of whether the connection was lost and a new connection was created.

The following code snippet demonstrates how to query the Universal Messaging Realm Server to see if the transaction was committed:

```
boolean committed = myTransaction.isCommitted(true);
```

Sending XML Dom Objects

Universal Messaging provides inbuilt support for XML based messaging.

XML can be published as either a String or a DOM Document object.

A summary of the code needed to publish and consume XML data is provided below. For more information please see the Universal Messaging publish XML and consume XML examples.

Publishing

The code to read an XML file and publish it as DOM Document is as follows:

```
//Create an input stream
InputStream is = new FileInputStream( aFile );
//Create a DOM Parser object
DOMParser p = new DOMParser();
//Parse from the input stream
p.parse( new InputSource( is ) );
//Get the XML Document
doc = p.getDocument();
//Publish the Dom Document
myChannel.publish( tag, doc )
```

Subscribing

The code to consume XML is as follows:

```
//The nConsumEventListener Callback
void go( nConsumeEvent evt ) {
        //get the DOM Document from the Universal Messaging event
        Document doc = evt.getDocument()>
        //pass it to the Universal Messaging xmlHelper class
        xmlHelper xh = new xmlHelper( doc );
        //output the XML to standard out
        xh.dumpDoc();
}
```

Asynchronous Subscriber

Asynchronous channel subscribers consume events from a callback on an interface that all asynchronous subscribers must implement. We call this interface an nEventListener.

The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

An example of such a simple listener is shown below:

```
public class mySubscriber implements nEventListener {
  public mySubscriber() throws Exception {
    // construct your session
    // and channel objects here
    // begin consuming events from the channel at event id 0
    // i.e. the beginning of the channel
    myChannel.addSubscriber(this , 0);
  }
  public void go(nConsumeEvent event) {
    System.out.println("Consumed event "+event.getEventID());
  }
  public static void main(String[] args) {
    new mySubscriber();
  }
}
```

Asynchronous consumers can also be created using a selector, which defines a set of event properties and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put(“BONDNAME”,”bond1”);
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the addSubscriber method shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Channel Iterator

Events can be synchronously consumed from a channel using a channel iterator object. The iterator will sequentially move through the channel and return events as and when the iterator getNext() method is called.

If you are using iterators so that you know when all events have been consumed from a channel please note that this can also be achieved using an asynchronous subscriber by calling the nConsumeEvents isEndOfChannel() method.

An example of how to use a channel iterator is shown below:

```
public class myIterator {
  nChannelIterator iterator = null;
  public myIterator() throws Excepetion {
    // construct your  session and channel objects
    // start the iterator at the beginning of the channel (event id 0)
    iterator = myChannel.createIterator(0);
```

```
  }
  public void start() {
    while (true) {
      nConsumeEvent event = iterator.getNext();
      go(event);
    }
  }
  public void go(nConsumeEvent event) {
    System.out.println("Consumed event "+event.getEventID());
  }
  public static void main(String[] args) {
    myIterator itr = new myIterator();
    itr.start();
  }
}
```

Synchronous consumers can also be created using a selector, which defines a set of event properties and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'"
```

And pass this string into the createIterator method shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Batched Subscribe

If a client application needs to subscribe to multiple channels it is more efficient to batch these subscriptions into a single server call. This is achieved using the subscribe method of nSession rather than first finding the nChannel object and then calling the subscribe method of nChannel.

The following code snippet demonstrates how to subscribe to two Universal Messaging channels in one server call:

```
public class myEventListener implements nEventListener {
  public void go(nConsumeEvent evt) {
    System.out.println("Received an event!");
  }
}
public void demo(){
  nSubscriptionAttributes[] arr = new nSubscriptionAttributes[2];
  arr[0] = new nSubscriptionAttributes("myChan1", "", 0, myLis1);
  arr[1] = new nSubscriptionAttributes("myChan2", "", 0, myLis2);
  arr = mySession.subscribe(arr);
  for (int i = 0; i < arr.length; i++) {
    if (!arr[i].wasSuccessful()) {
      handleSubscriptionFailure(arr[i]);
    }
    //subscription successful
  }
}
public void handleSubscriptionFailure(nSubscriptionAttributes subAtts){
```

```
  subAtts.getException().printStackTrace();
}
```

The nSubscriptionAttributes class is used to specify which channels to subscribe to. The second two parameters of the constructor represent the selector to use for the subscription and the event ID to subscribe from.

It is possible that the subscription may fail; for example, the channel may not exist or the user may not have the required privileges. In this situation, calling wasSuccessful() on the nSubscriptionAttributes will return false and getException() will return the exception that was thrown.

If the subscription is successful then the nChannel object can be obtained from the nSubscriptionAttributes as shown in the following code snippet:

```
  nChannel chan = subAtts.getChannel();
```

Batched Find

In client applications, it is quite common to have multiple Channels or Queues that one is trying to find. In these scenarios, the batched find call built into nSession is extremely useful.

The following code snippet demonstrates how to find 2 Universal Messaging Channels in one server call:

```
public void demo(){
  nChannelAttributes[] arr = new nChannelAttributes[2];
  nChannel[] channels = new nChannels[2];
  arr[0] = new nChannelAttributes("myChan1");
  arr[1] = new nChannelAttributes("myChan2");
  nFindResult[] results = mySession.find(arr);
  for (int i = 0; i < results.length; i++) {
    if (!results[i].wasSuccessful()) {
      handleSubscriptionFailure(results[i]);
    } else if (results[i].isChannel) {
      channels[i] = results[i].getChannel();
    }
  }
}
public void handleSubscriptionFailure(nFindResult result){
  result.getException().printStackTrace();
}
```

To perform the same operation for Queues, simply use the example above and exchange nChannel for nQueue, and check each result returned to see if the isQueue() flag is set.

Using Durable Subscriptions

Universal Messaging provides the ability for both asynchronous and synchronous consumers to be durable. A durable subscription allows state to be kept at the server with regard to what events have been consumed by a specific consumer of data.

Asynchronous Durable Consumer

An example of how to create a durable subscription that begins from event ID 0, is persistent and is used in conjunction with an asynchronous event consumer:

```
public class mySubscriber implements nEventListener {
  public mySubscriber() throws Exception {

    // construct your session and channel objects here.
    // create the durable subscription and begin consuming events from
    // the channel at event id 0, i.e. the beginning of the channel
    nDurableAttributes.nDurableType type =
      nDurableAttributes.nDurableType.Named;
    nDurableAttributes attr = nDurableAttributes.create(type, "unique1");
    attr.setPersistent(true);
    attr.setClustered(false);
    attr.setStartEID(0);
    nDurable named = myChannel.getDurableManager().add(attr);
    myChannel.addSubscriber(this, named);
  }
  public void go(nConsumeEvent event) {
    System.out.println("Consumed event "+event.getEventID());
  }
  public static void main(String[] args) {
    new mySubscriber();
  }
}
```

Synchronous Durable Consumer

An example of how to create a durable subscription that begins from event ID 0, is persistent and is used in conjunction with a synchronous event consumer:

```
public class myIterator {
  nChannelIterator iterator = null;
  public myIterator() throws Exception {
    // construct your session and channel objects here
    // create the durable subscription and begin consuming events from the channel
    // at event id 0, i.e. the beginning of the channel
    nDurableAttributes.nDurableType type =
      nDurableAttributes.nDurableType.Named;
    nDurableAttributes attr = nDurableAttributes.create(type, "unique2");
    attr.setPersistent(true);
    attr.setClustered(false);
    attr.setStartEID(0);
    nDurable named = myChannel.getDurableManager().add(attr);
    iterator = myChannel.createIterator(named);
  }
  public void start() {
    while (true) {
      nConsumeEvent event = iterator.getNext();
      go(event);
    }
  }
  public void go(nConsumeEvent event) {
    System.out.println("Consumed event "+event.getEventID());
  }
```

```
  public static void main(String[] args) {
  myIterator itr = new myIterator();
  itr.start();
}
```

Both synchronous and asynchronous channel consumers allow message selectors to be used in conjunction with durable subscriptions. Please see the API documentation for more information.

There are also different ways in which events consumed by named consumers can be acknowledged. By specifying that 'auto acknowledge' is true when constructing either the synchronous or asynchronous consumers, then each event is acknowledged as consumed automatically. If 'auto acknowledge' is set to false, then each event consumed has to be acknowledged by calling the ack() method:

```
public void go(nConsumeEvent event) {
  System.out.println("Consumed event "+event.getEventID());
  event.ack();
}
```

Shared Durable Consumer

Multiple subscribers can hold a subscription to the same durable and all the subscribers will process events in a round-robin manner.

```
nDurableAttributes.nDurableType type =
    nDurableAttributes.nDurableType.Shared;
nDurableAttributes attr = nDurableAttributes.create(type, "shared durable");
attr.setPersistent(persistent);
attr.setClustered(isClusterWide);
attr.setStartEID(startEid);
nDurable shared = channels.getDurableManager().add(attr);
```

Serial Durable Consumer

Multiple subscribers can hold a subscription to the same durable and all the subscribers will process events in a serial manner.

```
nDurableAttributes.nDurableType type =
    nDurableAttributes.nDurableType.Serial;
nDurableAttributes attr = nDurableAttributes.create(type, "serial durable");
attr.setPersistent(persistent);
attr.setClustered(isClusterWide);
attr.setStartEID(startEid);
nDurable serial = channels.getDurableManager().add(attr);
```

The Merge Engine and Event Deltas

In order to streamline publish/subscribe applications it is possible to deliver only the portion of an event's data that has changed rather than the entire event. These event deltas minimise the amount of data sent from the publisher and ultimately delivered to the subscribers.

The publisher simply registers an event and can then publish changes to individual keys within the event. The subscriber will receive a full event on initial subscription, which contains the most up to date state of the event. After the initial message, only the key/value pairs which have changed since the last message will be sent to the client.

Publisher - Registered Events

In order to publish event deltas the publisher uses the Registered Event facility available on a Universal Messaging Channel. Please note that the channel must have been created with the Merge Engine and it must have a single Publish Key. The publish key represents the primary key for the channel and the registered events. So for example if you are publishing currency rates you would setup a channel as such:

```
   nChannelAttributes cattr
            = new nChannelAttributes("RatesChannel", 0, 0,
nChannelAttributes.SIMPLE_TYPE);
//
// This next line tells the server to Merge incoming events based on the publish
//  key name and the name of the registered event
//
        cattr.useMergeEngine(true);
//
// Now create the Publish Key (See publish Keys for a full description
//
        nChannelPublishKeys[] pks = new nChannelPublishKeys[1];
        pks[0] = new nChannelPublishKeys("ccy", 1);
        cattr.setPublishKeys(pks);
//
// Now create the channel
//
        myChannel = mySession.createChannel(cattr);
```

At this point the server will have a channel created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent audEvent = myChannel.createRegisteredEvent("AUD");
        nEventProperties props = audEvent.getProperties();
        props.put("bid", 0.8999);
        props.put("offer", 0.9999);
        props.put("close", "0.8990");
        audEvent.commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to a ccy value of "AUD". We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props.put("bid", 0.9999);
        audEvent.commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

Subscriber - nEventListener

The subscriber implements nEventListener in the usual way and does not need to do anything different in order to receive either event deltas or snapshots containing the result of one or more merge operations. The standard nEventListener will receive a full event when the subscriptions is initiated. Thereafter it will receive only deltas. If at any time the user is disconnected then it will receive a fresh update of the full event on reconnection - followed by a resumption of delta delivery.

If you wish to differentiate between snapshot events and delta events then the nConsumeEvent attributes can be used as follows:

```
event.getAttributes().isDelta();
```

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same channel or queue. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages. The priority is a numeric value in the range 0 (lowest priority) to 9 (highest priority).

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent evt;
    ...
evt.getAttributes().setPriority((byte) 9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically, events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis, and the effects become more noticeable as load increases.

Note:
If events are stored for replay at a later stage, for example for a durable subscriber who is currently not consuming events, higher priority events will be delivered earlier than lower priority events when the durable subscriber starts consuming the events, even if the lower priority events were created much earlier .

Publish / Subscribe Using Data Streams and Data Groups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging Data Groups are lightweight structures designed to facilitate Publish/Subscribe . When using Data Groups, user subscriptions are managed remotely in a way that is transparent to subscribers. Universal Messaging Channels provide an alternative style of Publish/Subscribe where the subscribers manage their subscriptions directly.

This section demonstrates Universal Messaging pub / sub using Data Groups in Java, and provides example code snippets for all relevant concepts:

DataStreamListener

If a nSession is created with a nDataStreamListener then it will receive asynchronous callbacks via the onMessage implementation of the nDataStreamListener interface. The nDataStreamListener will receive events when:

- An event is published directly to this particular nDataStream
- An event is published to any nDataGroup which contains this nDataStream
- An event is published to an nDataGroup which contains a nested nDataGroup containing this nDataStream
- An example of how to create a session with an nDataStreamListener interface is shown below:

```
public class DataGroupClient implements nDataStreamListener{

    nSession mySession;
    public DataGroupClient( String realmURLs){
        nSessionAttributes nsa = new nSessionAttributes(realmURLs);
        mySession = nSessionFactory.create(nsa, this);
        mySession.init(this);
    }

    ////
    // nDataStreamListener Implementation
    ////

    //Callback received when event is available
    public void onMessage(nConsumeEvent event){

            //some code to process the message

    }
}
```

Creating and Deleting DataGroups

Creating Universal Messaging DataGroups

nDataGroups can be created programmatically as detailed below, or they can be created using the Universal Messaging enterprise manager.

In order to create a nDataGroup, first of all you must create an nSession object, which is effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using an RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession object instance 'mySession', you can then create DataGroups. The create DataGroup methods will return the nDataGroup if it already exists.

The code snippets below demonstrate the creation of nDataGroups:

Create a Single nDataGroup

```
nDataGroup myGroup = mySession.createDataGroups("myGroup");
```

Create Multiple nDataGroups

```
String[] groups = {"myFirstGroup", "mySecondGroup"};
nDataGroup[] myGroups = mySession.createDataGroups(groups);
```

Creating DataGroups with DataGroupListeners and ConflationAttributes

It is also possible to specify additional properties when creating DataGroups:

- nDataGroupListener - To specify a listener for DataGroup membership changes
- nConflationAttributes - To specify attributes which control event merging and delivery throttling for the DataGroup

Now we have a reference to a Universal Messaging DataGroup it is possible to publish events

Deleting Universal Messaging DataGroups

There are various deleteDataGroup methods available on nSession which will delete DataGroups. It is possible to specify single nDataGroups or arrays of nDataGroups.

Managing DataGroup Membership

DataGroups are extremely lightweight from both client and server perspectives; a back-end process, such as a Complex Event Processing engine, can simply create DataGroups and then add or remove users (or even entire nested DataGroups) based on bespoke business logic. A user who is removed from one DataGroup and added to another will continue to receive events without any interruption to service, or indeed explicit awareness that any DataGroup change has occurred.

This page details some of the typical operations that DataGroup management process would carry out. Please see our Java sample apps for more detailed examples of DataGroup management.

Tracking Changes to DataGroup Membership (DataGroupListener)

The nDataGroupListener interface is used to provide asynchronous notifications when nDataGroup membership changes occur. Each time a user (nDataStream) or nDataGroup is added or removed from a nDataGroup a callback will be received.

```
public class datagroupListener implements nDataGroupListener {
    nSession mySession;
    public datagroupListener(nSession session){
        mySession = session;
       //add this class as a listener for all nDataGroups on this Universal Messaging
        // realm
        mySession.getDataGroups(this);
    }

    ////
    //DataGroupListener Implementation
    ///
    public void addedGroup (nDataGroup to, nDataGroup group, int count){
        //Called when a group has been added to the 'to' data group.
        //count is the number of nDataStreams that will receive any events published

        //to this nDataGroup
    }
    public void addedStream (nDataGroup group, nDataStream stream, int count){
        //Called when a new stream has been added to the data group.
    }
    public void createdGroup (nDataGroup group){
        //Called when a group has been created.
    }
    public void deletedGroup (nDataGroup group){
        //Called when a group has been deleted.
    }
    public void deletedStream (nDataGroup group, nDataStream stream, int count,
                               boolean serverRemoved){
        //Called when a stream has been deleted from the data group.
       //serverRemoved is true if the nDataStream was removed because of flow control
    }
    public void removedGroup (nDataGroup from, nDataGroup group, int count){
        //Called when a group has been removed from the 'from' data group.
    }

}
```

There are three ways in which the nDataGroupListener can be used:

Listening to an individual DataGroup

Listeners can be added to individual DataGroups when they are created or at any time after creation. The code snippets illustrate both approaches:

```
mySession.createDataGroup(dataGroupName, datagroupListener);
```

```
myDataGroup.addListener(datagroupListener);
```

Listening to the Default DataGroup

The Default nDataGroup is a DataGroup to which all nDataStreams are added by default. If you add a DataGroupListener to the default DataGroup then callbacks will be received when:

- a nDataStream is connected/disconnected
- a nDataGroup is created or deleted

Listening to all DataGroups on a Universal Messaging Realm

The code snippet below will listen on all nDataGroups (including the default DataGroup).

```
mySession.getDataGroups(datagroupListener);
```

Adding and Removing DataGroup Members

The nDataGroup class provides various methods for adding and removing nDataStreams and nDataGroups. Please see the nDataGroup API documentation for a full list of methods. Examples of some of these are provided below:

```
    //Add a nDataStream (user) to a nDataGroup
    public void addStreamToDataGroup(nDataGroup group, nDataStream user){
        group.add(user);
    }
    //Remove a nDataStream (user) from a nDataGroup
    public void removeStreamFromDataGroup(nDataGroup group, nDataStream user){
        group.remove(user);
    }
    //Add a nDataGroup to a nDataGroup
    public void addNestedDataGroup(nDataGroup parent, nDataGroup child){
        parent.add(child);
    }
    //Remove a nDataGroup from a nDataGroup
    public void removeNestedDataGroup(nDataGroup parent, nDataGroup child){
        parent.remove(child);
    }
```

DataGroup Conflation Attributes

Enabling Conflation on DataGroups

Universal Messaging DataGroups can be configured so that conflation (merging and throttling of events) occurs when messages are published. Conflation can be carried out in several ways and these are specified using a nConflationAttributes object. The ConflationAttributes object is passed in to the DataGroup when it is created initially.

The nConflationAttributes object has two properties action and interval. Both of these are passed into the constructor.

The action property specifies whether published events should replace previous events in the DataGroup or be merged with them. These properties are defined by static fields:

```
nConflationAttributes.sMergeEvents
nConflationAttributes.sDropEvents
```

The interval property specifies the interval in milliseconds between event fanout to subscribers. An interval of zero implies events will be fanned out immediately.

Creating a Conflation Attributes Object

```
//ConflationAttributes specifying merge events  and no throttled delivery
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 0);
//ConflationAttributes specifying merge events and throttled delivery at 1 second
intervals
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 1000);
//ConflationAttributes specifying drop events and throttled delivery at 1 second
intervals
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sDropEvent, 1000);
```

Create a Single nDataGroup with Conflation Attributes

```
public class datagroupListener implements nDataGroupListener {
    nSession mySession;
    nDataGroup myDataGroup;
    public datagroupListener(nSession session, nConflationAttributes confattribs,
                             String dataGroupName){
        mySession = session;
        //create a DataGroup passing in this class as a nDataGroupListener and a
        //ConflationAttributes
        myDataGroup = mySession.createDataGroup(dataGroupName, this, confattribs)
    }
}
```

Create Multiple nDataGroups with Conflation Attributes

```
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 1000);
String[] groups = {"myFirstGroup", "mySecondGroup"};
nDataGroup[] myGroups = mySession.createDataGroups(groups, confattribs);
```

Publishing Events to Conflated DataGroups With A Merge Policy

At this point the server will have a nDataGroup created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent audEvent = myDataGroup.createRegisteredEvent();
        nEventProperties props = audEvent.getProperties();
        props.put("bid", 0.8999);
        props.put("offer", 0.9999);
        props.put("close", "0.8990");
        audEvent.commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to the data group that could be called 'aud'. We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props.put("bid", 0.9999);
        audEvent.commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

When a data group has been created with Merge conflation, all registered events published to that data group will have their nEventProperties merged into the snapshot event, before the delta event is delivered to the consumers.

When using Merge conflation with an interval (ie throttling), all updates will be merged into a conflated event (as well as the snapshot event) that will be delivered within the chosen interval. For example, consider the following with a merge conflated group and an interval set to 100ms (i.e. maximum of 10 events a second):

```
Scenario 1
t0      - Publish Message1, Bid=1.234       (This message will be immediately
                                            delivered, and merged into the snapshot)
t10     - Publish Message2, Offer=1.234     (This message will be held as a
                                             conflation event, and merged into the
snapshot)
t20     - Publish Message3, Bid=1.345       (This message will be merged with the
                                             conflated event, and with the snapshot)
t100    - Interval hit                      (Conflated event containing
Offer=1.234,Bid=1.345
                                             is delivered to consumers)
                                             Interval timer reset to +100ms, ie t200
t101    - Publish Message4, Offer=1.345     (This message will be held as a conflation
 event,
                                             and merged into the snapshot)
Where t0...tn is the time frame in milliseconds from now.
Scenario 2
t0      - Publish Message1, Bid=1.234       (This message will be immediately
                                            delivered, and merged into the snapshot)
t100    - Interval hit                      (Nothing is sent as there has been no
update
                                             since t0)
t101    - Publish Message2, Offer=1.234     (This message will be immediately
                                            delivered, and merged into the snapshot)
                                             Interval timer reset to +100ms, ie t201
```

Meanwhile, if any new consumers are added to the Data Group, they will always consume the most up to date snapshot and then begin consuming any conflated updates after that.

Publishing Events to Conflated DataGroups With A Drop Policy

If you have specified a "Drop" policy in your ConflationAttributes then events are published in the normal way rather than using nRegisteredEvent.

Consuming Conflated Events from a DataGroup

The subscriber doesn't need to do anything different to receive events from a DataGroup with conflation enabled. If nRegisteredEvents are being delivered then the events will contain only the fields that have changed will be delivered. In all other circumstances an entire event is delivered to all consumers.

DataGroups Event Publishing

You can get references to any DataGroup from the nSession object. There are various writeDataGroup methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataGroups.

```
myDataGroup = mySession.getDataGroup("myGroup");
nEventProperties props = new nEventProperties();
//You can add other types in a dictionary object
props.put("key0string"+x, "1"+x);
props.put("key1int", (int) 1);
props.put("key2long", (long) -11);
nConsumeEvent evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession.writeDataGroup(evt1, myDataGroup);
```

DataStream Event Publishing

You can get references to any nDataStream (user) from the nSession object if you call getDefaultDataGroup(). You can also access nDataStreams by implementing the nDataGroupListener interface. Please see DataGroup management for more information. This will deliver callbacks as users are connected/disconnected. There are various writeDataStream methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataStreams.

```
nEventProperties props = new nEventProperties();
//You can add other types in a dictionary object
props.put("key0string"+x, "1"+x);
props.put("key1int", (int) 1);
props.put("key2long", (long) -11);
nConsumeEvent evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession.writeDataStream(evt1, myDataStream)
```

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same datagroup. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages. The priority is a numeric value in the range 0 (lowest priority) to 9 (highest priority).

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent evt;
    ...
evt.getAttributes().setPriority((byte) 9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically, events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis, and the effects become more noticeable as load increases.

Note:
If events are stored for replay at a later stage, for example for a durable subscriber who is currently not consuming events, higher priority events will be delivered earlier than lower priority events when the durable subscriber starts consuming the events, even if the lower priority events were created much earlier .

Message Queues

Message queues are one of several messaging paradigms supported by Universal Messaging.

Universal Messaging provides message queue functionality through the use of queue objects. Queues are the logical rendezvous point for publishers (producers) and subscribers (consumers) or data (events).

Message queues differ from publish / subscribe channels in the way that events are delivered to consumers. Whilst queues may have multiple consumers, each event is typically only delivered to one consumer, and once consumed (popped) it is removed from the queue.

Universal Messaging also supports non destructive reads (peeks) from queues which enable consumers to see what events are on a queue without removing it from the queue. Any event which has been peeked will still be queued for popping in the normal way. The Universal Messaging enterprise manager also supports the ability to visually peek a queue using it s snoop capability.

This section demonstrates how Universal Messaging message queues work in Java, and provide examples code snippets for all relevant concepts:

Creating a Queue

In order to create a queue, first of all you must create your nSession object, which is your effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then begin creating the queue object. Queues have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them.

To create a queue, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setChannelMode(nChannelAttributes.QUEUE_MODE);
cattrib.setMaxEvents(0);
cattrib.setTTL(0);
cattrib.setType(nChannelAttributes.PERSISTENT_TYPE);
cattrib.setName("myqueue");
nQueue myQueue=mySession.createQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Note:
For information about the set of permissible characters you can use to name a queue, see the section *Creating Queues* in the Enterprise Manager section of the *Administration Guide*.

Setting User and Group Permissions during Queue Creation

User and a group permissions can be created using the factory methods defined in the nStorePermission class.

For example, a user permission can be created with

```
nStorePermission.createForUser(<username>, <host>, <permission_mask>)
```

or

```
nStorePermission.createForUser(<subject>, <permission_mask>)
```

where <username> and <host> are String parameters, <subject> is a String in the format "<username>@<host>" and <permission_mask> is a long representing the mask for the corresponding user/group.

The permission mask must be generated using the nStorePermissionGenerator class. The utility provides methods for building queue permissions from an EnumSet. The following enumeration is defined:

- nQueueStorePermission - for all permissions that can be applied on a queue

Here is an example for generating the permission mask:

```
long queuePermission = nStorePermissionGenerator.getQueuePermissions
    (EnumSet.of(nQueueStorePermission.MANAGE, nQueueStorePermission.PUSH));
```

The Client API contains the following method which is accessible from a session instance:

```
createQueue(nChannelAttributes attributes, Collection<nStorePermission>
queuePermissions)
```

The method for creating multiple stores is overloaded and accepts collection with permissions that can be applied to the corresponding store. Here is an example:

```
create(nChannelAttributes[] attr, Collection<Collection<nStorePermission>>
permissionsList)
```

The permissions for each store are also a collection, since multiple permissions can be applied on a single store during creation. If the create procedure fails for one of the stores, the others are created successfully. The reason for the failure can be found using the methods defined in the nCreateResult class which is used as a returned value for each store when multiple queues are requested to be created from the client.

If applying the requested permissions fails when attempting to create a single queue, an nSecurityException is thrown containing the name of the subject for which the operation has failed. For example, if the client tries to grant permissions for a group which does not exist, the operation will fail and the queue will not be created. The client is authorized to grant permissions on store creation only for existing groups.

Here is a code sample illustrating the usage for creating a queue:

```
long userPermission = nStorePermissionGenerator.getQueuePermissions(
    EnumSet.of(nQueueStorePermission.MANAGE, nQueueStorePermission.PUSH));
long groupPermissionMask = nStorePermissionGenerator.getQueuePermissions(
    EnumSet.of(nQueueStorePermission.PEEK, nQueueStorePermission.PURGE));
nStorePermission firstPermission = nStorePermission.createForUser(
    "user", "127.0.0.1", userPermission);
nStorePermission secondPermission = nStorePermission.createForUser(
    "secondUser", "10.0.34.71", secondUserPermission);
Collection<nStorePermission> queuePermissions =
    Arrays.asList(firstPermission, secondPermission);
nChannelAttributes channelAttributes = new nChannelAttributes("queueToCreate");
session.createQueue(channelAttributes, queuePermissions);
```

Finding a Queue

In order to find a queue, first of all the queue must be created. This can be achieved through the Universal Messaging Enterprise Manager, or programmatically. First of all you must create your nSession object, which is your effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the queue object. Queues have an associated set of attributes, that define their behavior within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them,

To find a queue previously created, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setName("myqueue");
nQueue myQueue=mySession.findQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Behavior if a queue has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "QUEUE DELETED" and an event data component equal to "The queue has been deleted".

Queue Publish

There are 2 types of publish available in Universal Messaging for queues:

Reliable publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once you have established a session and find a queue, you then need to construct an event and publish the event onto the queue.

For reliable publish, here is the example code for how to publish events to a queue. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
myQueue.push(new nConsumeEvent("TAG", message.getBytes()));
// publishiing an XML document
InputStream is = new FileInputStream( aFile );
DOMParser p = new DOMParser();
p.parse( new InputSource( is ) );
Document doc = p.getDocument();
myQueue.push( "XML", doc );
```

Transactional Publish

Transactional publishing provides us with a method of verifying that the server receives the events from the publisher, and provides guaranteed delivery.

There are similar prototypes available to the developer for transaction publishing. Once we have established our session and our queue, we then need to construct our events and our transaction and publish these events to the transaction. Then the transaction will be committed and the events available to consumers to the queue.

Below is a code snippet of how transactional publishing is achieved:

```
Vector messages=new Vector();
Messages.addElement(message1);
nTransactionAttributes tattrib=new nTransactionAttributes(myQueue);
nTransaction myTransaction=nTransactionFactory.create(tattrib);
myTransaction.publish(messages);
myTransaction.commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify that a transaction has been committed or aborted, an call can be made on the transaction that will determine if the events within the transactional were successfully received by the Universal Messaging Realm Server.

```
boolean committed = myTransaction.isCommitted(true);
```

Which will query the Universal Messaging Realm Server to see if the transaction was committed.

Asynchronous Queue Consuming

Asynchronous queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

An example of an asynchronous queue reader is shown below:

```
public class myAsyncQueueReader implements nEventListener {
        nQueue myQueue = null;
        public myAsyncQueueReader() throws Exception {
                // construct your session and queue objects here
                // begin consuming events from the queue
                nQueueReaderContext ctx = new nQueueReaderContext(this, 10);
                nQueueAsyncReader reader = myQueue.createAsyncReader(ctx);
        }
        public void go(nConsumeEvent event) {
                System.out.println("Consumed event "+event.getEventID());
        }
        public static void main(String[] args) {
                try {
                        new myAsyncQueueReader();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
 }
```

Asynchronous queue consumers can also be created using a selector, which defines a set of event properties and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put(“BONDNAME”,”bond1”);
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Synchronous Queue Consuming

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

An example of a synchronous queue reader is shown below:

```
public class mySyncQueueReader {
        nQueueSyncReader reader = null;
        nQueue myQueue = null;
        public mySyncQueueReader() throws Exception {
                // construct your session and queue objects  here
                // construct the queue reader
                nQueueReaderContext ctx = new
                nQueueReaderContext(this, 10);
                reader = myQueue.createReader(ctx);
        }
        public void start() throws Exception {
```

```
                while (true) {
                        // pop events from the queue
                        nConsumeEvent event = reader.pop();
                        go(event);
                }
        }
        public void go(nConsumeEvent event) {
                System.out.println("Consumed event "+event.getEventID());
        }
        public static void main(String[] args) {
                try {
                        mySyncQueueReader sqr = new mySyncQueueReader();
                         sqr.start();
                } catch (Exception e) {
                         e.printStackTrace();
                }
        }
}
```

Synchronous queue consumers can also be created using a selector, which defines a set of event properties and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Asynchronous Transactional Queue Consuming

Asynchronous transactional queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional asynchronous queue reader is shown below:

```
public class myAsyncTxQueueReader implements nEventListener {
        nQueueAsyncTransactionalReader reader = null;
        nQueue myQueue = null;
        public myAsyncTxQueueReader() throws Exception {
                // construct your session and queue objects here
                // begin consuming events from the queue
                nQueueReaderContext ctx = new
```

```
                nQueueReaderContext(this, 10);
                reader = myQueue.createAsyncTransactionalReader(ctx);
        }
        public void go(nConsumeEvent event) {
                System.out.println("Consumed event "+event.getEventID());
                 reader.commit();
        }
        public static void main(String[] args) {
                try {
                        new myAsyncTxQueueReader();
                } catch (Exception e) {
                         e.printStackTrace();
                }
        }
}
```

As previously mentioned, the big difference between a transactional asynchronous reader and a standard asynchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the .rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with event id's 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Asynchronous queue consumers can also be created using a selector, which defines a set of event properties and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put(“BONDNAME”,”bond1”);
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Synchronous Transactional Queue Consuming

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional synchronous queue reader is shown below:

```
public class mySyncTxQueueReader {
        nQueueSyncTransactionReader reader = null;
        nQueue myQueue = null;
        public mySyncTxQueueReader() throws Exception {
                // construct your session and queue objects  here
                // construct the transactional queue reader
                nQueueReaderContext ctx = new
                nQueueReaderContext(this, 10);
                reader = myQueue.createTransactionalReader(ctx);
        }
        public void start() throws Exception {
                while (true) {
                        // pop events from the queue
                        nConsumeEvent event = reader.pop();
                        go(event);
                        // commit each event consumed
                        reader.commit(event.getEventID());
                }
        }
        public void go(nConsumeEvent event) {
                System.out.println("Consumed event "+event.getEventID());
        }
        public static void main(String[] args) {
                try {
                        mySyncTxQueueReadersqr = new mySyncTxQueueReader();
                        sqr.start();
                }catch (Exception e) {
                        e.printStackTrace();
                }
        }
 }
```

As previously mentioned, the big difference between a transactional synchronous reader and a standard synchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the .rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with event id's 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Synchronous queue consumers can also be created using a selector, which defines a set of event properties and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put(“BONDNAME”,”bond1”);
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Queue Browsing / Peeking

Universal Messaging provides a mechanism for browsing (peeking) queues. Queue browsing is a non-destructive read of events from a queue. The queue reader used by the peek will return an array of events, the size of the array being dependent on how many events are in the queue, and the window size defined when your reader context is created. For more information, please see the Universal Messaging Client API documentation.

An example of a queue browser is shown below:

```
public class myQueueBrowser {
        nQueueReader reader = null;
        nQueuePeekContext ctx = null;
        nQueue myQueue = null;
        public myQueueBrowser() throws Exception {
                // construct your session and queue objects here
                // create the queue reader
                reader = myQueue.createReader(new
                nQueueReaderContext());
                ctx = nQueueReader.createContext(10);
        }
        public void start() throws Exception {
                boolean more = true;
                long eid =0;
                while (more) {
                        // browse (peek) the queue
                        nConsumeEvent[] evts = reader.peek(ctx);
                        for (int x=0; x < evts.length; x++) {
                                go(evts[x]);
                        }
                        more = ctx.hasMore();
                }
        }
        public void go(nConsumeEvent event) {
                System.out.println("Consumed event "+event.getEventID());
        }
        public static void main(String[] args) {
                try {
                        myQueueBrowser qbrowse = new myQueueBrowser();
                         qbrowse.start();
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
 }
```

Queue browsers can also be created using a selector, which defines a set of event properties and their values that a browser is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put(“BONDNAME”,”bond1”);
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueuePeekContext object shown in the example code, then your browser will only receive messages that contain the correct value for the event property BONDNAME.

Request Response

Subscriber Based Publish

Universal Messaging can easily be used to issue request/response message exchanges. To accomplish this, the requester simply publishes an event to a request queue and then listens for a response to be issued on a response queue. The responder tags this response with the username of the requester, and this ensures that only the requester will see the response event.

Requester

The requester publishes an event to a request queue and then listens for a response to be issued on a response queue. The response will be tagged with the tag of the requester. This tag is specified during the initial configuration of the session, as shown below:

```
mySession = nSessionFactory.create(nsa, this,"subscriber tag");
```

After setting this, the requester simply publishes an event to the request queue and listens for a reply on the response queue.

An example Java requester is available in the examples section.

Responder

The responder listens to the request channel and responds to each request event. To ensure the message is only delivered to the correct recipient, the Subscriber Name must be set on the response event. The response event's data can contain the relevant information the user needs.

```
//Having received a request event req, and established a connection to
//a response queue respQueue.
System.out.println("Received request");

//Retrieve username of request sender.
String requester = req.getPublishUser();
//Construct reply message.
String text = "Response: " + new String(req.getEventData());
//Construct reply event
nEventProperties atr = new nEventProperties();
nConsumeEvent resp = new nConsumeEvent(atr, text.getBytes());
//Set recipient of the event to the requester's tag to reply.
resp.setSubscriberName(requester.getBytes());
respQueue.push(resp);
```

An example Java responder is available in the examples section.

Provider for JMS

Overview of the Provider for JMS

This guide describes the programmatic steps you can take in order to use Universal Messaging Provider for JMS. There is also a section that will help you discover how to perform administration of JMS objects in the Universal Messaging Enterprise Manager section.

Note:
The JMS Client for Universal Messaging supports JMS 1.1. Currently we do not support JMS 2.0.

Topics and Queues

Universal Messaging Enterprise Server includes support for JMS functionality such as topics and queues.

JMS topics correspond to channels in Universal Messaging publish / subscribe, and JMS queues correspond to Universal Messaging message queues.

Communication Drivers

The pluggable communications drivers enable JMS to be used on public, private and wireless networks transparently. JMS functionality can be delivered over normal TCP/IP based sockets, SSL enabled sockets, HTTP and HTTPS. When supporting JMS using HTTP or HTTPS, Universal Messaging can traverse proxy servers and network address translation devices, and it does not require any additional web server to perform.

JMS Message Filtering

JMS message selector support is offered via Universal Messaging's high performance server side message filtering engine. This ensures that only messages with content that your clients register an interest in are delivered over the network, thus conserving network bandwidth.

Horizontal Scalability Connection Factories

The Universal Messaging API for JMS allows you to configure *horizontal scalability* connection factories. These factories allow clients to publish messages in a round-robin fashion, so that one message or transaction gets published to the first realm node or cluster, the next message to the next realm node or cluster, and so on. These connection factories have the following limitations:

1. Event consumption is not supported through these factories, so for example message listeners cannot be registered and consumers cannot be created via the sessions created from these connection factories.

2. The sessions created through these connections factories do not support distributed (XA) transactions.

Message Batching

Universal Messaging provides a non-standard extension to JMS that allows clients to send messages to the JMS provider in batches. This mechanism can lead to performance improvements, because multiple messages can be sent in a single API call without waiting for the acknowledgment for each message.

This is particularly relevant for synchronous publishing (PERSISTENT messages and SyncPersistent enabled), as multiple messages in a single API call will allow you to send the messages in a single transaction to achieve higher throughput.

If you are using asynchronous publishing (NON_PERSISTENT messages), you can also batch the messages, but no performance gain will be expected. This is because multiple messages can be sent without batching in separate API calls without waiting for any confirmation from the server.

JMSAdmin: Sample application for creating realm resources

Universal Messaging's Enterprise Manager tool supports JNDI using the same Universal Messaging Channel based context used by the JMSAdmin example.

The example (jmsdmin.java) source code demonstrates how to store Universal Messaging Provider for JMS components into a JNDI service provider. The default service provider for the example uses Universal Messaging's own Universal Messaging Context to store JMS objects references, however any JNDI context provider can be used, from LDAP through to NIS. The Universal Messaging context is discussed in more detail here. The Universal Messaging Context stores references in a channel called `/naming/defaultContext`.

JMSAdmin creates all required resources on a Universal Messaging realm. The command syntax is as follows:

```
java com.pcbsys.nirvana.nSpace.JMSAdmin
  -DRNAME [-DPRINCIPAL] [-DPASSWORD] -DCONTEXT_FACTORY
  -DPROVIDER_URL JMSAdmin bind | unbind | list | queueFactory |
  topicFactory |connectionFactory | queue | topic name / alias
```

where the -D parameters have the following meaning:

- RNAME is the realm name of the Universal Messaging server you wish to connect to. If no RNAME is provided the default RNAME of nsp://localhost:9000 is used.

  You can specify a cluster of realms by specifying a comma-separated list of connection URLs, for example "nsp://localhost:9000,nsp://localhost:9010".

  If you want to use a *horizontal scalability* connection factory, you can specify several connection URLs, using the *horizontal scalability URL syntax*.

  See the section *URL Syntax for Horizontal Scalability* in the *Concepts* guide for details of this syntax.

  Example:

  (UM1,UM2)(UM3,UM4) - Indicates 2 clusters, one consisting of UM1 and UM2 and the other consisting of UM3 and UM4, so only 2 connections will be constructed here.

- PRINCIPAL is the subject (if any) that your JNDI service provider requires.
- PASSWORD is the PRINCIPAL's password for the JNDI service provider used.
- CONTEXT_FACTORY is the fully qualified class name of the provider's context factory implementation. The default CONTEXT FACTORY is com.pcbsys.nirvana.nSpace.NirvanaContextFactory and is set automatically if no CONTEXT_FACTORY parameter is provided.
- PROVIDER_URL is the custom URL required by the context factory and provider implementation. If no PROVIDER_URL parameter is passed, the default used is nsp://localhost:9000/.

As an example, assume we want to create a TOPIC called rates on a Universal Messaging realm running on our local machine. Typing:

```
java com.pcbsys.nirvana.nSpace.JMSAdmin bind topic rates
```

Will create an event in the `/naming/defaultContext` channel with the following information in the event properties of the event:

```
rates/RefAddr/0/Content=rates
rates/RefAddr/0/Type=Topic
rates/ClassName=javax.JMS.Topic
rates/FactoryName=com.pcbsys.nirvana.nJMS.TopicFactory
rates/RefAddr/0/Encoding=String
```

The topic rates will automatically be created on the Universal Messaging realm running on the PROVIDER_URL value. Assuming you wish to reference your local realm as a TopicConnectionFactory named TopicConnectionFactory in JMS, use the following command:

```
java com.pcbsys.nirvana.nSpace.JMSAdmin bind topicFactory TopicConnectionFactory
```

This will publish an event to the `/naming/defaultContext` channel with the following information in the event dictionary:

```
TopicConnectionFactory/RefAddr/0/Type=TopicConnectionFactory
TopicConnectionFactory
/FactoryName=com.pcbsys.nirvana.nJMS.TopicConnectionFactoryFactory
TopicConnectionFactory/RefAddr/0/Encoding=String
TopicConnectionFactory/ClassName=javax.JMS.TopicConnectionFactory
TopicConnectionFactory/RefAddr/0/Content=nsp\://127.0.0.1\:9000
TopicConnectionFactory/RefAddr/0/Encoding=String
```

Creating a queue can be achieved using the following command:

```
java com.pcbsys.nirvana.nSpace.JMSAdmin bind queue movie
```

Likewise, a JMS queue connection factory called QueueConnectionFactory can be bound into a name space using the following command

```
java com.pcbsys.nirvana.nSpace.JMSAdmin bind queueFactory QueueConnectionFactory
```

Having run both queue related commands, the `/naming/defaultContext` channel will contain 4 events, each one pertaining to the 4 objects that have been bound, namely TopicConnectionFactory, QueueConnectionFactory, rates and movie. The Universal Messaging Context used with your JMS application will now be able to look up these objects and use them within your application.

JMS Client SSL Configuration

This section describes how to use SSL in your Universal Messaging Provider for JMS applications. Universal Messaging supports various wire protocols including SSL enabled sockets and HTTPS.

Once you have created an SSL enabled interface for your realm you need to ensure that your client application passes the required SSL properties either on the connection factory or via system properties used by your JSSE-enabled JVM. The Universal Messaging download contains some sample Java keystore files that will be used in this example.

The first such keystore is the client keystore, called `client.jks`, which can be found in your installation directory, under the `/server/Universal Messaging/bin` directory. The second is the truststore called `nirvanacacerts.jks`, which is also located in the `/server/Universal Messaging/bin` directory.

Custom SSL Properties

Using the sample keystores, you can set custom SSL attributes on JMS as follows:

Setting the SSL Attributes on the JNDI Context

In your properties object the following properties will set SSL attributes on the JNDI Context.

```
env = new Properties();
env.setProperty("java.naming.factory.initial",
    "com.pcbsys.nirvana.nSpace.NirvanaContextFactory");
env.setProperty("java.naming.provider.url", rname);
env.setProperty("nirvana.ssl.keystore.path",
    %INSTALLDIR%\client\Universal Messaging\bin\client.jks);
env.setProperty("nirvana.ssl.keystore.pass", password);
env.setProperty("nirvana.ssl.keystore.cert", certAlias);
    // Certificate alias for the client to use when connecting to an interface
    // with client validation enabled
env.setProperty("nirvana.ssl.truststore.path",
    %INSTALLDIR%\client\Universal Messaging\bin\nirvanacacerts.jks);
env.setProperty("nirvana.ssl.truststore.pass", password);
env.setProperty("nirvana.ssl.protocol", "TLS");
```

Setting the SSL Attributes on the Connection Factory

- You can set the SSL attributes using the same Properties object like this:

  ```
  connectionFactory.setProperties(env);
  Connection con = connectionFactory.createConnection();
  ```

- You can set the SSL attributes using the available setters:

  ```
  connectionFactory.setSSLStores(String keyStorePath, String keyStorePass,
      String trustStorePath, String trustStorePass);
  connectionFactory.setSSLStores(String keyStorePath, String keyStorePass,
      String certificateAlias, String trustStorePath, String trustStorePass);
  connectionFactory.setSSLProtocol(String protocol);
  ```

```
connectionFactory.setSSLEnabledCiphers(String[] enabledCiphers);
Connection con = connectionFactory.createConnection();
```

Setting the SSL Attributes on the Connection

```
Connection con = connectionFactory.createConnection(keyStorePath, keyStorePass,
   keyStoreCert, trustStorePath, trustStorePass, cipherSuite, protocol)
```

JSSE SSL System Properties

The following system properties are used by the jsse implementation in your JVM. You can specify the SSL properties by passing the following as part of the command line for your JMS application:

```
 -Djavax.net.ssl.keyStore=%INSTALLDIR%\client\Universal Messaging\bin\client.jks
 -Djavax.net.ssl.keyStorePassword=password
 -Djavax.net.ssl.trustStore=%INSTALLDIR%\client\Universal
Messaging\bin\nirvanacacerts.jks
 -Djavax.net.ssl.trustStorePassword=password
```

where :

- javax.net.ssl.keyStore is the client keystore location
- javax.net.ssl.keyStorePassword is the password for the client keystore
- javax.net.ssl.trustStore is the CA keystore file location
- javax.net.ssl.trustStorePassword is password for the CA keystore

As well as the above system properties, if you are intending to use https, your JMS applications will require the following system property to be passed in the command line:

```
 -Djava.protocol.handler.pkgs="com.sun.net.ssl.internal.www.protocol"
```

As well as the above, the RNAME used by the JMS application must correspond to the correct type of SSL interface, and the correct hostname and port that was configured earlier.

In JMS, the RNAME corresponds to a JNDI reference. The example JMSADmin application can be used to create a sample file based JNDI context, where the RNAME is specified as the content of the TopicConnectionFactoryFactory reference. Once your SSL interface is created you can simply change this value in your JNDI context to be the RNAME you require your JMS applications to use.

Using Universal Messaging Client System Properties

Instead of the JSSE system properties, you can use the Universal Messaging client system properties to configure secure communication with Universal Messaging realms. The Universal Messaging client system properties configure only the connections to Universal Messaging realms and have no impact on the connections established to other endpoints, unlike the JSSE system properties. If both Universal Messaging client and JSSE system properties are configured, when you create a session to a Universal Messaging realm, the Universal Messaging client properties take precedence.

To configure secure communication in your own applications, set the following system properties:

```
-Dcom.softwareag.um.client.ssl.keystore_path=
%INSTALLDIR%\client\Universal Messaging\bin\client.jks
-Dcom.softwareag.um.client.ssl.keystore_password=password
-Dcom.softwareag.um.client.ssl.certificate_alias=alias
-Dcom.softwareag.um.client.ssl.truststore_path=
%INSTALLDIR%\client\Universal Messaging\bin\nirvanacacerts.jks
-Dcom.softwareag.um.client.ssl.truststore_password=password
-Dcom.softwareag.um.client.ssl.enabled_ciphers=AES-128,AES-192,AES-256
-Dcom.softwareag.um.client.ssl.ssl_protocol=TLS
```

where:

- com.softwareag.um.client.ssl.keystore_path is the client keystore location
- com.softwareag.um.client.ssl.keystore_password is the password for the client keystore
- com.softwareag.um.client.ssl.certificate_alias is the alias of the certificate in the client keystore that is sent to the server if client certificate authentication is required
- com.softwareag.um.client.ssl.truststore_path is the CA keystore file location
- com.softwareag.um.client.ssl.truststore_password is the password for the CA keystore
- com.softwareag.um.client.ssl.enabled_ciphers is a comma-separated list of ciphers from which the client is allowed to choose for secure communication
- com.softwareag.um.client.ssl.ssl_protocol is the protocol that is used for secure communication

JMS Message / Event Mapping

Universal Messaging provides interoperability between JMS and Non-JMS client APIs. The API for the Universal Messaging Provider for JMS shares the same event structures sent over the wire as other Universal Messaging Client APIs. The nConsumeEvent in the Universal Messaging client APIs is the basic structure of all events published and subscribed whether JMS or Non-JMS, Java or C#.

The JMS Message has a distinct structure: the header, the message properties and the body. In the Universal Messaging client API, the nConsumeEvent is the container for the JMS message structure. Any JMS message consumer on a topic or queue expects the nConsumeEvent to be in a predefined format with specific JMS header values, message properties and a message body. The JMS Header values are stored in the nEventAttributes of the nConsumeEvent and any message properties are stored in the nEventProperties objects for the same event. The message body is different for each of the JMS message types (bytes, map, stream, object, text) but it is always stored in the byte[] payload of the nConsumeEvent.

Usability

Publishing a JMS Message using the API for the Universal Messaging Provider for JMS sends an nConsumeEvent to the server with the message body stored in the event payload, i.e. the event byte[]. Each JMS Header exists in the nEventAttributes, and any JMS message properties are stored in the nEventProperties. The Java, C++ and C# Client APIs use the same structure for nConsumeEvent and can therefore all consume JMS Message objects. As there is no equivalent

JMS C# or C++ specification, these APIs will treat these messages as normal nConsumeEvent objects.

JMS provides a Map Message type, within Universal Messaging the map object is represented by an nEventProperties. When the message is published this map is serialised and stored in the event payload. In order to consume this message from C# you can convert the payload back to an nEventProperties using the getPayloadAsDictionary() method.

Publishing a non-JMS Message for consumption by JMS-based API clients also provides a level of interoperability. The API for JMS will interpret any nConsumeEvent objects published by any other non-JMS client API (Java, C#, Javascript, Mobile etc.) as BytesMessage objects and deliver them to the JMS consumers as such.

JMS Message Type Conversion

JMS message types are exposed so that you can publish a native nConsumeEvent and have it received by JMS subscribers in the specified type instead of in the default BytesMessage type.

The JMS Message types are assigned integer values as shown below. The integer values can be used directly when setting the message type, or accessed via the following public static constants in nEventAttributes:

```
JMS_BASE_MESSAGE_TYPE = 0
JMS_MAP_MESSAGE_TYPE = 1
JMS_BYTES_MESSAGE_TYPE = 2
JMS_OBJECT_MESSAGE_TYPE = 3
JMS_STREAM_MESSAGE_TYPE = 4
JMS_TEXT_MESSAGE_TYPE = 5
```

You can set the message type on an nConsumeEvent in the following way (using the message type `JMS_OBJECT_MESSAGE_TYPE` as an example):

```
nEventAttributes eventAttributes = new nEventAttributes();
eventAttributes.setMessageType(nEventAttributes.JMS_OBJECT_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", bytes);
evt.setAttributes(eventAttributes);
```

The data portion of the nConsumeEvent will contain the message body and will need to be set according to the message type as described in the following sections:

JMS_BASE_MESSAGE_TYPE

Publisher:

```
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_BASE_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", byteArray);
evt.setAttributes(attributes);
```

JMS Subscriber:

```
Message message = topicConsumer.receive(2000);
MessageImpl baseMessageImpl = (MessageImpl)message;
baseMessageImpl.getBuffer();
```

The buffer will be equal to the byteArray sent in the data payload of the nConsumeEvent.

JMS_BYTES_MESSAGE_TYPE

Publisher:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
DataOutputStream os = new DataOutputStream(out);
os.writeInt(32);
os.writeInt(1); //true
os.writeBoolean(true);
os.flush();
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_BYTES_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", out.toByteArray());
evt.setAttributes(attributes);
```

JMS Subscriber:

```
Message message = topicConsumer.receive(2000);
BytesMessage bytemessage = (BytesMessage)message;
bytemessage.readInt(); //32
bytemessage.readInt(); //1
bytemessage.readBoolean(); //true
```

JMS_OBJECT_MESSAGE_TYPE

Publisher:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
ObjectOutputStream os = new ObjectOutputStream(out);
os.writeObject(anyObjectThatIsSerializable);
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_OBJECT_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", out.toByteArray());
evt.setAttributes(attributes);
```

JMS Subscriber:

```
Message message = topicConsumer.receive(2000);
ObjectMessage objectMessage = (ObjectMessage) message;
Object object = objectMessage.getObject();
```

The returned object will be the deserialization of the object sent in the data payload of the nConsumeEvent.

JMS_MAP_MESSAGE_TYPE

Publisher:

No direct mapping can be made as Universal Messaging uses internal collections to construct the underlying map message. The publisher will not be able to send an nConsumeEvent with the data portion containing this as we do not expose the serialization of our internals. An option that is however available and can also be applied to other JMS Message types is the use of nEventProperties.

```
nEventProperties props = new nEventProperties();//populate with key/value properties
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_MAP_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", null);
evt.setAttributes(attributes);
evt.setProperties(props);
```

JMS Subscriber:

```
Message message = topicConsumer.receive(2000);
MapMessage mapMessage = (MapMessage)message;
```

Users can then get data using the 'key' of any of the entries in nEventProperties set on the nConsumeEvent by invoking for example:

```
mapMessage.getObjectProperty("key"); //if the type is unknown
mapMessage.getIntProperty("key");
```

and various other built-in getter methods that return different types of properties.

JMS_STREAM_MESSAGE_TYPE

Publisher:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
ObjectOutputStream os = new ObjectOutputStream(out);
Vector<Object> vector = new Vector<Object>();
vector.add(true);
vector.add(110110110);
vector.add(1.05f);
vector.add("abcdef");
os.writeObject(vector);
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_STREAM_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", out.toByteArray());
evt.setAttributes(attributes);
```

JMS Subscriber:

```
Message message = topicConsumer.receive(2000);
StreamMessage streamMessage = (StreamMessage)message;
streamMessage.readBoolean(); //true
streamMessage.readLong(); //110110110
streamMessage.readFloat(); //1.05f
streamMessage.readString(); //abcdef
```

JMS_TEXT_MESSAGE_TYPE

Publisher:

```
String textMessage = "This will be in the text portion of the message";
ByteArrayOutputStream baos = new ByteArrayOutputStream();
baos.write(textMessage.getBytes(StandardCharsets.UTF_8));
nEventAttributes attributes = new nEventAttributes();
attributes.setMessageType(nEventAttributes.JMS_TEXT_MESSAGE_TYPE);
nConsumeEvent evt = new nConsumeEvent("Message", baos.toByteArray());
evt.setAttributes(attributes);
```

JMS Subscriber:

```
Message message_received = topicConsumer.receive(2000);
TextMessage received = (TextMessage)message_received;
String text = received.getText());
```

This will return the same string sent in the data payload of the nConsumeEvent.

Fanout Engine

The Universal Messaging Queue and Channel Fanout Engines are used to store and forward events based on the channel type. JMS uses topics and messages which are equivalent to Universal Messaging channels and events respectively.

Universal Messaging offers several channel types, each of which have different requirements when storing data. The available channel types are summarized in the section *Channel Attributes* in the Concepts guide.

The Fanout Engine for Universal Messaging Provider for JMS uses different criteria to determine storage of events. *No replay of messages* means that it is not necessary to store events if there is no *interest* on the channel or once they have been consumed regardless of the channel type. *Durable Subscribers* require the engine to store the events until the subscriber becomes active and consumes the events. For more information, see "Engine Differences" on page 59.

Interest

The Fanout Engine for Universal Messaging Provider for JMS deals with events published to channels based on 'interest'. If there is no interest present on the channel then any events published can be immediately discarded due to *no replay of messages*. The channel is said to have no interest if there are no durable or active subscribers.

Durable Subscribers

It is often the case that a subscriber needs to receive all events published to a channel including the events published when the subscriber is inactive. With a durable subscriber, any events published while the subscriber is inactive are stored until the subscriber reconnects and consumes the events missed.

See the section *Durable Subscriptions* in the Concepts guide for related information.

No replay of messages

When a JMS subscription is made to a channel, the subscription always begins from the last issued event ID. As no events can be consumed more than once, there is no need to store events once they are consumed. This improves the efficiency of the system because all events can be fanned out to subscribers and then dropped straight away (as long as there are no synchronous consumers or inactive durable subscriptions). This greatly reduces the overhead caused by I/O.

Only in the case of inactive durable subscribers or synchronous consumers are events stored. Once all durable subscribers or synchronous consumers have consumed an event, it is removed from storage as there is no need for it to be kept. Synchronous consumers require the events to be stored because they do not receive events fanned out to all consumers, instead they iterate through the events requesting each event in turn.

Recovery

In the case that a subscriber loses connection to the server, the JMS engine will register a need to temporarily store events for a configurable period of time or until the client reconnects. The time period is defined by the TTL value of the event (if this is non zero) or the EventTimeout value stored in the realm configuration/ClientTimeoutValues under the config tab in the Enterprise Manager which is 60 seconds by default.

Engine Differences

The tables below shows the storage differences between the JMS Engine and the Universal Messaging Queue and Channel Engines. The Universal Messaging engines store events based on the channel type whereas the JMS Engine only stores events when there are synchronous consumers or inactive durable subscribers. The channel type does however determine where the data is stored.

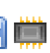
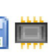
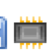
 - Events to be stored on disk prior to delivery

 - Events to be stored in memory prior to delivery

 - Events are not stored prior to delivery

On a Mixed channel, persistent storage to disk or to memory can be individually set on a per-event basis. When appropriate, events on Persistent channels will be stored to disk, and events on Reliable channels will be stored in memory.

JMS Engine

Channel Type	Mixed	Persistent	Reliable
Active Durable Subscribers			
Active shared Durable Subscribers (see note 1)			
Active regular Durable Subscribers, with auto-acknowledgement (see note 2)			
Active regular Durable Subscribers, with client acknowledgement (see note 3)	 	 	
One or more Synchronous Consumers or Inactive Durable Subscribers	 		
No Durable Subscribers			

Channel Type	Mixed	Persistent	Reliable
No Subscribers			

Note:

1. For a shared durable consumer, events will always be persisted in an internal queue for that specific subscriber, but not on the channel.

2. For a regular durable consumer, using auto-acknowledgment, events will be passed on to the subscriber and not stored on the channel.

3. For a regular durable consumer, using client acknowledgment, events will be stored on the channel. When the subscriber acknowledgment arrives, the event will be removed from the channel.

For further information on storage methods, see the section *Using Durable Subscriptions with Multiple Clients* in the Concepts guide.

Universal Messaging Channel Engine

Channel Type	Mixed	Persistent	Reliable
Active Durable Subscribers			
One or more Synchronous Consumers or Inactive Durable Subscribers			
No Durable Subscribers			
No Subscribers			

Universal Messaging Queue Engine

Queue Type	Mixed	Persistent	Reliable
Active Consumers			
No Subscribers			

Resource Adapter for JMS

For *Red Hat JBoss EAP* and *IBM WebSphere Application Server*, the Universal Messaging installation contains a product-specific generic resource adapter for JMS. For details of the supported releases of these third party products, refer to the *System Requirements for Software AG Products* document on the Software AG documentation web site.

The following description uses the term *UMRA* to refer to this Universal Messaging-specific adapter. UMRA is based on the Generic Resource Adapter for JMS.

The UMRA adapter is a `.rar` file that can be deployed and configured using the standard administration tools of your chosen application server. The necessary deployment and configuration steps vary from application server to application server, so refer to the documentation of your application server for the steps involved in this process.

The UMRA `.rar` file is located in `<InstallDir>/`UniversalMessaging`/j2ee/umra.rar` and contains the modified `genericra.jar`, the `nJMS.jar` and `nClient.jar`. A `README.txt` is included and outlines the changes made to the resource adapter. The `ra.xml` descriptor file is located in the `META-INF` directory.

Configurable Properties of UMRA

The Generic Resource Adapter for JMS offers a set of properties that you can configure according to your requirements. UMRA supports a subset of these properties, as listed in the following table.

Parameter	Description	Universal Messaging recommendation	Default
connectionFactoryJndiName	The name of the JMS connection factory.	The name of the JMS connection factory.	
ConnectionValidation	If true, the application server will register an error listener with the JMS connection and invalidate the connection if the error is communicated by Universal Messaging. This will render the connection unusable from the application server perspective and will prevent the native Universal Messaging reconnection mechanism from recovering such connections. So in general, it is recommended to set this parameter to false.		false
DeliveryType	For inbound delivery there are 2 modes: "Asynchronous" and	Purely depending on the requirements. If	

Parameter	Description	Universal Messaging recommendation	Default
	"Synchronous". If the value is left empty, this assumes "Asynchronous" delivery. Synchronous delivery spawns several threads (according to the maxPoolSize property, see below), which are several consumers if the destination is not a Topic destinationType, and blocks each one waiting to consume messages within the thread. This can lead to warnings in some application servers that some threads are stuck. With asynchronous delivery the messages are retrieved serially by Universal Messaging in threads owned by the Universal Messaging client, but then they are dispatched for concurrent processing to the MDBs (message driven bean). The acknowedgement is serial again (a limitation in the implementation).	there are many MDBs it's most probably better to have "Asynchronous" delivery.	
destinationJndiName	The JNDI name of the destination.	The JNDI name of the destination.	
destinationType	Can be javax.jms.Destination, javax.jms.Queue or javax.jms.Topic	Place the proper type of the destination.	javax.jms.Destination
endpointReleaseTimeout	The time to wait for the message driven beans (MDBs) already processing messages to complete. Larger values mean that, when stopping	Use the default value: 180.	180

Parameter	Description	Universal Messaging recommendation	Default
	(undeploying) the application, it may take longer to wait until all MDBs are done.		
JndiProperties	The properties to pass to the JNDI context, e.g. "java.naming.factory.initial=com.pcbsys.nirvana.nSpace.NirvanaContextFactory java.naming.provider.url="nsp://localhost:2600", java.naming.security.authentication=simple"	Set the appropriate properties to connect to Universal Messaging.	
maxPoolSize	The MDB thread pool size for an asynchronous consumer, i.e., the maximum number of messages that can be processed in parallel by MDBs for this consumer. For a synchronous consumer, if not 'SERIAL' deliveryConcurrencyMode and not Topic destinationType, this is the number of threads to block waiting for a message.	Set depending on the load. For high load with heavy MDB processing it could be beneficial to have a higher number here (take into account the capabilities of the hardware and whether MDB is CPU bound, I/O bound, etc.)	8
maxWaitTime	The maximum time to wait to obtain an MDB from the pool in order to process the message. If MDBs are taking their time processing the messages or we have huge redelivery interval, the incoming message might need to wait until a resource is free to process it.	Set depending on the requirements.	300
messageSelector	The message selector for the subscription.	Set depending on the requirements.	
Password			

Parameter	Description	Universal Messaging recommendation	Default
ProviderIntegrationMode	Should be JNDI or ClassName	Set to "JNDI".	
reconnectAttempts	Number of times to try to reconnect if Universal Messaging indicates a connection problem via the JMS Connection listener.	If this is 0, the reconnection will be accomplished by the Universal Messaging native reconnection mechanism, which is recommended approach.	0
reconnectInterval	Reconnection interval in seconds.	See the above item.	0
subscriptionDurability	"Durable" or "Non-Durable" topic subscription	Set depending on the requirements.	Non-Durable
subscriptionName	The name of the durable subscription, if any.	Set depending on the requirements	
useProxyMessages	This should be set to "true" if the application code needs to use the `java.jms.Message#setJMSReplyTo` method - otherwise the call will fail with an exception. The default is "false".	Set depending on the requirements.	false

Unsupported and partially supported properties

In addition to the supported properties listed in the table above, the General Resource Adapter for JMS offers some additional properties that UMRA recognizes but does not support, or supports only partially. These properties are listed in the following table.

Parameter	Description	Universal Messaging recommendation
ackTimeOut	Applicable when HUMode is true.	Don't use this property, UMRA will assign a default value.

Parameter	Description	Universal Messaging recommendation
applicationName	Used if "monitoring" is enabled and will provide JMX monitoring. This is the name of the resource pool (e.g. Server session pool - free and busy resources, wait time etc. to monitor.).	Not supported in Universal Messaging.
batchSize	Doesn't seem to be used in the code.	Don't use this property, UMRA will assign a default value.
CommonSetterMethodName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
ConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
ConnectionFactoryProperties	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
deadMessageConnectionFactoryProperties	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
deadMessageDestinationClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
deadMessageDestinationJndiName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
deadMessageDestinationProperties	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
deadMessageConnectionFactoryJndiName	JNDI name of the JMS connection factory to place dead messages.	Not supported in Universal Messaging.

Parameter	Description	Universal Messaging recommendation
deadMessageDestinationType	javax.jms.Destination, javax.jms.Queue or javax.jms.Topic	Not supported in Universal Messaging.
deliveryConcurrencyMode	Applicable for synchronous delivery. Value is empty or 'SERIAL'. SERIAL means only one thread devoted to message consumption and processing.	Not supported in Universal Messaging.
destinationProperties	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
HUAMode	HUA stands for "hold until ack".	Use the default value. Should be "false" given that scenario in which this is "true" is not well understood.
instanceCount	The number of instances on which UMRA is deployed (value > 1 is applicable only if "loadBalancingRequired" is true). It also can be part of generated JMS clientId if "ClientId" parameter is set.	Load balancing is not supported in Universal Messaging.
loadBalancingRequired	This will provide a kind of load balancing by adding an artificial or provided filter to apply on the message by Universal Messaging.	Set "false". Load balancing is not supported with Universal Messaging.
MDBDeploymentRetryAttempt	Doesn't seem to be used.	Don't use this property, UMRA will assign a default value.
MDBDeploymentRetryInterval	Doesn't seem to be used.	Don't use this property, UMRA will assign a default value.
monitoring	Enable JMX monitoring for the resource pool. Related to "ApplicationName" parameter.	Not supported with Universal Messaging.
Password		
QueueClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.

Parameter	Description	Universal Messaging recommendation
QueueConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
redeliveryAttempts	Applicable in the presence of XA support. This takes effect when a message driven bean (MDB) throws an exception. The redelivery happen purely within the UMRA client, no session recovery, rollbacks whatsoever used.	Not applicable, UMRA doesn't support XA.
redeliveryInterval	Applicable in the presence of XA support. This takes effect when an MDB throws an exception. The redelivery happen purely within the UMRA client; no session recovery, rollbacks whatsoever is used.	Not applicable, UMRA doesn't support XA.
RMPolicy	Applies to XA Support. When RMPolicy is set to "OnePerPhysicalConnection", the XAResource wrapper implementation's isSameRM in the Generic JMS RA would check if both the XAResources use the same physical connection, before delegating to the wrapped objects.	Don't use this property, UMRA will assign a default value. UMRA doesn't support XA.
sendBadMessagesToDMD	If messages are to be sent to a dead message destination (DMD) when there is a problem during the consumption.	Not supported with Universal Messaging.
shareClientid	When we use load balancing, this indicates if all the deployments should use the same JMS Client ID.	Not supported with Universal Messaging.
TopicClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
TopicConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.

Parameter	Description	Universal Messaging recommendation
UnifiedDestinationClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
UseFirstXAForRedelivery	Applicable for XA transactions, covers some corner case, looks there is a problem when set to "true".	Don't use this property, UMRA will assign a default value.
XAConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
XAQueueConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.
XATopicConnectionFactoryClassName	Applicable when the provider integration mode is ClassName.	Don't use this property, UMRA will assign a default value. UMRA integration mode is JNDI.

Other sources of information

Software AG occasionally publishes information about the configuration process for UMRA in the Wiki of the Technical Community at http://techcommunity.softwareag.com/pwiki. Note that the material on the Wiki is not part of the official product documentation and is not necessarily up to date with the latest version of the product, but nevertheless you might find the information useful.

Support for XA Transactions

Universal Messaging is a non-XA resource and manages its own transactions within the boundaries of a single topic or queue, i.e. a local transaction. A local transaction allows multiple messages to be published and/or subscribed to on a single queue or topic and then committed or rolled back.

For transactional behaviour where multiple resources are involved, including non-Universal Messaging resources, it is possible to include Universal Messaging if:

- there is a single Universal Messaging step as part of the multi-part XA transaction.
- the Universal Messaging step is the first step to be committed.

In this instance the rollback or commit may not be properly honoured, as Universal Messaging does not lock the resources identified in the XA Prepare statement.

Automatic purging of individual events

If the JMS engine is enabled on a channel which uses Shared or Serial subscribers, then automatic purging of individual events that have no more subscriber interest will occur when certain conditions are met.

The frequency of the automatic purges is determined by the realm configuration property `JMSEngineAutoPurgeTime`.

To determine whether or not the JMS Engine can purge individual events from a channel, the following rules apply:

- If only Shared and/or Serial durable subscriptions are used on the channel, then all the durables of the channel will be scanned and all common events which are already processed will be purged.
- If there are no Shared or Serial durable subscriptions, then a purge of individual events cannot be done. In this case, the JMS purge engine will delete events up to the oldest event ID for which the channel has interest.
- If there is a mix of Shared/Serial durable subscriptions and other subscriptions on the channel, then a purge of individual events will be performed only for the events for which Shared or Serial durable consumers don't have interest and whose event ID is lower than the lowest event ID of the non-Shared/Serial durable consumers.

Note:
You can deactivate the JMS engine's functionality for the automatic purge of individual events by setting the value of the system property `com.softwareag.um.server.store.JMSEngineIndividualPurgeEnabled` to "false". By default, the value of this property is "true".

Basic Authentication

Client-side Authentication APIs

The API extensions for Java consist of the following new overloaded variants of existing session-creation methods.

nSessionFactory:

```
public static nSession create(nSessionAttributes attr,
   String username, String password)
```

```
public static nSession create(nSessionAttributes attr,
   nReconnectHandler handler, String username, String password)
```

```
public static nSession createMultiplexed(nSession session,
   String user, String passwd)
```

```
public static nSession createMultiplexed(nSessionAttributes sessionAttributes,
   String userName, String passwd)
```

nRealmNode:

```
Constructor -
nRealmNode(nSessionAttributes sAttr, String username, String passwd)
```

nRealmAdmin:

```
Constructor -
nRealmAdmin(nSessionAttributes sAttr, String username, String password)
```

```
Constructor -
nRealmAdmin(nSessionAttributes sAttr, String username, String password,
   boolean followMaster)
```

JNDI:

If you're using the NirvanaContextFactory class (loads the Universal Messaging provider for JMS) as the value of the standard java.naming.factory.initial context-environment key, then the standard context-environment keys java.naming.security.principal and java.naming.security.credentials should be assigned the username and password respectively.

This is standard JNDI configuration, and is compatible with all JNDI-based providers.

Code Examples

This section provides self-contained examples which include full application source code. You can use them for learning purposes or as a starting point for your own code development.

Pub/Sub Channels

Java Client: Channel Publisher

This example publishes events onto a Universal Messaging Channel.

Usage

```
npubchan <channel name> [count] [size]
<Required Arguments>
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Transactional Channel Publisher

This example publishes events transactionally to a Universal Messaging Channel. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been committed successfully.

Usage

```
npubtxchan <channel name> [count] [size] [tx size]
<Required Arguments>
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[tx size] - The number of events per transaction (default: 1)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Asynchronous Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel. See also: “ Synchronous Subscription” on page 71

Usage

```
nsubchan <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Synchronous Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel. See also: “ Asynchronous Subscription” on page 71.

Usage

```
channeliterator <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
```

```
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Asynchronous Durable Channel Consumer

This example shows how to asynchronously consume events on a Universal Messaging Channel using a durable subscription.

Usage

```
nnamedsubchan <channel name> [name] [start eid] [debug] [count] [auto ack]
                             [cluster wide] [persistent] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name]         - Specifies the unique name to be used for a durable subscription
                 (default: OS username)
[start eid]    - The Event ID to start subscribing from if the durable subscription
needs
                 to be created (doesn't exist)
[debug]        - The level of output from each event, 0 - none, 1 - summary,
                 2 - EIDs, 3 - All
[count]        - The number of events to wait before printing out summary information

                 (default: 1000)
[auto ack]     - Specifies whether each event will be automatically acknowledged by
                 the api (default: true)
[cluster wide] - Specifies whether the durable subscription is to be used across a
cluster
                 (default: false)
[persistent]   - Specifies whether the named object state is to be stored to disk or

                 held in server memory (default: false)
[priority]     - The priority of the subscriber.
[selector]     - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Synchronous Durable Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel using a durable subscription and a channel iterator.

Usage

```
nnamediterator <channel name> [name] [start eid] [debug] [count]
                              [cluster wide] [persistent] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name]         - Specifies the unique name to be used for a durable subscription
                 (default: OS username)
[start eid]    - The Event ID to start subscribing from if name subscriber
                 is to be created (doesn't already exist)
[debug]        - The level of output from each event, 0 - none, 1 - summary,
                 2 - EIDs, 3 - All
[count]        - The number of events to wait for before printing out summary
                 information (default: 1000)
[cluster wide] - Specifies whether the durable subscription is to be used across a
                 cluster (default: false)
[persistent]   - Specifies whether the durable subscription state is to be stored to

                 disk or held in server memory (default: false)
[selector]     - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: XML Channel Publisher

This example publishes XML events onto a Universal Messaging Channel

Usage

```
nxmlpub <channel name> <xml file> [count] [size]
<Required Arguments>
<channel name> - Channel name parameter for the channel to publish to
<xml file> - The full path of the xml file to publish
[Optional Arguments]
[count] -The number of events to publish (default: 10)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Asynchronous XML Channel Consumer

This example shows how to asynchronously subscribe to XML events on a Universal Messaging Channel.

Usage

```
nxmlsub <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Event Delta Delivery

This example shows how to publish and receive registered events.

Usage

```
RegisteredEvent <rname> <channel name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
```

Application Source Code

See the online documentation for a code example.

Java Client: Batching Server Calls

This example shows how to find multiple channels and queues in one call to the server.

Usage

```
findChannelsAndQueues <name> <name> <name>.....
<Arguments>
<name> - The name(s) of the channels to find
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Batching Subscribe Calls

This example of batching shows how to subscribe to multiple Universal Messaging Channels in one server call.

Usage

```
sessionsubscriber <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<channel names> - Comma separated list of channels to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Pub/Sub Data Groups

Java Client: DataStream Listener

This example shows how to initialise a session with a DataStream listener and start receiving data.

Usage

```
DataStreamListener [debug] [count]
<Required Arguments>
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

Java Client: DataGroup Publishing with Conflation

This example shows how to publish to DataGroups, with optional conflation.

Usage

```
DataGroupPublish <group name> [count] [size] [enable multicast] [conflate]
                              [conflation merge or drop] [conflation interval]
<Required Arguments>
```

```
<group name> - Data group name parameter to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[enable multicast] - enable the data group for multicast delivery
[conflate] - enable conflation true or false
[conflation merge or drop] - merge to enable merge or drop to enable drop
                            (default: merge)
[conflation interval] - the interval for conflation to publish(default: 500
```

Application Source Code

See the online documentation for a code example.

Java Client: DataGroup Manager

This is an example of how to run a DataGroup manager application

Usage

```
dataGroupsManager <Properties File Location>
<Required Arguments>
<Properties File Location Data Groups> - The location of the property file to
      use for mapping data groups to data groups
<Properties File Location Data Streams> - The location of the property file to
      use for mapping data streams to data groups
<Auto Recreate Data Groups> - True or False to auto recreate data groups takes
      the data group property file and creates channels
      a group for every name mentioned on the left of equals sign
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Delete DataGroup

This is a simple example of how to delete a DataGroup

Usage

```
deleteDataGroups <data group name> <delete type>
<Required Arguments>
<data group name> - Data group name parameter to delete
<Delete Type> - Data group delete by string(1) or object(2)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: DataGroup Delta Delivery

This example shows how to use delta delivery with DataGroups.

Usage

```
DataGroupDeltaDelivery [count]
[Optional Arguments]
[count] - the number of times to commit the registered events
```

Application Source Code

See the online documentation for a code example.

Message Queues

Java Client: Queue Publisher

This example publishes events onto a Universal Messaging Queue.

Usage

```
npushq <queue name> [count] [size]
<Required Arguments>
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Transactional Queue Publisher

This example publishes events transactionally to a Universal Messaging Queue. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been committed successfully.

Usage

```
npushtxq <queue name> [count] [size] [txsize]
<Required Arguments>
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[txsize] - The number of events to publish per transaction (default: 1)
```

```
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Asynchronous Queue Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Queue. See also: " Synchronous Queue Subscription" on page 79.

Usage

```
npopqasync <queue name> [debug] [count] [selector]
<Required Arguments>
<queue name> - Queue name parameter for the queue to pop from
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Asynchronous Transactional Queue Consumer

This example shows how to transactionally asynchronously subscribe to events on a Universal Messaging Queue. See also: " Synchronous Queue Subscription." on page 79

Usage

```
npoptxqasync <queue name> [debug] [count] [selector]
<Required Arguments>
<queue name> - Queue name parameter for the queue to pop from
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Synchronous Queue Consumer

This example shows how to synchronously consume events from a Universal Messaging Queue. See also: " Asynchronous Queue Subscription" on page 78.

Usage

```
npopq <queue name> [timeout] [debug] [count] [selector]
<Required Arguments>
<queue name> - Queue name parameter for the queue to pop from
[Optional Arguments]
[timeout] - The timeout for the dequeue operation
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Synchronous Transactional Queue Consumer

This example shows how to synchronously consume events from a Universal Messaging Queue. See also: " Asynchronous Queue Subscription" on page 78.

Usage

```
npoptxq <queue name> [timeout] [debug] [count] [selector]
<Required Arguments>
<queue name> - Queue name parameter for the queue to pop from
[Optional Arguments]
[timeout] - The timeout for the dequeue operation
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Peek events on a Queue

This example shows how to peek events on a Universal Messaging Queue. See also: " Asynchronous Queue Subscription" on page 78.

Usage

```
npeekq <queue name> [debug] [count] [selector]
<Required Arguments>
<queue name> - Queue name parameter for the queue to peek
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Requester - Request/Response

This example shows how to request a response in a request/response fashion.

Usage

```
requester <request queue> <request queue>
<Required Arguments>
<request queue>  - Queue onto which request are published
<response queue> - Queue onto which responses are published
[Optional Arguments]
[asynchronous]  - Whether to use asynchronous producing and consuming
                  true/false, default false.
[transactional] - Whether to use transactional production and consumption of
                  events - true/false, default false.
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Responder - Request/Response

This example shows how to respond to a request in performed in a request/response fashion.

Usage

```
responder <request queue> <response queue>
<Required Arguments>
<request queue>  - Queue onto which request are published
<response queue> - Queue onto which responses are published
[Optional Arguments]
[asynchronous]  - Whether to use asynchronous producing and consuming
                  true/false, default false.
[transactional] - Whether to use transactional production and consumption of
                  events - true/false, default false.
```

```
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Administration API

Java Client: Add a Queue ACL Entry

This example demonstrates how to add an ACL entry to a Universal Messaging Queue.

Usage

```
naddqueueacl <queue name> <user> <host> [list_acl] [modify_acl]
                                        [full] [peek] [push] [purge] [pop]
<Required Arguments>
<queue name> - Queue name parameter for the queue to add the ACL entry to
<user> - User name parameter for the queue to add the ACL entry to
<host> - Host name parameter for the queue to add the ACL entry to
[Optional Arguments]
[list_acl] - Specifies that the list acl permission should be added
[modify_acl] - Specifies that the modify acl permission should be added
[full] - Specifies that the full permission should be added
[peek] - Specifies that the peak permission should be added
[push] - Specifies that the push permission should be added
[purge] - Specifies that the purge permission should be added
[pop] - Specifies that the pop permission should be added
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Modify a Channel ACL Entry

This example demonstrates how to modify the permissions of an ACL entry on a Universal Messaging Channel.

Usage

```
nchangechanacl <channel name> <user> <host> [+/-list_acl] [+/-modify_acl]
    [+/-full] [+/-last_eid] [+/-read] [+/-write] [+/-purge]
    [+/-named] [+/-all_perms]
<Required Arguments>
<channel name> - Channel name parameter for the channel to change the ACL entry for
<user> - User name parameter for the channel to change the ACL entry for
<host> - Host name parameter for the channel to change the ACL entry for
[Optional Arguments]
[+/-] - Prepending + or - specifies whether to add or remove a permission
[list_acl] - Specifies that the list acl permission should be added/removed
[modify_acl] - Specifies that the modify acl permission should be added/removed
```

```
[full] - Specifies that the full permission should be added/removed
[last_eid] - Specifies that the get last EID permission should be added/removed
[read] - Specifies that the read permission should be added/removed
[write] - Specifies that the write permission should be added/removed
[purge] - Specifies that the purge permission should be added/removed
[named] - Specifies that the used named subscriber permission should be added/removed
[all_perms] - Specifies that all permissions should be added/removed
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Delete a Realm ACL Entry

This example demonstrates how to delete an ACL entry from a realm on a Universal Messaging Channel.

Usage

```
ndelrealmacl <user> <host> [-r]
<Required Arguments>
<user> - User name parameter to delete the realm ACL entry from
<host> - Host name parameter to delete the realm ACL entry from
[Optional Arguments]
[-r] - Specifies whether recursive traversal of the namespace should be done
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Add a Schedule to a Universal Messaging Realm

This example demonstrates how to read a schedule from a file and add the schedule to a realm.

Usage

```
naddschedule <source> [subject] [clusterwide]
<Required Arguments>
<source> - location of the schedule script file
[Optional Arguments]
[subject] - The subject of the schedule (default : os username)
[clusterwide] - Whether or not the schedule is cluster wide (default : false)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Simple authentication server

This demonstrates how to set security permissions when connection attempts are made on the realm.

Application Source Code

See the online documentation for a code example.

Java Client: Monitor realms for cluster creation, and cluster events

This example demonstrates how to monitor a realm or realms for cluster events.

Application Source Code

See the online documentation for a code example.

Java Client: Monitor realms for client connections coming and going

This example demonstrates how to monitor for connections to the realm and its channels.

Usage

```
nconnectionwatch
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Copy a channel and its attributes

This example demonstrates how to copy a given channel and its attributes onto the same realm or a different realm, thereby creating a clone of the original channel. Note that events on the original channel are not copied to the new channel.

Usage

```
nadmincopychan <channel> [-r toRealm] [-n toChannelName] [-a channel ttl]
                         [-c channel capacity] [-t channel type]
<Required Arguments>
<channel> - Channel name parameter for the channel to copy
[Optional Arguments]
<-r toRealm>          - The name of the linked remote realm to copy the channel to
<-n toChannelName>    - The name you wish to give the copied channel
<-a channel ttl>      - The ttl you wish to give the copied channel
<-c channel capacity> - The capacity you wish to give the copied channel
<-t channel type>     - The channel you wish the copied channel to be any of
                       (P | R | M | S | T)
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Monitor the remote realm log and audit file

This example demonstrates how to monitor a realm's log and audit files.

Usage

```
nauditandloglistener <-l logfile> <-a auditfile> <-replay>
[Optional Arguments]
<-l logfile>    - A file name to store the log messages to (without this it
                  will go to system.err
<-a auditfile> - A file name to store the audit messages to (without this it
                  will go to system.err
<-replay>       - Specifies if the entire audit file will be replayed
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Export a realm to XML

This example demonstrates how to export a realm's cluster, joins, security, channels / queues, scheduling, interfaces / plugins, configuration information and JNDI assets to an XML file so that it can be imported into any other realm (see "Java Client: Import a realm's configuration information" on page 84).

Usage

```
nexportrealmxml [export_file_location]
<Optional Arguments>
    -all -realms -cluster -realmacl -realmcfg -channels -jndi
    -channelfilter=<filter> -channeacls -datagroups -datagroupfilter=<filter>
    -joins -queues -queuefilter=<filter> -queueacls -interfaces -plugins -via
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Import a realm's configuration information

This example demonstrates how to import a realm's cluster, joins, security, channels / queues, scheduling, interfaces / plugins, configuration information and JNDI assets from an XML file that was previously created by exporting a realm (see "Java Client: Export a realm to XML" on page 84).

Usage

```
nimportrealmxml file_name
<Optional Arguments>
    -all -realmacl -realmcfg -channels -durables -jndi -channelfilter=<filter>
    -channelacls -queues -queuefilter=<filter> -queueacls -interfaces
    -datagroups -datagroupfilter=<filter>
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Console-based Realm Monitor

This example demonstrates how to monitor a realm's cluster, joins, security, channels / queues, scheduling, interfaces / plugins and configuration information.

Usage

```
nTop [refreshRate]
[Optional Arguments]
[refreshRate] - the rate at which the information is reloaded on screen (milliseconds)

Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Realm Monitor

Monitors a Universal Messaging Realm and output results to CSV files.

Usage

```
java RealmMonitor <rnames> [config file]
<Required Parameters>
<rname> : comma separated list of rnames to monitor.
[Optional Parameters]
[config file] : configuration file location e.g. c:\\config.txt
All other parameters can be specified in the config file.
If realm is clustered then other realms in cluster will
be found automatically.
```

Application Source Code

See the online documentation for a code example.

Description of Classes and Output

Contained within the application are 5 separate classes each set up to monitor different aspects of a Universal Messaging Realm:

- "Connection Monitor" on page 87 - monitors all connections to the realm
- "Realm Monitor" on page 87 - regularly outputs current state of the realm
- "Thread Monitor" on page 87 - monitors for unexpected thread behaviour
- "Log Monitor" on page 88 - listens for key words in the Realm log
- "Channel Monitor" on page 88 - monitors channel and queue state for potential issues

The application takes 2 arguments:

1. RNAMEs

   Comma separated list of RNAMEs of the realms to monitor.

   If the realm is part of a cluster, the application will also monitor the cluster members so there is no need to list the RNAMEs of the other cluster members.

2. Configuration file

   [optional] the location of the configuration file (including the name of the file).

   If this is not specified, a default file will be created.

Each monitor will output different information and at different intervals. The information to be output can be specified in the 'methods' field in the configuration file for each monitor. The data is written to CSV files so that it can be easily plugged into graphing tools or other monitoring applications. Most of the monitors will only output data if a certain condition is met for example the log monitor will only write data if a keyword is found in a line of the Universal Messaging Realm log. The realm monitor on the other hand will constantly output data to the CSV every X seconds. If a realm goes down, the realm monitor will output 0s which will make any problems apparent if the data is put into a graph.

Configuration File

The configuration file provides flexibility to change when and what data is output for each monitor. The application will generate a default configuration file if no command line argument is specified. This default file contains all the necessary parameters to start monitoring but all parameters are customisable. Each monitor has different triggers for when to output data. These triggers can all be specified in this file. More detail on the monitor specific parameters can be found in the relevant sections below. You can also specify the working directory for the tests [default: ./RealmMonitorOut/] and which tests to run.

'methods' parameter

Each monitor requires this parameter to be set in the configuration file. This is a comma separated list of method names to be invoked on the object that the monitor is observing. The result of each of these methods will be output to the CSV file under the corresponding heading in the 'headings' parameter.

For example the connection monitor is monitoring connections so will invoke the methods on the nConnectionDetails object. You can specify any methods here that are a member of the nConnectionDetails class. It is possible that the return type of these methods will not be a type that is easily represented as a string or you may wish to display the object in a certain way. Most of these cases have been dealt with already but in order to change the behaviour you can simply override the method name in the relevant monitor class. For example the channel monitor invokes the methods on nLeafNode which has a method called getUsedSpace. getUsedSpace returns a long representing the number of bytes used, however this application will return the used space in kilobytes because the method is overridden:

```
public String getUsedSpace(Object o){
  long used = ((nLeafNode)o).getUsedSpace();
  used = used/1024;
  return used+"";
}
```

The above method overrides the getUsedSpace method on the nLeafNode. By simply creating this method inside the ChannelMonitor class, whenever getUsedSpace is required, this method will be called instead with the nLeafNode as a parameter.

Connection Monitor

The connection monitor maintains a list of the current connections to the realm. Every x seconds the monitor will check each connection for potential problems and write the details of that connection to the CSV file if any trigger is hit.

There are two triggers currently available:

- maxQueuedEvents - maximum number of events allowed to be queued for a connection before the monitor will output to the CSV
- maxTimeOfLastTransmit - maximum time taken to transmit the last event.

This monitor invokes the method in the 'methods' parameter on the nConnectionDetails object.

Realm Monitor

The realm monitor prints to the CSV file every x seconds (configurable by the refreshRate parameter). There are no triggers for this monitor as the information should be available at all times. If no data is available (realm is down) then by default the monitor will output 0s to the CSV.

This monitor invokes the method in the 'methods' parameter on the nRealmNode object.

Thread Monitor

The thread monitor constantly checks the thread pools for unexpected values. There are 2 triggers which will cause the monitor to write to the CSV:

- maxQueueSize - if the number of queued tasks in the thread pool exceeds this value then the information on this tread pool will be output to the CSV
- minIdle - the minimum number of idle threads available before the monitor will output to the CSV

This monitor invokes the method in the 'methods' parameter on the nThreadPool object.

Log Monitor

This monitor listens to the Universal Messaging log and will write to the CSV whenever any line of the log contains a keyword. The list of keywords can be specified in the config file.

The 'methods' parameter in the configuration file will be invoked on the LogMonitor class or the enclosing RealmMonitor class e.g. getTime.

Channel Monitor

The channel monitor keeps track of the current channels and queues on the realm. If any triggers are hit for a leaf node then the details of that leaf node are output to the CSV.

There are many triggers that can be set for this monitor. For example there is a parameter called 'minCurrentCons' and an associated 'maxCurrentCons'. If the number of current connections falls outside of this range then the details of the leaf node are printed to the CSV.

This monitor invokes the method in the 'methods' parameter on the nLeafNode object.

Java Client: Create Cluster

This example demonstrates how to create a cluster.

Usage

```
nmakecluster <cluster name> <convert local stores> <rnames> [-r]
<Required Arguments>
<cluster name> -
    The name for the new cluster.
    The cluster name must be alphanumeric.
<convert local stores> -
    Flag to indicate whether the local stores of the master
    should be converted to cluster wide stores.
<rnames> -
    Server URLs to be included in the cluster.
    There can be several names, separated by a space.
    The proper format is:
    [nsp/nhp/nsps/nhps]://[hostname]:[port]
    or
    shm://[path/to/file]
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Create Security Group

This example demonstrates how to create a security group.

Usage

```
nmakesecgroup <security group name> [-r]
<Required Arguments>
<security group name> - The name of the security group
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Add Security Group Subject

This example demonstrates how to add a subject to a security group.

Usage

```
naddsecgrpsubject <security group name> <subject> [-r]
<Required Arguments>
<security group name> - The name of the security group.
<subject> - The subject to be added, in the format "user@host".
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Delete Security Group Subject

This example demonstrates how to delete a subject from a security group.

Usage

```
ndelsecgrpsubject <security group name> <subject> [-r]
<Required Arguments>
<security group name> - The name of the security group.
<subject> - The subject to be removed, in the format "user@host".
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Delete Security Group

This example demonstrates how to delete a security group.

Usage

```
ndelsecgroup <security group name> [-r]
<Required Arguments>
<security group name> - The name of the security group.
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Provider for JMS

Using the AMQP Protocol

The JMS sample applications (with the exception of the JMS Queue Browser) can be used over the Advanced Message Queuing Protocol (AMQP) protocol. AMQP is defined by the ISO/IEC 19464 standard.

In order to switch to AMQP, you need to change the following environment variables, prior to running the JMS sample application:

Environment variable	Description
CONTEXT_FACTORY	The name of the ContextFactory class used by the sample application (default: com.pcbsys.nirvana.nSpace.NirvanaContextFactory). When using an AMQP connection this should be changed to: ■ org.apache.qpid.jms.jndi.JmsInitialContextFactory - to use the QPID Proton JMS Client libraries, or ■ org.apache.qpid.amqp_1_0.jms.jndi.PropertiesFileInitialContextFactory - to use the QPID Legacy JMS Client libraries
PROVIDER_URL	The URL of your local nirvana realm from which JNDI entries will be looked up (default: `nsp://localhost:9000`). When using an AMQP connection this should be changed to: ■ `amqp://localhost:9000` - when using a plain AMQP connection, connecting to a standard nsp interface; ■ `amqps://localhost:9000` - when using a plain AMQP connection over a secure socket connection, connecting to a nsps interface.

Note:

AMQP connections require the AMQP plugin to be enabled on the realm and work only over the nsp and nsps interfaces. The connections will not work over nhp or nhps.

Note:
When using AMQP with all JMS sample applications, the `<factoryname>` parameter is ignored, but it is still required. This is due to the internal implementation of the client libraries.

Plain AMQP pub/sub example

The following example assumes that you have an existing topic `MyTopic` configured on the realm server and bound in the JNDI context, and that the client tries to connect to a NSP interface.

To start a subscriber:

1. Start a "Java examples" command prompt
2. Set the following environment variables:

```
set CONTEXT_FACTORY=org.apache.qpid.jms.jndi.JmsInitialContextFactory
set PROVIDER_URL=amqp://<hostname>:<port>
```

3. Start the `jmssub` application by running the following command:

```
jmssub ignored topic.MyTopic
```

At this point the `jmssub` application will be running over AMQP and will receive any messages published on the `MyTopic` topic

To start a publisher:

1. Start a "Java examples" command prompt
2. Set the following environment variables:

```
set CONTEXT_FACTORY=org.apache.qpid.jms.jndi.JmsInitialContextFactory
set PROVIDER_URL=amqp://<hostname>:<port>
```

3. Start the `jmssub` application by running the following command:

```
jmspub ignored topic.MyTopic 10
```

At this point the `jmspub` application will be running over AMQP and will send 10 messages to the `MyTopic` topic.

AMQP over alternative TLS example

The following example assumes that you have an existing topic `MyTopic` configured on the realm server and bound in the JNDI context and that the client tries to connect to a NSPS interface.

To start a subscriber:

1. Start a "Java examples" command prompt

2. Set the following environment variables:

```
set CONTEXT_FACTORY=org.apache.qpid.jms.jndi.JmsInitialContextFactory
set PROVIDER_URL=amqps://<hostname>:<port>
```

3. Start the `jmssub` application by running the following command:

```
jmssub ignored topic.MyTopic
```

At this point the `jmssub` application will be running over AMQP and will receive any messages published on the `MyTopic` topic

To start a publisher:

1. Start a "Java examples" command prompt
2. Set the following environment variables:

```
set CONTEXT_FACTORY=org.apache.qpid.jms.jndi.JmsInitialContextFactory
set PROVIDER_URL=amqps://<hostname>:<port>
```

3. Start the `jmssub` application by running the following command:

```
jmspub ignored topic.MyTopic 10
```

At this point the `jmspub` application will be running over AMQP and will send 10 messages to the `MyTopic` topic.

Note:
If the NSPS interface on the realm has been configured to use the certificates generated by the "CertificateGenerator" then you don't need to make any changes to the client applications. However if you are using different certificates, then in each command prompt, you need to additionally set the certificate locations by executing the following set commands:

```
set CAKEYSTORE=<TRUST KEYSTORE PATH>
set CAKEYSTOREPASSWD=<TRUST KEYSTORE PASSWORD>
set CKEYSTORE=<CLIENT KEYSTORE PATH>
set CKEYSTOREPASSWD=<CLIENT KEYSTORE PASSWORD>
```

Java Client: JMS BytesMessage Publisher

This example uses Universal Messaging Provider for JMS to publish Bytes Messages to a JMS Topic.

Usage

```
jmsbytespub <factoryname> <topicName> <count> <transacted>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm).
                If you are using AMQP, this argument is ignored
                but it is still required.
<topicName>   - JMS Topic to publish on.
                When using AMQP, this should be in the
                format topic.<topicName>
<count>       - Number of events to publish
<transacted>  - Whether the session is transacted
```

```
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS BytesMessage Subscriber

This example uses Universal Messaging Provider for JMS to consume Bytes Messages from a JMS Topic.

Usage

```
jmsbytessub <factoryname> <destinationName> <transacted> <durablename> <selector>
<Required Arguments>
<factoryname>     - JMS Factory (Must exist in target realm).
                    If you are using AMQP, this argument is ignored
                    but it is still required.
<destinationName> - JMS Destination to subscribe to.
                    When using AMQP, this should be in the
                    format topic.<topicName> or queue.<queueName>
<transacted>      - Whether the session is transacted
<durablename>     - The name of a durable subscriber
<selector>        - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS MapMessage Publisher

This example uses Universal Messaging Provider for JMS to publish Map Messages to a JMS Topic.

Usage

```
jmsmappub <factoryname> <topicName> <count> <transacted>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm).
                If you are using AMQP, this argument is ignored
                but it is still required.
<topicName>   - JMS Topic to publish on.
                When using AMQP this should be in the
                format topic.<topicName>
<count>       - Number of events to publish
<transacted>  - Whether the session is transacted
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS MapMessage Subscriber

This example uses Universal Messaging Provider for JMS to consume Map Messages from a JMS Topic.

Usage

```
jmsmapsub <factoryname> <destinationName> <transacted> <selector>
<Required Arguments>
<factoryname>     - JMS Factory (Must exist in target realm).
                    If you are using AMQP, this argument is ignored
                    but it is still required.
<destinationName> - JMS Destination to subscribe to.
                    When using AMQP this should be in the
                    format topic.<topicName> or queue.<queueName>
<transacted>      - Whether the session is transacted
<selector>        - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS ObjectMessage Publisher

This example uses Universal Messaging Provider for JMS to publish Object Messages to a JMS Topic.

Usage

```
jmsobjectpub <factoryname> <topicName> <count> <transacted>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm).
                If you are using AMQP, this argument is ignored
                but it is still required.
<topicName>   - JMS Topic to publish on.
                When using AMQP this should be in the
                format topic.<topicName>
<count>       - Number of events to publish
<transacted>  - Whether the session is transacted
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS ObjectMessage Subscriber

This example uses Universal Messaging Provider for JMS to consume Object Messages from a JMS Topic.

Usage

```
jmsobjectsub <factoryname> <destinationName> <transacted> <selector>
<Required Arguments>
<factoryname>     - JMS Factory (Must exist in target realm).
                    If you are using AMQP, this argument is ignored
                    but it is still required.
<destinationName> - JMS Destination to subscribe to.
                    When using AMQP this should be in the
                    format topic.<topicName> or queue.<queueName>
<transacted>      - Whether the session is transacted
<selector>        - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS StreamMessage Publisher

This example uses Universal Messaging Provider for JMS to publish Stream Messages to a JMS Topic.

Usage

```
jmsstreampub <factoryname> <topicName> <count> <transacted>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm).
                If you are using AMQP, this argument is ignored
                but it is still required.
<topicName>   - JMS Topic to publish on.
                When using AMQP this should be in the
                format topic.<topicName>
<count>       - Number of events to publish
<transacted>  - Whether the session is transacted
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS StreamMessage Subscriber

This example uses Universal Messaging Provider for JMS to consume Stream Messages from a JMS Topic.

Usage

```
jmsstreamsub <factoryname> <destinationName> <transacted> <selector>
<Required Arguments>
<factoryname>     - JMS Factory (Must exist in target realm)
                    If you are using AMQP, this argument is ignored
```

```
                    but it is still required.
<destinationName> - JMS Destination to subscribe to.
                    When using AMQP this should be in the
                    format topic.<topicName> or queue.<queueName>
<transacted>      - Whether the session is transacted
<selector>        - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS BytesMessage Queue Publisher

This example uses Universal Messaging Provider for JMS to publish Bytes Messages to a JMS Queue.

Usage

```
jmsbytesqpub <factoryname> <queueName> <count> <transacted>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm).
                If you are using AMQP, this argument is ignored
                but it is still required.
<queueName>   - JMS Queue to publish on.
                When using AMQP this should be in the
                format queue.<queueName>
<count>       - Number of events to publish
<transacted>  - Whether the session is transacted
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS BytesMessage Queue Subscriber

This example uses Universal Messaging Provider for JMS to consume Bytes Messages from a JMS Queue.

Usage

```
jmsbytesqsub <factoryname> <destinationName> <transacted> <selector>
<Required Arguments>
<factoryname>     - JMS Factory (Must exist in target realm).
                    If you are using AMQP, this argument is ignored
                    but it is still required.
<destinationName> - JMS Destination to subscribe to.
                    When using AMQP this should be in the
                    format topic.<topicName> or queue.<queueName>
<transacted>      - Whether the session is transacted
<selector>        - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: JMS Queue Browser

This example shows how to browse a Universal Messaging Provider for JMS Queue in JMS.

Usage

```
jmsqbrowse <factoryname> <destinationName> <selector>
<Required Arguments>
<factoryname> - JMS Factory (Must exist in target realm)
<destinationName> - JMS Destination to subscribe to
<selector> - An optional message selector
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Channel / Queue / Realm Management

Java Client: Creating a Channel

This example demonstrates how to create a Universal Messaging channel programmatically.

Usage

```
nmakechan <channel name> [time to live] [capacity] [type] [cluster wide]
   [start eid] [use jms engine] [honor capacity when full]
<Required Arguments>
<channel name> - Channel name parameter for the channel to be created.
[Optional Arguments]
[time to live] - The Time To Live parameter for the new channel (default: 0)
[capacity] - The Capacity parameter for the new channel (default: 0)
[type] - The type parameter for the new channel (default: S)
         R - For a reliable (stored in memory) channel with persistent eids
         P - For a persistent (stored on disk) channel
         M - For a Mixed (allows both memory and persistent events) channel
[cluster wide] - Whether the channel is cluster wide. Will only work if the
     realm is part of a cluster.
[start eid] - The initial start event id for the new channel (default: 0).
[use jms engine] - Sets whether to use the JMS style fanout engine.
[honor capacity when full] - Whether the channel / queue capacity setting will
    prevent publishing of any more data once full.
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Deleting a Channel

This example demonstrates how to delete a Universal Messaging channel programmatically.

Usage

```
ndelchan <channel name>
<Required Arguments>
<channel name> - Channel name parameter for the channel to delete
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Creating a Queue

This example demonstrates how to create a Universal Messaging queue programmatically.

Usage

```
nmakeq <queue name> [time to live] [capacity] [type] [cluster wide]
<Required Arguments>
<queue name> - Queue name parameter for the queue to be created
[Optional Arguments]
[time to live] - The Time To Live parameter for the new queue (default: 0)
[capacity] - The Capacity parameter for the new queue (default: 0)
[type] - The type parameter for the new queue (default: S)
R - For a reliable (stored in memory) queue with persistent eids
P - For a persistent (stored on disk) queue
M - For a Mixed (allows both memory and persistent events) queue
[cluster wide] - Whether the queue is cluster wide. Will only work if the
                 realm is part of a cluster
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Deleting a Queue

This example demonstrates how to delete a Universal Messaging queue programmatically.

Usage

```
ndelq <queue name>
<Required Arguments>
<queue name> - Queue name parameter for the channel to delete
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Create a Channel Join

This is a class that demonstrates how to create a channel join.

Usage

```
nmakechanjoin <source channel name> <destination channel name>
              [max hops] [selector]
<Required Arguments>
<source channel name>      - Channel name parameter of the local channel name
                             to join
<destination channel name> - Channel name parameter of the remote channel name
                             to join
[Optional Arguments]
[max hops]    - The maximum number of join hops a message can travel through
[selector]    - The event filter string to use on messages travelling through
                this join
[Allow Purge] - If allow purge is true then when the source channel is purged
                events will also be purged
[archive]     - true/false, defaults to false, set if you wish to perform an
                archive join to a queue
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Delete a Channel Join

This is a class that demonstrates how to delete a channel join.

Usage

```
ndelchanjoin <source channel name> <destination channel name>
<Required Arguments>
<source channel name> - Source Channel name parameter of the join to be deleted
<destination channel name> - Destination Channel name parameter of the join to
                             be deleted
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Purge events from a channel

This class demonstrates how to purge events from a channel.

Usage

```
npurgechan <channel name> <start eid> <end eid> <filter>
<Required Arguments>
<channel name> - Channel name parameter for the channel to be purged
<start eid> - The start eid of the range of events to be purged
<end eid> - The end eid of the range of events to be purged
<filter> - An optional filter string for events to be purged
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Find the event id of the last event

This class demonstrates how to find the last event id published on a specific channel.

Usage

```
ngetlasteid <channel name>
<Required Arguments>
<channel name> - Channel name parameter to get the last EID for
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Add a realm to another realm

This is a class that demonstrates how to add a realm to another realm, either mounted into the namespace or not.

Usage

```
naddrealm <realm name> <realm details> [mount point]
<Required Arguments>
<realm name> - Realm name parameter for the realm to add
<realm details> - Realm details parameter for the realm to add.
                  Same form as RNAME
[Optional Arguments]
[mount point] - Where you would like to mount the realm within the namespace,
                for example /eur/uk
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: Multiplex a Session

Multiplex two Universal Messaging sessions over one channel.

Usage

```
nsubchan <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Client API Package Documentation

API documentation is available for the following Client API:

- Java Client API: com.pcbsys.nirvana.client

The API documentation is available in the *Universal Messaging Reference Guide* section of the documentation.

Enterprise Developer's Guide for C++

This guide describes how to develop and deploy C++ applications using Universal Messaging, and assumes you already have Universal Messaging installed.

General Features

Creating a Session

To interact with a Universal Messaging Server, the first thing to do is create a Universal Messaging Session (nSession) object, which is effectively your logical and physical connection to a Universal Messaging Realm.

Creating a Universal Messaging Session Object

1. Create a nSessionAttributes object with the RNAME value of your choice

   ```
   std::string[] RNAME={"nsp://127.0.0.1:9000"};
   int length = 1;
   ```

```
nSessionAttributes *nsa=new nSessionAttributes(RNAME,length)
```

2. Call the create method on nSessionFactory to create your session

```
Session *mySession = nSessionFactory::create(nsa);
```

Alternatively, if you require the use of a session reconnect handler to intercept the automatic reconnection attempts, pass an instance of that class too in the create method:

```
class myReconnectHandler :
    public nReconnectHandler
     {
     //implement tasks associated with reconnection
     }
 myReconnectHandler rhandler=new myReconnectHandler();
 nSession *mySession=nSessionFactory::create(nsa, rhandler);
```

Initializing a Universal Messaging Session

1. Initialise the session object to open the connection to the Universal Messaging Realm

```
mySession->init();
```

Universal Messaging Events

A Universal Messaging Event (nConsumeEvent) is the object that is published to a Universal Messaging channel or queue. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an Event Dictionary (see "Event Dictionaries" on page 102).

Each nConsumeEvent object has an nEventAtrributes object associated with it which contains all available meta data associated with the event.

Constructing an Event

In this C++ code snippet, we construct our Universal Messaging Event object (`nConsumeEvent`), and, in this example, pass a byte array data into the constructor:

```
std::string strLine = "Hello World";
int length = 0;
unsigned char *pLine = nConstants::encode(strLine, length);
nEventProperties *pProps = new nEventProperties();
nConsumeEvent *evt = new nConsumeEvent(pProps, pLine, length);
```

Event Dictionaries

Universal Messaging Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within an event (for related information, see "Universal Messaging Events" on page 102).

Event Dictionaries are quite similar to a hash table, supporting primitive types, arrays, and nested dictionaries.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's dictionary.

Event dictionaries can facilitate the automated purging of data from channels through the use of Publish Keys.

Constructing an Event

In this code snippet, we assume we want to publish an event containing the definition of a bond, say, with a name of "bond1":

```
nEventProperties *props = new nEventProperties();
props->put("bondname", "bond1");
props->put("price", 100.00);
nConsumeEvent *evt = new nConsumeEvent(props, "atag");
channel->publish(evt);
```

Note that in this example code, we also create a new Universal Messaging Event object (`nConsumeEvent`, see "Universal Messaging Events" on page 102) to make use of our Event Dictionary (`nEventProperties`).

Channel Joins

Joining a channel to another channel or queue allows you to set up content routing such that events published to the source channel will be passed on to the destination channel/queue automatically. Joins also support the use of *filters* thus enabling dynamic content routing.

Please note that while channels can be joined to both resources, queues cannot be used as the source of a join.

When creating a join there is one compulsory parameter and two optional ones. The compulsory parameter is the destination channel. The optional parameters are the maximum join hops and a filter to be applied to the join.

Creating Channel Joins

Universal Messaging joins are created as follows:

```
//Obtain a reference to the source channel
nChannel *mySrcChannel = mySession->findChannel( nca );
//Obtain a reference to the destination channel
nChannel *myDstChannel = mySession->findChannel( dest );
//Obtain a reference to the destination channel's realm
nRealm *realm = myDstChannel->getChannelAttributes()->getRealm();
//create the join
mySrcChannel->joinChannel( myDstChannel, true, jhc, SELECTOR );
```

For a sample application please see the section *C++ Client: Create Channel Join* in the online documentation.

Deleting Channel Joins

Channel joins can also be deleted. Please see the sample application *C++ Client: Delete a Channel Join* in the online documentation.

Related Links

For a conceptional overview of channel joins, see the section *Data Routing using Channel Joins* in the *Concepts* guide.

For a description of how to set up and manage channel joins, see the section *Creating Channel Joins* in the *Administration Guide*. The description details the usage based on the Enterprise Manager, but the same general principles apply if you are using the API.

Google Protocol Buffers

Overview

Google Protocol Buffers are a way of efficiently serializing structured data. They are language and platform neutral and have been designed to be easily extensible. The structure of your data is defined once, and then specific serialization and deserialization code is produced specifically to handle your data format efficiently.

For general details on using Google protocol buffers, see the section *Google Protocol Buffers* in the *Concepts* guide.

Using Google Protocol Buffers with Universal Messaging

To create a nProtobuf event, simply build your protocol buffer as normal and pass it into the nProtobuf constructor along with the message type used.

nProtobuf events are received by subscribers in the normal way.

The Enterprise Manager can be used to view, edit and republish protocol buffer events, even if the Enterprise Manager is not running on the same machine as the server. The Enterprise Manager is able to parse the protocol buffer message and display the contents, rather than the binary data.

All descriptors will be automatically synced across the cluster if the channel is cluster-wide.

Publish / Subscribe using Channel Topics

Publish / Subscribe Using Channels/Topics

The Universal Messaging C++ API provides publish subscribe functionality through the use of channel objects. Channels are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Universal Messaging Data Streams and Data Groups provide an alternative style of Publish/Subscribe where user subscriptions can be managed remotely on behalf of clients.

Under the publish / subscribe paradigm, each event is delivered to each subscriber once and only once per subscription, and is not removed from the channel after being consumed.

This section demonstrates how Universal Messaging pub / sub works.

Creating a Channel

Channels can be created programmatically as detailed below, or they can be created using the Enterprise Manager.

In order to create a channel, first of all you must create an nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using an RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
std::string[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nSession *mySession = nSessionFactory::create(nsa);
mySession->init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then begin creating the channel object. Channels have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them,

To create a channel, we do the following:

```
nChannelAttributes *cattrib = new nChannelAttributes();
cattrib->setMaxEvents(0);
cattrib->setTTL(0);
cattrib->setType(nChannelAttributes::PERSISTENT_TYPE);
cattrib->setName("mychannel");
nChannel *myChannel = mySession->createChannel(cattrib);
```

Now we have a reference to a Universal Messaging channel within the realm.

Note:
The set of permissible characters you can use to name a channel is described in the section *Creating Channels* in the Enterprise Manager section of the *Administration Guide*.

Finding a Channel

In order to find a channel programmatically you must create your nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using the correct RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
std::string* RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nSession *mySession = nSessionFactory::create(nsa);
mySession->init();
```

Once the nSession->init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the channel object. Channels have an associated set of attributes, that define their behavior within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them,

To find a channel previously created, we do the following:

```
nChannelAttributes *cattrib = new nChannelAttributes();
cattrib->setName("mychannel");
nChannel *myChannel = mySession->findChannel(cattrib);
```

This returns a reference to a Universal Messaging channel within the realm.

Behavior if a channel has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "CHANNEL DELETED" and an event data component equal to "The channel has been deleted".

How to publish events to a Channel

There are 2 types of publish available in Universal Messaging for channels:

Reliable Publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional Publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once the session has been established with the Universal Messaging realm server and the channel has been located, an event must be constructed prior to a publish call being made to the channel.

For reliable publish, there are a number of method prototypes on a channel that allow us to publish different types of events onto a channel. Here are examples of some of them. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
myChannel->publish(new nConsumeEvent("TAG", message->getBytes()));
```

Transactional Publish

Transactional publishing provides a means of verifying that the server received the events from the publisher, and therefore provides guaranteed delivery.

There are similar prototypes available to the developer for transactional publishing. Once the session is established and the channel located, we then need to construct the events for the transaction and publish these events to the transaction. Only when the transaction has been committed will the events become available to subscribers on the channel.

Below is a code snippet for transactional publishing:

```
std::list<nConsumeEvent*> messages;
messages->push_back(message1);
nTransactionAttributes *tattrib=new nTransactionAttributes(myChannel);
nTransaction *myTransaction=nTransactionFactory::create(tattrib);
myTransaction->publish(messages);
myTransaction->commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify that a transaction has been committed or aborted, a call can be made on the transaction that will determine if the events within the transaction were successfully received by the Universal Messaging Realm Server. This call can be made regardless of whether the connection was lost and a new connection was created.

The following code snippet demonstrates how to query the Universal Messaging Realm Server to see if the transaction was committed:

```
bool committed = myTransaction->isCommitted(true);
```

Asynchronous Subscriber

Asynchronous channel subscribers consume events from a callback on an interface that all asynchronous subscribers must implement. We call this interface an nEventListener.

The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

An example of such a simple listener is shown below:

```
class subscriber : public nEventListener{
  public:
   mySubscriber(){
    // construct your session
    // and channel objects here
    // begin consuming events from the channel at event id 0
    // i.e. the beginning of the channel
    myChannel->addSubscriber(this , 0);
   }
   void go(nConsumeEvent *pEvt) {
    printf("Consumed event %d",pEvt->getEventID());
   }
   int main(int argc, char** argv) {
    new mySubscriber();
    return 0;
   }
}
```

Asynchronous consumers can also be created using a selector, which defines a set of event properties and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties *props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'";
```

And pass this string into the addSubscriber method shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Channel Iterator

Events can be synchronously consumed from a channel using a channel iterator object. The iterator will sequentially move through the channel and return events as and when the iterator getNext() method is called.

If you are using iterators so that you know when all events have been consumed from a channel please note that this can also be achieved using an asynchronous subscriber by calling the nConsumeEvents isEndOfChannel() method.

An example of how to use a channel iterator is shown below:

```
class myIterator {
 private:
  nChannelIterator *iterator = null;
 public:
   myIterator(){
    // construct your  session and channel objects
    // start the iterator at the beginning of the channel (event id 0)
```

```
    iterator = myChannel->createIterator(0);
   }
   void start() {
    while (true) {
      nConsumeEvent *event = iterator->getNext();
      go(event);
    }
   }
   void go(nConsumeEvent *event) {
    printf("Consumed event %d",event->getEventID());
   }
   int main(int argc, char** argv) {
    myIterator *itr = new myIterator();
    itr->start();
    return 0;
   }
}
```

Synchronous consumers can also be created using a selector, which defines a set of event properties and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties *props = new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'"
```

And pass this string into the createIterator method shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Batched Subscribe

If a client application needs to subscribe to multiple channels it is more efficient to batch these subscriptions into a single server call. This is achieved using the subscribe method of nSession rather than first finding the nChannel object and then calling the subscribe method of nChannel.

The following code snippet demonstrates how to subscribe to two Universal Messaging channels in one server call:

```
public class myEventListener : public nEventListener {
  public void go(nConsumeEvent* evt) {
    cout<<"Received an event!";
  }
}
public void demo(){
  int numChans = 2;
  nSubscriptionAttributes **arr = new nSubscriptionAttributes*[numChans];
  arr[0] = new nSubscriptionAttributes("myChan1", "", 0, myLis1);
  arr[1] = new nSubscriptionAttributes("myChan2", "", 0, myLis2);
  mySession->subscribe(arr,numChans);
  for (int i = 0; i < arr.length; i++) {
    if (!arr[i]->wasSuccessful()) {
      handleSubscriptionFailure(arr[i]);
    }
    //subscription successful
```

```
  }
}
public void handleSubscriptionFailure(nSubscriptionAttributes* subAtts){
  cout<< subAtts.getException().StackTrace;
}
```

The nSubscriptionAttributes class is used to specify which channels to subscribe to. The second two parameters of the constructor represent the selector to use for the subscription and the event ID to subscribe from.

It is possible that the subscription may fail; for example, the channel may not exist or the user may not have the required privileges. In this situation, calling wasSuccessful() on the nSubscriptionAttributes will return false and getException() will return the exception that was thrown.

If the subscription is successful then the nChannel object can be obtained from the nSubscriptionAttributes as shown in the following code snippet:

```
  nChannel* chan = subAtts->getChannel();
```

Batched Find

In client applications, it is quite common to have multiple Channels or Queues that one is trying to find. In these scenarios, the batched find call built into nSession is extremely useful.

The following code snippet demonstrates how to find 2 Universal Messaging Channels in one server call:

```
void demo(){
  int numchans = 2;
  nChannelAttributes** arr = new nChannelAttributes*[numchans];
  nChannel** channels = new nChannels*[numchans];
  arr[0] = new nChannelAttributes("myChan1");
  arr[1] = new nChannelAttributes("myChan2");
  fSortedList<std::string, nFindResult*> *pArr = mySession->find(arr, numchans);
  int i =0;
  for (fSortedList<std::string, nFindResult*>::iterator iterator = pArr->begin();
    iterator != pArr->end(); iterator++)
  {
    if (!iterator->second->wasSuccessful())
    {
        handleSubscriptionFailure(iterator->second);
    }
    else if (iterator->second->isChannel())
    {
      channels[i] = iterator->second->getChannel();
    }
    i++;
  }
  public void handleSubscriptionFailure(nFindResult* result){
    // do something
  }
}
```

To perform the same operation for Queues, simply use the example above and exchange nChannel for nQueue, and check each result returned to see if the isQueue() flag is set.

The Merge Engine and Event Deltas

In order to streamline publish/subscribe applications it is possible to deliver only the portion of an event's data that has changed rather than the entire event. These event deltas minimise the amount of data sent from the publisher and ultimately delivered to the subscribers.

The publisher simply registers an event and can then publish changes to individual keys within the event. The subscriber will receive a full event on initial subscription, which contains the most up to date state of the event. After the initial message, only the key/value pairs which have changed since the last message will be sent to the client.

Publisher - Registered Events

In order to publish event deltas the publisher uses the Registered Event facility available on a Universal Messaging Channel. Please note that the channel must have been created with the Merge Engine and it must have a single Publish Key. The publish key represents the primary key for the channel and the registered events. So for example if you are publishing currency rates you would setup a channel as such

```
   nChannelAttributes* cattr
            = new nChannelAttributes("RatesChannel", 0, 0,
nChannelAttributes.SIMPLE_TYPE);
//
// This next line tells the server to Merge incoming events based on the publish
//  key name and the name of the registered event
//
        cattr->useMergeEngine(true);
//
// Now create the Publish Key (See publish Keys for a full description
//
        nChannelPublishKeys** pks = new nChannelPublishKeys[1];
        pks[0] = new nChannelPublishKeys("ccy", 1);
        cattr->setPublishKeys(pks);
//
// Now create the channel
//
        myChannel = mySession->createChannel(cattr);
```

At this point the server will have a channel created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent* audEvent = myChannel->createRegisteredEvent("AUD");
        nEventProperties* props = audEvent->getProperties();
        props->put("bid", 0.8999);
        props->put("offer", 0.9999);
        props->put("close", "0.8990");
        audEvent->commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to a ccy value of "AUD". We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props->put("bid", 0.9999);
        audEvent->commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

Subscriber - nEventListener v nRegisteredEventListener

The subscriber doesn't need to do anything different to receive these events. The standard nEventListener will appear to receive full events with all keys and data even though only the changed keys were transmitted. The events are reassembled on the client and are updated locally such that the subscriber receives the usual callback from the server.

If the client only wants to process the changes then they can choose to implement the nRegisteredEventListener interface rather than the nEventListener interface. The nRegisteredEventListener, has an update() method in addition to the usual go() method. The update method will be called whenever an update has been published.

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same channel or queue. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages. The priority is a numeric value in the range 0 (lowest priority) to 9 (highest priority).

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent* evt;
    ...
evt->getAttributes()->setPriority(9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically, events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis, and the effects become more noticeable as load increases.

Note:
If events are stored for replay at a later stage, for example for a durable subscriber who is currently not consuming events, higher priority events will be delivered earlier than lower

priority events when the durable subscriber starts consuming the events, even if the lower priority events were created much earlier .

Publish / Subscribe using Data Streams and Data Groups

Publish / Subscribe Using Data Streams and Data Groups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging Data Groups are lightweight structures designed to facilitate Publish/Subscribe . When using DataGroups, user subscriptions are managed remotely in a way that is transparent to subscribers. Universal Messaging Channels provide an alternative style of Publish/Subscribe where the subscribers manage their subscriptions directly.

This section demonstrates Universal Messaging pub / sub using Data Groups in C++, and provides example code snippets for all relevant concepts.

DataStreamListener

If a nSession is created with a nDataStreamListener then it will receive asynchronous callbacks via the onMessage implementation of the nDataStreamListener interface. The nDataStreamListener will receive events when:

- An event is published directly to this particular nDataStream
- An event is published to any nDataGroup which contains this nDataStream
- An event is published to an nDataGroup which contains a nested nDataGroup containing this nDataStream
- An example of how to create a session with an nDataStreamListener interface is shown below:

```
public class DataGroupClient : public nDataStreamListener{

    nSession* mySession;
    public DataGroupClient( std::string& realmURLs){
        nSessionAttributes* nsa = new nSessionAttributes(realmURLs);
        mySession = nSessionFactory::create(nsa, this);
        mySession->init(this);
    }

    ////
    // nDataStreamListener Implementation
    ////

    //Callback received when event is available
    public void onMessage(nConsumeEvent* event){

            //some code to process the message

    }
}
```

Creating and Deleting DataGroups

Creating Universal Messaging DataGroups

nDataGroups can be created programmatically as detailed below, or they can be created using the Universal Messaging enterprise manager.

In order to create a nDataGroup, first of all you must create an nSession object, which is effectively your the logical and physical connection to a Universal Messaging Realm. This is achieved by using an RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
std::string* RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes* nsa=new nSessionAttributes(RNAME);
nSession* mySession=nSessionFactory::create(nsa);
mySession->init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession object instance 'mySession', you can then create DataGroups. The create DataGroup methods will return the nDataGroup if it already exists.

The code snippets below demonstrate the creation of nDataGroups:

Create a Single nDataGroup

```
nDataGroup* myGroup = mySession->createDataGroups("myGroup");
```

Create Multiple nDataGroups

```
std::string* groups = {"myFirstGroup", "mySecondGroup"};
nDataGroup* myGroups = mySession->createDataGroups(groups);
```

Creating DataGroups with DataGroupListeners and ConflationAttributes

It is also possible to specify additional properties when creating DataGroups:

- nDataGroupListener - To specify a listener for DataGroup membership changes
- nConflationAttributes - To specify attributes which control event merging and delivery throttling for the DataGroup

Now we have a reference to a Universal Messaging DataGroup it is possible to publish events

Deleting Universal Messaging DataGroups

There are various deleteDataGroup methods available on nSession which will delete DataGroups. It is possible to specify single nDataGroups or arrays of nDataGroups.

Managing DataGroup Membership

DataGroups are extremely lightweight from both client and server perspectives; a back-end process, such as a Complex Event Processing engine, can simply create DataGroups and then add or remove users (or even entire nested DataGroups) based on bespoke business logic. A user who is removed from one DataGroup and added to another will continue to receive events without any interruption to service, or indeed explicit awareness that any DataGroup change has occurred.

This page details some of the typical operations that DataGroup management process would carry out.

Please see our C++ sample apps for more detailed examples of DataGroup management.

Tracking Changes to DataGroup Membership (DataGroupListener)

The nDataGroupListener interface is used to provide asynchronous notifications when nDataGroup membership changes occur. Each time a user (nDataStream) or nDataGroup is added or removed from a nDataGroup a callback will be received.

```
public class datagroupListener : public nDataGroupListener {
    nSession* mySession;
    public datagroupListener(nSession session){
        mySession = session;
        //add this class as a listener for all nDataGroups on this Universal
        //  Messaging realm
        mySession->getDataGroups(this);
    }

    ////
    //DataGroupListener Implementation
    ///
    public void addedGroup (nDataGroup* to, nDataGroup* group, int count){
        //Called when a group has been added to the 'to' data group.
        //count is the number of nDataStreams that will receive any events
        //published to this nDataGroup
    }
    public void addedStream (nDataGroup* group, nDataStream* stream, int count){
        //Called when a new stream has been added to the data group.
    }
    public void createdGroup (nDataGroup* group){
        //Called when a group has been created.
    }
    public void deletedGroup (nDataGroup* group){
        //Called when a group has been deleted.
    }
    public void deletedStream (nDataGroup* group, nDataStream* stream, int count,
           bool serverRemoved){
        //Called when a stream has been deleted from the data group.
       //serverRemoved is true if the nDataStream was removed because of flow control
    }
    public void removedGroup (nDataGroup* from, nDataGroup* group, int count){
        //Called when a group has been removed from the 'from' data group.
    }

}
```

There are three ways in which the nDataGroupListener can be used:

Listening to an individual DataGroup

Listeners can be added to individual DataGroups when they are created or at any time after creation. The code snippets illustrate both approaches:

```
mySession->createDataGroup(dataGroupName, datagroupListener);
```

```
myDataGroup->addListener(datagroupListener);
```

Listening to the Default DataGroup

The Default nDataGroup is a DataGroup to which all nDataStreams are added by default. If you add a DataGroupListener to the default DataGroup then callbacks will be received when:

- a nDataStream is connected/disconnected
- a nDataGroup is created or deleted

Listening to all DataGroups on a Universal Messaging Realm

The code snippet below will listen on all nDataGroups (including the default DataGroup).

```
mySession->getDataGroups(datagroupListener);
```

Adding and Removing DataGroup Members

The nDataGroup class provides various methods for adding and removing nDataStreams and nDataGroups. Please see the nDataGroup API documentation for a full list of methods. Examples of some of these are provided below:

```
    //Add a nDataStream (user) to a nDataGroup
    public void addStreamToDataGroup(nDataGroup* group, nDataStream* user){
        group->add(user);
    }
    //Remove a nDataStream (user) from a nDataGroup
    public void removeStreamFromDataGroup(nDataGroup* group, nDataStream* user){
        group->remove(user);
    }
    //Add a nDataGroup to a nDataGroup
    public void addNestedDataGroup(nDataGroup* parent, nDataGroup* child){
        parent->add(child);
    }
    //Remove a nDataGroup from a nDataGroup
    public void removeNestedDataGroup(nDataGroup* parent, nDataGroup* child){
        parent->remove(child);
    }
```

DataGroup Conflation Attributes

Enabling Conflation on DataGroups

Universal Messaging DataGroups can be configured so that conflation (merging and throttling of events) occurs when messages are published. Conflation can be carried out in several ways and these are specified using a nConflationAttributes object. The ConflationAttributes object is passed in to the DataGroup when it is created initially.

The nConflationAttributes object has two properties action and interval. Both of these are passed into the constructor.

The action property specifies whether published events should replace previous events in the DataGroup or be merged with them. These properties are defined by static fields:

```
nConflationAttributes::sMergeEvents
nConflationAttributes::sDropEvents
```

The interval property specifies the interval in milliseconds between event fanout to subscribers. An interval of zero implies events will be fanned out immediately.

Creating a Conflation Attributes Object

```
//ConflationAttributes specifying merge events  and no throttled delivery
nConflationAttributes* confattribs =
   new nConflationAttributes(nConflationAttributes::sMergeEvent, 0);
//ConflationAttributes specifying merge events and throttled delivery at
// 1 second intervals
nConflationAttributes* confattribs =
   new nConflationAttributes(nConflationAttributes::sMergeEvent, 1000);
//ConflationAttributes specifying drop events and throttled delivery at
// 1 second intervals
nConflationAttributes* confattribs =
   new nConflationAttributes(nConflationAttributes::sDropEvent, 1000);
```

Create a Single nDataGroup with Conflation Attributes

```
//create a DataGroup passing in this class as a nDataGroupListener and
// a ConflationAttributes
myDataGroup = mySession->createDataGroup(dataGroupName, this, confattribs);
```

Create Multiple nDataGroups with Conflation Attributes

```
nConflationAttributes* confattribs =
   new nConflationAttributes(nConflationAttributes::sMergeEvent, 1000);
std::string[] groups = {"myFirstGroup", "mySecondGroup"};
nDataGroup[] myGroups = mySession->createDataGroups(groups, confattribs);
```

Publishing Events to Conflated DataGroups With A Merge Policy

At this point the server will have a nDataGroup created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent* audEvent = myDataGroup->createRegisteredEvent();
        nEventProperties* props = audEvent->getProperties();
```

```
        props->put("bid", 0.8999);
        props->put("offer", 0.9999);
        props->put("close", "0.8990");
        audEvent->commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to a ccy value of "AUD". We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props->put("bid", 0.9999);
        audEvent->commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

Publishing Events to Conflated DataGroups With A Drop Policy

If you have specified a "Drop" policy in your ConflationAttributes then events are published in the normal way rather than using nRegisteredEvent.

Consuming Conflated Events from a DataGroup

The subscriber doesn't need to do anything different to receive events from a DataGroup with conflation enabled. If nRegisteredEvents are being delivered then the events will contain only the fields that have changed will be delivered. In all other circumstances an entire event is delivered to all consumers.

DataGroups Event Publishing

You can get references to any DataGroup from the nSession object. There are various writeDataGroup methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataGroups.

```
myDataGroup* = mySession->getDataGroup("myGroup");
nEventProperties* props = new nEventProperties();
//You can add other types in a dictionary object
props->put("key0string"+x, "1"+x);
props->put("key1int", (int) 1);
props->put("key2long", (long) -11);
nConsumeEvent* evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession->writeDataGroup(evt1, myDataGroup);
```

DataStream Event Publishing

You can get references to any nDataStream (user) from the nSession object if you call getDefaultDataGroup(). You can also access nDataStreams by implementing the nDataGroupListener interface. Please refer to DataGroup management (see "Managing DataGroup Membership" on page 115) for more information. This will deliver callbacks as users are connected/disconnected. There are various writeDataStream methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataStreams.

```
nEventProperties* props = new nEventProperties();
//You can add other types in a dictionary object
props->put("key0string"+x, "1"+x);
props->put("key1int", (int) 1);
props->put("key2long", (long) -11);
nConsumeEvent* evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession->writeDataStream(evt1, myDataStream)
```

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same datagroup. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages.

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent evt;
    ...
evt->getAttributes()->setPriority(9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis and the effects become more noticeable as load increases.

It is possible to specify multiple levels of priority for events on the same datagroup. This behaviour will cause the events to be delivered highest priority first. When doing this it is important to realise that events on a datagroup will no longer be delivered on a first in first out basis.

Message Queues

Message Queues

Universal Messaging provides message queue functionality through the use of queue objects. Queues are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Message queues differ from publish / subscribe channels in the way that events are delivered to consumers. Whilst queues may have multiple consumers, each event is typically only delivered to one consumer, and once consumed (popped) it is removed from the queue.

Universal Messaging also supports non destructive reads (peeks) from queues which enable consumers to see what events are on a queue without removing it from the queue. Any event which has been peeked will still be queued for popping in the normal way. The Universal Messaging enterprise manager also supports the ability to visually peek a queue using its snoop capability.

This section demonstrates how Universal Messaging message queues work in C++, and provide examples code snippets for all relevant concepts.

Creating a Queue

In order to create a queue, first of all you must create your nSession object, which is effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
std::string[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nSession *mySession = nSessionFactory::create(nsa);
mySession->init();
```

Once the mySession->init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then begin creating the queue object. Queues have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them,

To create a queue, we do the following:

```
nChannelAttributes *cattrib = new nChannelAttributes();
cattrib->setChannelMode(nChannelAttributes::QUEUE_MODE);
cattrib->setMaxEvents(0);
cattrib->setTTL(0);
cattrib->setType(nChannelAttributes::PERSISTENT_TYPE);
cattrib->setName("myqueue");
nQueue *myQueue=mySession->createQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Note:
For information about the set of permissible characters you can use to name a queue, see the section *Creating Queues* in the Enterprise Manager section of the *Administration Guide*.

Finding a Queue

In order to find a queue, first of all the queue must be created. This can be achieved through the Universal Messaging Enterprise Manager, or programmatically (see "Creating a Queue" on page 120). First of all you must create your nSession object, which is your effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
std::string[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes *nsa=new nSessionAttributes(RNAME);
nSession *mySession = nSessionFactory->create(nsa);
mySession->init();
```

Once the nSession->init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the queue object. Queues have an associated set of attributes, that define their behavior within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them.

To find a queue previously created, we do the following:

```
nChannelAttributes *cattrib = new nChannelAttributes();
cattrib->setName("myqueue");
nQueue *myQueue = mySession->findQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Behavior if a queue has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "QUEUE DELETED" and an event data component equal to "The queue has been deleted".

Queue Publish

There are 2 types of publish available in Universal Messaging for queues:

- "Reliable Publish" on page 122
- "Transactional Publish" on page 122

Reliable publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once you have established a session and found a queue, you then need to construct an event (see "Universal Messaging Events" on page 102) and publish the event onto the queue.

For reliable publish, here is the example code for how to publish events to a queue. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
 myQueue->push(new nConsumeEvent("TAG", message->getBytes(), size);
```

Transactional Publish

Transactional publishing provides us with a method of verifying that the server receives the events from the publisher, and provides guaranteed delivery.

There are similar prototypes available to the developer for transaction publishing. Once we have established our session (see "Creating a Session" on page 101) and our queue (see "Finding a Queue" on page 120), we then need to construct our events (see "Universal Messaging Events" on page 102) and our transaction and publish these events to the transaction. Then the transaction will be committed and the events available to consumers to the queue.

Below is a code snippet of how transactional publishing is achieved:

```
std::list<nConsumeEvent*> messages;
messages->push_back(message1);
nTransactionAttributes *tattrib=new nTransactionAttributes(myQueue);
nTransaction *myTransaction=nTransactionFactory::create(tattrib);
myTransaction->publish(messages);
myTransaction->commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify that a transaction has been committed or aborted, a call can be made on the transaction that will determine if the events within the transactional were successfully received by the Universal Messaging Realm Server.

```
bool committed = myTransaction->isCommitted(true);
```

Which will query the Universal Messaging Realm Server to see if the transaction was committed.

An example of publishing events onto a queue can be found on the examples page under "Push Queue". An example of how to transactionally publish events to a queue can be found on the examples page under "TX Push Queue".

Asynchronous Queue Consuming

Asynchronous queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

An example of an asynchronous queue reader is shown below:

```
class myAsyncQueueReader : public nEventListener {

   private:
    nQueue *myQueue = null;
    myAsyncQueueReader(){
        // construct your session and queue objects here
        // begin consuming events from the queue
        nQueueReaderContext *ctx = new
        nQueueReaderContext(this, 10);
        nQueueAsyncReader *reader = myQueue->createAsyncReader(ctx);
    }
    void go(nConsumeEvent event) {
        printf("Consumed event %d",event.getEventID());
    }
    int main(int argc, char** argv) {
        new myAsyncQueueReader();
        return 0;
    }
 }
```

Asynchronous queue consumers can also be created using a selector, which defines a set of event properties (see “Event Dictionaries” on page 102) and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties *props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

An example of an asynchronous queue reader can be found on the examples page under "Queue Subscriber".

Synchronous Queue Consuming

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

An example of a synchronous queue reader is shown below:

```
class mySyncQueueReader {
   private:
        nQueueSyncReader *reader = null;
        nQueue *myQueue = null;
   public:
    mySyncQueueReader(){
        // construct your session and queue objects  here
        // construct the queue reader
        nQueueReaderContext *ctx = new
        nQueueReaderContext(this, 10);
        reader = myQueue->createReader(ctx);
    }
    void start(){
        while (true) {
            // pop events from the queue
            nConsumeEvent *event = reader->pop();
            go(event);
        }
    }
    void go(nConsumeEvent *event) {
        printf("Consumed event %d",event->getEventID());
    }
    int main(int argc, char** argv) {
        mySyncQueueReader *sqr = new mySyncQueueReader();
        sqr->start();
        return 0;
    }
 }
```

Synchronous queue consumers can also be created using a selector, which defines a set of event properties (see "Event Dictionaries" on page 102) and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std:string selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

An example of a synchronous queue consumer can be found on the examples page under "Queue Reader".

Asynchronous Transactional Queue Consuming

Asynchronous transactional queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional asynchronous queue reader is shown below:

```
class myAsyncTxQueueReader : public nEventListener {
   private:
    nQueueAsyncTransactionalReader *reader = null;
    nQueue *myQueue = null;
   public:
    myAsyncTxQueueReader(){
        // construct your session and queue objects here
        // begin consuming events from the queue
        nQueueReaderContext *ctx = new
        nQueueReaderContext(this, 10);
        reader = myQueue->createAsyncTransactionalReader(ctx);
    }
    void go(nConsumeEvent *event) {
        printf("Consumed event %d",event->getEventID());
        reader->commit();
    }
    int main(int argc, char** argv) {
        new myAsyncTxQueueReader();
        return 0;
    }
 }
```

As previously mentioned, the big difference between a transactional asynchronous reader and a standard asynchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the .rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with event id's 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Asynchronous queue consumers can also be created using a selector, which defines a set of event properties (see "Event Dictionaries" on page 102) and their values that a subscriber is interested in. For example if events are being published with the following event properties:

```
nEventProperties *props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Synchronous Transactional Queue Consuming

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional synchronous queue reader is shown below:

```
class mySyncTxQueueReader{
    nQueueSyncTransactionReader *reader = null;
    nQueue *myQueue = null;
   public:
    mySyncTxQueueReader(){
        // construct your session and queue objects here
        // construct the transactional queue reader
        nQueueReaderContext *ctx = new
        nQueueReaderContext(this, 10);
        reader = myQueue->createTransactionalReader(ctx);
    }
    void start(){
        while (true) {
            // pop events from the queue
            nConsumeEvent *event = reader->pop();
            go(event);
            // commit each event consumed
            reader->commit(event->getEventID());
        }
    }
    void go(nConsumeEvent *event) {
        printf("Consumed event %d",event->getEventID());
    }
    int main(int argc, char** argv) {
        new mySyncTxQueueReader();
        sqr->start();
        return 0;
    }
 }
```

As previously mentioned, the big difference between a transactional synchronous reader and a standard synchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the .rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with event id's 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Synchronous queue consumers can also be created using a selector, which defines a set of event properties (see "Event Dictionaries" on page 102) and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueueReaderContext object shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

An example of a synchronous queue consumer can be found on the examples page under "Queue Reader".

Queue Browsing / Peeking

Universal Messaging provides a mechanism for browsing (peeking) queues. Queue browsing is a non-destructive read of events from a queue. The queue reader used by the peek will return an array of events, the size of the array being dependent on how many events are in the queue, and the window size defined when your reader context is created. For more information, please see the Universal Messaging Client API documentation.

An example of a queue browser is shown below:

```
public class myQueueBrowser {
    private:
     nQueueSyncReader *reader;
     nQueuePeekContext *ctx;
     nQueue *myQueue;
    public:
     myQueueBrowser(){
        // construct your session and queue objects here
        // create the queue reader
        reader = myQueue->createReader(new
        nQueueReaderContext());
        ctx = nQueueReader::createContext(10);
     }
     void start(){
        bool more = true;
        long eid =0;
        while (more) {
            // browse (peek) the queue
            int size;
            nConsumeEvent **evts = reader->peek(ctx,size);
```

```
            for (int x=0; x < size; x++) {
                go(evts[x]);
            }
            more = ctx->hasMore();
        }
     }
     void go(nConsumeEvent *event) {
        printf("Consumed event %d",event->getEventID());
     }
    int main(int argc, char** argv) {
        myQueueBrowser *qbrowse = new myQueueBrowser();
        qbrowse->start();
        return 0;
     }
}
```

Queue browsers can also be created using a selector, which defines a set of event properties (see "Event Dictionaries" on page 102) and their values that a browser is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props->put("BONDNAME","bond1");
```

If you then provide a message selector string in the form of:

```
std::string selector = "BONDNAME='bond1'";
```

And pass this string into the constructor for the nQueuePeekContext object shown in the example code, then your browser will only receive messages that contain the correct value for the event property BONDNAME.

An example of an queue browser can be found on the examples page under "Queue Peek".

Code Examples for C++

Publish / Subscribe using Channel Topics

C++ Client: Channel Publisher

This example publishes events onto a Universal Messaging Channel.

Usage

```
publish <rname> <channel name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Transactional Channel Publisher

This example publishes events transactionally to a Universal Messaging Channel. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been commited succesfully.

Usage

```
txpublish <rname>  <channel name> [count] [size] [tx size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[tx size] - The number of events per transaction (default: 1)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Asynchronous Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel. See also: “ Synchronous Subscription” on page 130

Usage

```
subscriber <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - URL of realm to connect to
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

C++ Client: Synchronous Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel. See also: " Asynchronous Subscription" on page 129

Usage

```
channeliterator <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

C++ Client: Event Delta Delivery

This example shows how to publish and receive registered events.

Usage

```
RegisteredEvent <rname> <channel name> [count]
<Required Arguments>
<rname> - Rname of the server to connect to
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Batching Server Calls

This example shows how to find multiple channels and queues in one call to the server.

Usage

```
findChannelsAndQueues <RNAME> <name> <name> <name>.....
<Required Arguments>
<RNAME> - The RNAME of the realm you wish to connect to
<name> - The name(s) of the channels to find
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

C++ Client: Batching Subscribe Calls

This example of batching shows how to subscribe to multiple Universal Messaging Channels in one server call.

Usage

```
sessionSubscriber <RNAME> <channelnames>
<Required Arguments>
<RNAME> - The RNAME of the realm you wish to connect to
<channelnames> - Comma separated list of channels to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

Publish / Subscribe using Data Streams and Data Groups

C++ Client: DataStream Listener

This example shows how to initialise a session with a DataStream listener and start receiving data.

Usage

```
DataStreamListener <rname> [debug] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

C++ Client: DataGroup Publishing with Conflation

This example shows how to publish to DataGroups, with optional conflation.

Usage

```
DataGroupPublish <rname> <group name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<group name> - Data group name parameter to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
```

Application Source Code

See the online documentation for a code example.

C++ Client: DataGroup Manager

This is an example of how to run a DataGroup manager application

Usage

```
dataGroupsManager <rname> <Properties File Location>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<Properties File Location Data Groups> - The location of the property file to use
    for mapping data groups to data groups
<Properties File Location Data Streams> - The location of the property file to use
    for mapping data streams to data groups
<Auto Recreate Data Groups> - True or False to auto recreate data groups takes the
    data group property file and creates channels
    a group for every name mentioned on the left of equals sign
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

C++ Client: Delete DataGroup

This is a simple example of how to delete a DataGroup

Usage

```
deleteDataGroup <rname> <group name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<group name> - Data group name parameter to delete
```

Application Source Code

See the online documentation for a code example.

C++ Client: DataGroup Delta Delivery

This example shows how to use delta delivery with DataGroups.

Usage

```
DataGroupDeltaDelivery <rname>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
[Optional Arguments]
[count] - the number of times to commit the registered events (default : 10)
```

Application Source Code

See the online documentation for a code example.

Message Queues

C++ Client: Queue Publisher

This example publishes events onto a Universal Messaging Queue.

Usage

```
pushq <rname> <queue name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Transactional Queue Publisher

This example publishes events transactionally to a Universal Messaging Queue. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been commited succesfully.

Usage

```
txpushq <rname> <queue name> [count] [size] [tx size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
```

```
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[tx size] - The number of events per transaction (default: 1)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Asynchronous Queue Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Queue. See also: “ Synchronous Queue Subscription” on page 134

Usage

```
qsubscriber <rname> <queue name> [debug] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name to pop from
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[transactional] - true / false whether the subscriber is transactional,
                  if true, each event consumed will be ack'd to confirm receipt
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

C++ Client: Synchronous Queue Consumer

This example shows how to synchronously consume events from a Universal Messaging Queue. See also: “ Asynchronous Queue Subscription” on page 134

Usage

```
qreader <rname> <queue name> [debug] [timeout] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name to pop from
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[timeout] - The timeout for the synchronous pop call
[transactional] - true / false whether the subscriber is transactional,
                  if true, each event consumed will be ack'd to confirm receipt
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

C++ Client: Peek Events on a Queue

Consume events from a Universal Messaging Queue in a non-destructive manner

Usage

```
qpeek <rname> <queue name> [debug] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name to pop from
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

Administration API

C++ Client: Add a Queue ACL Entry

This example demonstrates how to add an ACL entry to a Universal Messaging Queue.

Usage

```
naddqueueacl <rname> <user> <host> <queue name> [list_acl] [modify_acl] [full]
                                                [peek] [write] [purge] [pop]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter for the new ACL entry
<host> - Host name parameter for the new ACL entry
<queue name> - Queue name parameter for the new ACL entry
[Optional Arguments]
[list_acl] - Specifies that the list acl permission should be added
[modify_acl] - Specifies that the modify acl permission should be added
[full] - Specifies that the full permission should be added
[peek] - Specifies that the read permission should be added
[write] - Specifies that the write permission should be added
[purge] - Specifies that the purge permission should be added
[pop] - Specifies that the pop permission should be added
```

Application Source Code

See the online documentation for a code example.

C++ Client: Modify a Channel ACL Entry

This example demonstrates how to modify the permissions of an ACL entry on a Universal Messaging Channel..

Usage

```
nmodchanacl <rname> <user> <host> <channel name>
        [list_acl] [modify_acl] [full] [last_eid] [read] [write] [purge] [named]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter for the new ACL entry
<host> - Host name parameter for the new ACL entry
<channel name> - Channel name parameter for the new ACL entry
[Optional Arguments]
[+/-] - Prepending + or - specifies whether to add or remove a permission
[list_acl] - Specifies that the list acl permission should be added
[modify_acl] - Specifies that the modify acl permission should be added
[full] - Specifies that the full permission should be added
[last_eid] - Specifies that the get last EID permission should be added
[read] - Specifies that the read permission should be added
[write] - Specifies that the write permission should be added
[purge] - Specifies that the purge permission should be added
[named] - Specifies that the used named subscriber permission should be added
[all_perms] - Specifies that the pop permission should be added/removed
```

Application Source Code

See the online documentation for a code example.

C++ Client: Delete a Realm ACL Entry

This example demonstrates how to delete an ACL entry from a realm on a Universal Messaging Channel.

Usage

```
delrealmacl <rname> <user> <host>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter for the ACL entry to delete
<host> - Host name parameter for the ACL entry to delete
```

Application Source Code

See the online documentation for a code example.

C++ Client: Monitor realms for client connections coming and going

This example demonstrates how to monitor for connections to the realm and its channels.

Application Source Code

See the online documentation for a code example.

C++ Client: Console-based Realm Monitor

This example demonstrates how to monitor live realm status.

Application Source Code

See the online documentation for a code example.

C++ Client: Remove Node ACL

This shows how the ACL for an nNode can be removed.

Usage

```
ndelnodeacl <rname> <user> <host> <channel name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name
<host> - Host name
<node> - Channel / Queue name to remove the entry from
```

Application Source Code

See the online documentation for a code example.

C++ Client: Authserver

This demonstrates how to set security permissions when connection attempts are made on the realm.

Application Source Code

See the online documentation for a code example.

Channel / Queue / Realm Management

C++ Client: Creating a Channel

This example demonstrates how to create a Universal Messaging channel programmatically

Usage

```
makechan <rname> <channel name> [time to live] [capacity] [type] [cluster wide]
                                [start eid]
```

```
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to be created
[Optional Arguments]
[time to live] - The Time To Live parameter for the new channel (default: 0)
[capacity] - The Capacity parameter for the new channel (default: 0)
[type] - The type parameter for the new channel (default: S)
R - For a reliable (stored in memory) channel with persistent eids
P - For a persistent (stored on disk) channel
M - For a Mixed (allows both memory and persistent events) channel
[cluster wide] - Whether the channel is cluster wide. Will only work if the realm
                 is part of a cluster
[start eid] - The initial start event id for the new channel (default: 0)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Deleting a Channel

This example demonstrates how to delete a Universal Messaging channel programmatically.

Usage

```
deletechan <rname> <channel name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to be deleted
```

Application Source Code

See the online documentation for a code example.

C++ Client: Creating a Queue

This example demonstrates how to create a Universal Messaging queue programmatically.

Usage

```
makequeue <rname> <queue name> [time to live] [capacity] [type] [cluster wide]
                               [start eid]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - queue name parameter for the queue to be created
[Optional Arguments]
[time to live] - The Time To Live parameter for the new queue (default: 0)
[capacity] - The Capacity parameter for the new queue (default: 0)
[type] - The type parameter for the new queue (default: S)
R - For a reliable (stored in memory) queue with persistent eids
P - For a persistent (stored on disk) queue
M - For a Mixed (allows both memory and persistent events) queue
[cluster wide] - Whether the queue is cluster wide. Will only work if the realm
                 is part of a cluster
[start eid] - The initial start event id for the new queue (default: 0)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Deleting a Queue

This example demonstrates how to delete a Universal Messaging queue programmatically.

Usage

```
deletequeue <rname> <queue name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name parameter for the queue to be deleted
```

Application Source Code

See the online documentation for a code example.

C++ Client: Create Channel Join

Create a join between two Universal Messaging Channels

Usage

```
makechanneljoin <rname> <source channel name> <destination channel name>
    [max hops] [selector] [allow purge]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination channel name> - Channel name parameter of the remote channel name to join
[Optional Arguments]
[max hops] - The maximum number of join hops a message can travel through
[selector] - The event filter std::string to use on messages travelling through
             this join
[allow purge] - boolean to specify whether purging is allowed (default : true)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Delete a Channel Join

Delete a join between two Universal Messaging Channels

Usage

```
deletechanneljoin <rname> <source channel name> <destination channel name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
```

```
<source channel name> - Channel name parameter of the local channel name to join
<destination channel name> - Channel name parameter of the remote channel name to join
```

Application Source Code

See the online documentation for a code example.

C++ Client: Purge Events From a Channel

Purge events from a Universal Messaging channel

Usage

```
purgechan <rname> <channel name> [start eid] [end eid] [selector] [wait]
<Required Arguments>
<rname> - URL of realm to connect to
<channel name> - Channel name parameter for the channel to purge to
[Optional Arguments]
[start eid] - The Event ID to start purging from
[end eid] - The Event ID to purge to
[selector] - The purge event filter string to use
[wait] - whether to wait for a response (synchronous) or not (asynchronous)
```

Application Source Code

See the online documentation for a code example.

C++ Client: Create Queue Join

Create a join between a Universal Messaging Queue and a Universal Messaging Channel

Usage

```
makequeuejoin <rname> <source channel name> <destination queue name>
              [max hops] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination queue name> - Queue name parameter of the remote queue name to join
[Optional Arguments]
[max hops] - The maximum number of join hops a message can travel through
[selector] - The event filter std::string to use on messages travelling through
             this join
```

Application Source Code

See the online documentation for a code example.

C++ Client: Delete Queue Join

Delete a join between a Universal Messaging Queue and a Universal Messaging Channel

Usage

```
deletequeuejoin <rname> <source channel name> <destination queue name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination queue name> - Queue name parameter of the remote queue name to join
```

Application Source Code

See the online documentation for a code example.

C++ Prerequisites

Prerequisites

Universal Messaging C++ makes use of certain non-standard C++ libraries.

POCO

Universal Messaging C++ uses the POCO C++ class libraries. The required POCO libraries are distributed with Universal Messaging so no installation is required, however please see the Environment Setup section below for further details on how to compile and run Universal Messaging C++ applications using these libraries.

For more information, please visit the POCO website at http://pocoproject.org/.

OpenSSL

The OpenSSL libraries required by the C++ libraries are delivered with the Universal Messaging installation.

To subscribe to a channel using an SSL interface, extra requirements must be met. SSL requires certificates to be set up on the client and server. The location of these certificates must be known to the applications. For instructions on how to run Universal Messaging C++ applications using an SSL enabled interface, please see "Client SSL Configuration" on page 142.

To learn more about SSL please see the SSL Concepts section.

Environment Setup

In order to compile and run applications using Universal Messaging C++, the environment must be set up correctly.

Environment setup is different for different operating systems:

- "Environment Setup: Windows" on page 143
- "Environment Setup: Linux and OS X" on page 144

Client SSL Configuration

Universal Messaging fully supports *SSL Encryption*. This section describes how to use SSL in your Universal Messaging C++ client applications.

Once you have created an SSL enabled interface you will need to create certificates for the server and the client. The Universal Messaging download contains a generator to create some example Java key store files to be used by the Universal Messaging server but may also be converted to Privacy Enhanced Mail Certificates (.pem) for use with a Universal Messaging C++ client.

Please refer to the Enterprise Manager guide to create your own client certificates. However please remember that in order to run a Universal Messaging C++ client, the certificate provided must be in .pem format.

Running a Universal Messaging C++ Client

A client can be run anonymously which means that any client can subscribe to a channel securely. The server can also be run with client validation such that only trusted clients can connect. To enable or disable client certificate validation you can use the Universal Messaging Enterprise Manager. Highlight the SSL enabled interface in the "Interface" tab for your realm then open the "Certificates" tab and check or uncheck the box labelled "Enable Client Cert Validation".

In order to run a client using SSL, the location of the key stores and the relevant passwords need to be specified in nConstants. This can be done by setting up the relevant environment variables (as necessary to run the sample applications), or by calling the relevant set methods (defined in nConstants) from the application code.

Different environment variables need to be set depending on whether client certificate validation is enabled:

With Client Certificate Validation

In this case, the client must hold a certificate to validate that it can be trusted. It must also have a trust store such that it can validate that the server is trusted. The key store located at CERTPATH also contains the client's private key and therefore must have a password associated with it. Therefore the following environment variables must be set:

- CERTPATH - The path where the client certificate is located
- CERTPASS - The password for the client certificate
- CAPATH - The path where the trust store is located

Without Client Certificate Validation

If client certificate validation has been disabled on the server then clients connect to the server anonymously. This means that clients do not need to have a certificate and therefore CERTPATH and CERTPASS do not need to be set. With Universal Messaging C++ server-side validation is also set to be non-strict. This means that the client does not need to have a trust store because it will not try to validate the server certificate, therefore it is not necessary to set the CAPATH.

See the SSL Concepts section for more detailed information.

Environment Setup: Windows

The guide below explains how the Universal Messaging C++ environment can be set up for compiling and running the applications on a Windows operating system.

Note:
Setting up the environment involves the use of the command prompt at the operating system level. General information about using the command prompt for this purpose is contained in the section "Running the Sample Applications" on page 11.

The Universal Messaging C++ and POCO libraries can be found in a platform-specific directory under the `<InstallDir>\UniversalMessaging\cplus\lib` directory. In order to run Universal Messaging C++ applications, the location of these libraries must be known to the system. There are several methods which can be used to achieve this:

1. By updating the PATH environment variable in the command prompt used to compile or run code:

   ```
   set PATH=<InstallDir>\UniversalMessaging\cplus\lib\x86_64;%PATH%
   ```

   where *<InstallDir>* is the root folder of the Universal Messaging installation.

   This will allow you to run applications in the current command prompt.

2. In order to update the PATH globally, you need to:

   - Open System in the Control Panel.
   - Expand the "Advanced" tab and click the button labelled "Environment Variables"
   - In the new window, the Path variable is found in the "System Variables" section. Highlighting the variable and clicking "edit" will open another window.
   - In this new window you should append the location of the libraries to the beginning of the "variable value" section.

3. Another way to make the libraries globally available is to copy them into the Windows System32 folder located at:

   C:\WINDOWS\System32

   This directory is looked in by default for Runtime libraries.

To compile applications, the compiler will need to know the location of the POCO lib files, `Nirvana.lib` and certain C++ header files. The location of the libs is described above, and the headers are located in `cplus\include`. The `cplus\examples` directory contains the source code for several sample applications as well as project files (.vcproj) which can be opened with Microsoft Visual Studio. Each application comes pre-compiled, the executable (.exe) can be found in the application's directory (`cplus\examples\applicationName`).

Compiling the Sample Applications

In order to make compilation easier, we have provided a CMake script which can be used to generate the build files for the target platform. CMake can be downloaded from http://www.cmake.org/cmake/resources/software.html.

In the following descriptions, "[AppName]" refers to the name of the application you want to compile. The source code of the sample applications is available at the location:

```
<InstallDir>/UniversalMessaging/cplus/examples/
```

On Windows, the Universal Messaging C++ libraries have been compiled using CMake and Microsoft Visual Studio 2015. To get the required environment for CMake you will need to run the Microsoft Visual Studio 2015 command prompt available via the "Visual Studio Tools" folder.

Windows 64 bit

Use the following commands at the Visual Studio 2015 command prompt:

1. cmake -Wno-dev -G "Visual Studio 14 Win64"
2. cd [AppName]
3. msbuild /p:Configuration=Release;PlatformType=x64 [AppName].sln
4. set PATH=%PATH%;[Location_Of_UM_Install]\cplus\lib\x86_64
5. Release\[AppName].exe

Environment Setup: Linux and OS X

The guide below explains how the C++ environment for Universal Messaging can be set up for compiling and running the applications on a Linux or OS X operating system.

> **Note:**
> Setting up the environment involves the use of the console at the operating system level. General information about using the console for this purpose is contained in the section "Running the Sample Applications" on page 11.

Running a C++ application for use with Universal Messaging requires the system to know the location of certain runtime libraries. OpenSSL is assumed to be installed and the location known to the system. The POCO libraries and `Nirvana.so` are found in `cplus/linux64/lib`. To make these libraries known to the system, several methods can be used:

1. By setting the LD_LIBRARY_PATH environment variable:

   ```
   export LD_LIBRARY_PATH=
      <InstallDir>/UniversalMessaging/cplus/linux/lib:$LD_LIBRARY_PATH
   ```

   This will allow programs to be compiled and run in the current shell.

2. In order to make the libraries globally available you can copy the libraries into `/usr/local/lib`.

3. Another method to make the libraries globally available is by using ldconfig. This requires root access to the system:

```
[root]$ cd /etc/ld.so.conf.d
[root]$ echo <InstallDir>/UniversalMessaging/cplus/linux/lib>nirvana.conf
[root]$ ldconfig
```

   The above code first navigates the required directory. It then creates a new file called `nirvana.conf` (this can be any file name with extension ".conf") containing the location of the libraries. Once this file is created, ldconfig is run (must be run as root) which creates the necessary links.

To compile a C++ application for use with Universal Messaging, the location of the shared libraries must be known by the system as described above. The compiler must also know the location of certain C++ headers. These headers are found in `cplus/include`. The `cplus/examples` directory contains sample applications written using the C++ API of Universal Messaging as well as the make files which can be used to compile them. In order to compile your own applications, please refer to these makefiles as a template. Each application comes pre-compiled, the executable (no file extension) can be found in the application's directory (`cplus/examples/applicationName`).

Compiling the Sample Applications

In order to make compilation easier, we have provided a CMake script which can be used to generate the build files for the target platform. CMake can be downloaded from http://www.cmake.org/cmake/resources/software.html.

In the following descriptions, "[AppName]" refers to the name of the application you want to compile. The source code of the sample applications is available at the location:

```
<InstallDir>/UniversalMessaging/cplus/examples/
```

Linux

On Linux, use the following commands at the console:

1. cmake .
2. cd [AppName]
3. make
4. export PATH=$PATH:[Location_Of_UM_Install]/cplus/lib/x86_64
5. ./[AppName]

OS X

On OS X, use the following commands at the console:

1. cmake .
2. cd [AppName]
3. make

4. export DYLD_LIBRARY_PATH=[Location_Of_UM_Install]/cplus/lib/x86_64
5. ./[AppName]

Enterprise Developer's Guide for C#

This guide describes how to develop and deploy C# .NET applications using Universal Messaging, and assumes you already have Universal Messaging installed.

General Features

Creating a Session for C#

To interact with a Universal Messaging Server, the first thing to do is create a Universal Messaging Session object, which is effectively your logical and physical connection to a Universal Messaging Realm. The steps below describe session creation.

1. Create a nSessionAttributes object with the RNAME value of your choice

   ```
   string[] RNAME={"nsp://127.0.0.1:9000"};
   nSessionAttributes nsa=new nSessionAttributes(RNAME);
   ```

2. Call the create method on nSessionFactory to create your session

   ```
   nSession mySession=nSessionFactory.create(nsa)
   ```

   Alternatively, if you require the use of a session reconnect handler to intercept the automatic reconnection attempts, pass an instance of that class too in the create method:

   ```
   public class myReconnectHandler : nReconnectHandler{
     //implement tasks associated with reconnection
   }
   myReconnectHandler rhandler=new myReconnectHandler();
   nSession mySession=nSessionFactory.create(nsa, rhandler);
   ```

3. Initialise the session object to open the connection to the Universal Messaging Realm

   ```
       mySession.init();
   ```

   To enable the use of DataGroups and to create an nDataStream , you should pass an instance of nDataStreamListener to the init call.

   ```
   public void SimpleStreamListener : nDataStreamListener{
     //implement onMessage callback for nDataStreamListener callbacks
   }
   nDataStreamListener myListener = new SimpleStreamListener();
   nDataStream myStream = mySession.init(myListener);
   ```

After initialising your Universal Messaging session, you will be connected to the Universal Messaging Realm. From that point, all functionality is subject to a Realm ACL check. If you call a method that requires a permission your credential does not have, you will receive an nSecurityException.

Universal Messaging Events

A Universal Messaging Event (nConsumeEvent) is the object that is published to a Universal Messaging channel or queue. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an Event Dictionary (see "Universal Messaging Event Dictionaries" on page 147).

Each nConsumeEvent object has an nEventAtrributes object associated with it which contains all available meta data associated with the event

Constructing an Event

In this C# .NET code snippet, we construct our Universal Messaging Event object (`nConsumeEvent`), and, in this example, pass a byte array data into the constructor:

```
nConsumeEvent evt = new nConsumeEvent( "String",
                   (new UTF8Encoding()).GetBytes("Hello World") );
```

Universal Messaging Event Dictionaries

Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within an "Universal Messaging Events" on page 147.

Event Dictionaries are quite similar to a hash table, supporting primitive types, arrays, and nested dictionaries.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's dictionary.

Event dictionaries can facilitate the automated purging of data from channels through the use of Publish Keys.

Constructing an Event

In this C# code snippet, we assume we want to publish an event containing the definition of a bond, say, with a name of "bond1":

```
nEventProperties props = new nEventProperties();
props.put("bondname", "bond1");
props.put("price", 100.00);
nConsumeEvent evt1 = new nConsumeEvent("atag", props);
```

Note that in this example code, we also create a new Universal Messaging Event object (`nConsumeEvent`) to make use of our Event Dictionary (`nEventProperties`).

Channel Joins

Joining a channel to another channel or queue allows you to set up content routing so that events published to the source channel will be passed on to the destination channel/queue automatically. Joins also support the use of filters, thus enabling dynamic content routing.

Please note that while channels can be joined to both channels and queues, queues cannot be used as the source of a join.

When creating a join there is one compulsory parameter and two optional ones. The compulsory parameter is the destination channel. The optional parameters are the maximum join hops and a filter to be applied to the join.

Creating and Deleting Channel Joins

For sample applications showing how to create and delete joins programmatically in C# .NET please see *Create Channel Join* and *Delete a Channel Join*.

Archive Joins

It is possible to archive messages from a given channel by using an archive join. To perform an archive join, the destination must be a queue in which the archived messages will be stored. An example of this can be seen below:

```
nChannelAttributes archiveAtr = new nChannelAttributes();
archiveAtr.setName(rchanName);
nQueue archiveQueue = mySession.findQueue(archiveAtr);
mySrcChannel.joinChannelToArchive(archiveQueue);
```

Inter-Cluster Joins

Inter-cluster joins are added and deleted in almost exactly the same way as normal joins. The only differences are that the two clusters must have an inter-cluster connection in place, and that since the clusters do not share a namespace, each channel must be retrieved from nodes in their respective clusters, rather than through the same node. For example :

```
nChannel cluster1chan1 = realmNode1.findChannel(channelattributes1);
nChannel cluster2chan1 = realmNode4.findChannel(channelattributes2);
cluster1chan1.joinChannel(cluster2chan1);
```

Related Links

For a conceptional overview of channel joins, see the section *Data Routing using Channel Joins* in the *Concepts* guide.

For a description of how to set up and manage channel joins, see the section *Creating Channel Joins* in the *Administration Guide*. The description details the usage based on the Enterprise Manager, but the same general principles apply if you are using the API.

Publish / Subscribe using Channel Topics

Creating a Channel

Channels can be created programmatically as detailed below, or they can be created using the Enterprise Manager.

In order to create a channel, first of all you must create an nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using an RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then begin creating the channel object. Channels have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them,

To create a channel, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setMaxEvents(0);
cattrib.setTTL(0);
cattrib.setType(nChannelAttributes.PERSISTENT_TYPE);
cattrib.setName("mychannel");
nChannel myChannel=mySession.createChannel(cattrib);
```

Now we have a reference to a Universal Messaging channel within the realm.

Note:
The set of permissible characters you can use to name a channel is described in the section *Creating Channels* in the Enterprise Manager section of the *Administration Guide*.

Finding a Channel

Finding a Universal Messaging Channel using the Universal Messaging C# .NET Client API

In order to find a channel programmatically you must create your nSession object, which is effectively your logical and physical connection to a Universal Messaging realm. This is achieved by using the correct RNAME for your Universal Messaging realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the channel object. Channels have an associated set of attributes, that define their behaviour within the Universal Messaging Realm Server. As well as the name of the channel, the attributes determine the availability of the events published to a channel to any subscribers wishing to consume them,

To find a channel previously created, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setName("mychannel");
nChannel myChannel=mySession.findChannel(cattrib);
```

Behavior if a channel has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "CHANNEL DELETED" and an event data component equal to "The channel has been deleted".

Publishing Events to a Channel

There are 2 types of publish available in Universal Messaging for channels:

Reliable Publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional Publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once the session has been established with the Universal Messaging realm server and the channel has been located, an event must be constructed prior to a publish call being made to the channel.

For reliable publish, there are a number of method prototypes on a channel that allow us to publish different types of events onto a channel. Here are examples of some of them. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
myChannel.publish(new nConsumeEvent("TAG", (new UTF8Encoding()).GetBytes(message)));
//Publishing a dictionary (nEventProperties)
nEventProperties props = new nEventProperties();
props.put("bondname", "bond1");
props.put("price", 100.00);
nConsumeEvent evt = new nConsumeEvent("atag", props);
myChannel.publish(evt);
// Publishing multiple messages in one publish call
List<nConsumeEvent> Messages = new List<nConsumeEvent>();
Messages.Add(message1);
Messages.Add(message2);
Messages.Add(message3);
myChannel.publish(Messages);
```

Transactional Publish

Transactional publishing provides a means of verifying that the server received the events from the publisher, and therefore provides guaranteed delivery.

There are similar prototypes available to the developer for transactional publishing. Once the session is established and the channel located, we then need to construct the events for the transaction and publish these events to the transaction. Only when the transaction has been committed will the events become available to subscribers on the channel.

Below is a code snippet for transactional publishing:

```
//Publishing a single event in a transaction
nTransactionAttributes attrib=new nTransactionAttributes(myChannel);
nTransaction myTransaction=nTransactionFactory.create(attrib);
myTransaction.publish(new nConsumeEvent("TAG", new UTF8Encoding()).GetBytes(message)));
myTransaction.commit();
//Publising multiple events in a transaction
List<nConsumeEvent> Messages = new List<nConsumeEvent>();
Messages.Add(message1);
nTransactionAttributes tattrib = new nTransactionAttributes(myChannel);
nTransaction myTransaction = nTransactionFactory.create(tattrib);
myTransaction.publish(Messages);
myTransaction.commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify that a transaction has been committed or aborted, a call can be made on the transaction that will determine if the events within the transaction were successfully received by the Universal Messaging Realm Server. This call can be made regardless of whether the connection was lost and a new connection was created.

The following code snippet demonstrates how to query the Universal Messaging Realm Server to see if the transaction was committed:

```
bool committed = myTransaction.isCommitted(true);
```

Subscribe Asynchronously to a Channel

Asynchronous channel subscribers consume events from a callback on an interface that all asynchronous subscribers must implement. We call this interface an nEventListener.

The listener interface defines one method called 'go' which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

A simple example of such a listener is shown below:

```
public class mySubscriber : nEventListener {
        public mySubscriber() {
                // construct your session and channel objects here
               // begin consuming events from the beginning of the channel (event id
 0)
                myChannel.addSubscriber(this, 0);
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event " + event.getEventID());
        }
        public static void Main(String[] args) {
                new mySubscriber();
        }
}
```

Subscription with a Filtering Selector

Asynchronous consumers can also be created using a selector, which allows the subscription to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the addSubscriber method shown in the example code, will ensure that the subscriber will only consume messages that contain the correct value for the event property BONDNAME.

Synchronous Consumers

Events can be synchronously consumed from a channel using a channel iterator object. The iterator will sequentially move through the channel and return events as and when the iterator getNext() method is called.

If you are using iterators so that you know when all events have been consumed from a channel please note that this can also be achieved using an asynchronous subscriber by calling the nConsumeEvent's isEndOfChannel() method.

An example of how to use a channel iterator is shown below:

```
public class myIterator {
        nChannelIterator iterator = null;
        public myIterator() {
                // construct your session and channel objects here
                // start the iterator at the beginning of the channel (event id 0)
                iterator = myChannel.createIterator(0);
        }
        public void start() {
                while (true) {
                        nConsumeEvent event = iterator.getNext();
                        go(event);
                }
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                myIterator itr = new myIterator();
                itr.start();
        }
}
```

Synchronous consumers can also be created using a selector, which defines a set of event properties and their values that a consumer is interested in. For example if events are being published with the following event properties:

```
nEventProperties props =new nEventProperties();
props.put("BONDNAME", "bond1");
```

If you then provide a message selector string in the form of:

```
String selector = "BONDNAME='bond1'";
```

And pass this string into the createIterator method shown in the example code, then your consumer will only consume messages that contain the correct value for the event property BONDNAME.

Batched Subscribe

If a client application needs to subscribe to multiple channels it is more efficient to batch these subscriptions into a single server call. This is achieved using the subscribe method of nSession rather than first finding the nChannel object and then calling the subscribe method of nChannel.

The following code snippet demonstrates how to subscribe to two Universal Messaging channels in one server call:

```
public class myEventListener implements nEventListener {
  public void go(nConsumeEvent evt) {
    Console.WriteLine("Received an event!");
  }
}
public void demo(){
  nSubscriptionAttributes[] arr = new nSubscriptionAttributes[2];
  arr[0] = new nSubscriptionAttributes("myChan1", "", 0, myLis1);
  arr[1] = new nSubscriptionAttributes("myChan2", "", 0, myLis2);
  arr = mySession.subscribe(arr);
```

```
  for (int i = 0; i < arr.length; i++) {
    if (!arr[i].wasSuccessful()) {
      handleSubscriptionFailure(arr[i]);
    }
    //subscription successful
  }
}
public void handleSubscriptionFailure(nSubscriptionAttributes subAtts){
  Console.WriteLine(subAtts.getException().StackTrace);
}
```

The nSubscriptionAttributes class is used to specify which channels to subscribe to. The second two parameters of the constructor represent the selector to use for the subscription and the event ID to subscribe from.

It is possible that the subscription may fail; for example, the channel may not exist or the user may not have the required privileges. In this situation, calling wasSuccessful() on the nSubscriptionAttributes will return false and getException() will return the exception that was thrown.

If the subscription is successful then the nChannel object can be obtained from the nSubscriptionAttributes as shown in the following code snippet:

```
  nChannel chan = subAtts.getChannel();
```

Batched Find

In client applications, it is quite common to have multiple Channels or Queues that one is trying to find. In these scenarios, the batched find call built into nSession is extremely useful.

The following code snippet demonstrates how to find 2 Universal Messaging Channels in one server call:

```
public void demo(){
  nChannelAttributes[] arr = new nChannelAttributes[2];
  nChannel[] channels = new nChannels[2];
  arr[0] = new nChannelAttributes("myChan1");
  arr[1] = new nChannelAttributes("myChan2");
  nFindResult[] results = mySession.find(arr);
  for (int i = 0; i < results.length; i++) {
    if (!results[i].wasSuccessful()) {
      handleSubscriptionFailure(results[i]);
    } else if (results[i].isChannel) {
      channels[i] = results[i].getChannel();
    }
  }
}
public void handleSubscriptionFailure(nFindResult result){
  Console.WriteLine(result.getException().StackTrace);
}
```

To perform the same operation for Queues, simply use the example above and exchange nChannel for nQueue, and check each result returned to see if the isQueue() flag is set.

Using Durable Objects

This section describes how to use *durable objects* and *shared durable objects* using the Universal Messaging API for C#.

Creating durable objects

For operations like creating, retrieving, deleting or unbinding a durable object, the nDurableManager must be used. Every channel has a durable manager associated with it. For information on how to use it, check the following examples.

Example 1

Getting the durable manager:

```
nSession session = nSessionFactory.create(new
nSessionAttributes("nsp://localhost:11000"));
session.init();
nChannelAttributes channelAttributes = new nChannelAttributes("testChannel");
nChannel channel = session.createChannel(channelAttributes);
nDurableManager durableManager = channel.getDurableManager();
```

Example 2

Creating durable objects using the durable manager instance from the example above:

```
nDurableAttributes exclusiveAttributes = nDurableAttributes.create(
    nDurableAttributes.nDurableType.Named, "exclusive");
nDurable exclusiveDurable = durableManager.add(exclusiveAttributes);
nDurableAttributes sharedAttributes = nDurableAttributes.create(
    nDurableAttributes.nDurableType.Shared, "shared");
nDurable sharedDurable = durableManager.add(sharedAttributes);
nDurableAttributes serialAttributes = nDurableAttributes.create(
    nDurableAttributes.nDurableType.Serial, "serial");
nDurable serialDurable = durableManager.add(serialAttributes);
```

Here we have listed the different options to create a new durable object. It can be either shared, serial or non-shared. The non-shared durable object can be a simple exclusive durable object where only one subscriber can be attached at a time.

Example 3

Deleting a durable object:

```
durableManager.delete(exclusiveDurable);
```

Example 4

Retrieving a durable object:

- Retrieving the durable object can be done using only the name of the durable object:

  ```
  nDurable returnedDurable = durableManager.get("shared");
  ```

- Or you can also retrieve all durables for the channel to which the durable manager corresponds:

  ```
  nDurable[] allDurables = durableManager.getAll();
  ```

Shared durables support purging events - by filter, by event id, by range of event ids or all of the events. The following methods defined on nDurable can be used. They provide an implementation only for shared durables:

```
void remove(long eid)

void remove(long start, long end)

void remove(string filter)

void removeAll()
```

For the other durable types an nIllegalStateException is thrown.

Creating a durable subscription

After durable subscriptions are created on the server, a consumer can be created using the nChannel instance. You can create both synchronous and asynchronous subscriptions.

For creating asynchronous durable consumers, the following methods can be used:

```
void addSubscriber(nEventListener listener, nDurable durable)

void addSubscriber(nEventListener listener, nDurable durable,
    string selector, bool autoAck)

void addSubscriber(nEventListener listener, nDurable durable,
    nNamedPriorityListener priorityListener)

void addSubscriber(nEventListener listener, nDurable durable,
    string selector, bool autoAck, nNamedPriorityListener priorityListener)
```

The API for creating synchronous durable subscribers is:

```
nChannelIterator createIterator(nDurable durable)

nChannelIterator createIterator(nDurable durable, string selector)
```

A public interface is available for committing and rolling back events. The methods are defined for the nDurable instance. To be able to apply these operations on a single event and not only on consecutive event IDs is of significant importance for the shared types.

Here are the methods defined in the nDurable class:

```
void acknowledge(nConsumeEventToken eventToken, bool isSynchronous)

void rollback(nConsumeEventToken eventToken, bool isSynchronous)
```

By default, these methods include previous outstanding events that were received in the same connection.

This functionality is extended for the durables which are shared. On an nSharedDurable instance you can also invoke:

```
void acknowledge(List<nConsumeEventToken> eventTokens, bool isSynchronous)

void acknowledge(nConsumeEventToken eventToken, bool isSynchronous,
```

```
    bool includePreviousEventsOutstanding)

void rollback(List<nConsumeEventToken> eventTokens, bool isSynchronous)

void rollback(nConsumeEventToken eventTocken, bool isSynchronous,
    bool includePreviousEventsOutstanding)
```

Note:
There is a difference in the behavior regarding the re-delivery of rolled back events. If the durable object is shared, events are redelivered immediately. But for exclusive durables, events are redelivered after re-subscribing.

The nConsumeEventToken is an abstraction for the event identifier and is built internally. The value can be retrieved using the public nConsumeEventToken getEventIdentifier() method defined for the nConsumeEvent instance.

nDurableViewer instances can be created for browsing the events on the durable object. Its API supports the following methods:

```
nConsumeEvent next()

void close()
```

for retrieving the next event and closing the viewer.

Restrictions

The nAdminAPI has no additional extensions for working with nDurable objects. For example, nDurableNode and nDurableConnectionNode, available in the client API for Java, are not ported in the client API for C# .

API Support for Reactive Extensions (Rx)

The Universal Messaging native client API provides support for Reactive Extensions (Rx). This support provides the ability to create shared or exclusive / named durable subscribers using the following method on the topic retrieved from the initialized session:

```
ITransactionalConsumer CreateDurableConsumer(
   string name, string unique, DurableType durableType,
   string filter, bool subscribeToPurge = false)
```

A DurableType enumeration has been added with the available options: Named and Shared.

See the following example:

```
using (var session = new Session("nsp://localhost:11000"))
  {
     // Initialize the session
     session.Initialize();
     // Create consumer & subscribe
     consumer = session.Topics.CreateDurableConsumer(
         "channel", "durable", DurableType.Named, "filter");
     var query =
         from e in consumer.ToObservable()
```

```
         select e.Message;
     // Subscribe
     query.Subscribe(ProcessMessage);
  }

 // Example function to process messages
void ProcessMessage(IMessage message)
{
  if (message != null)
 {
  Console.WriteLine("Message received {0}.", message.Id);
 }
}
```

The consumer implements ITransactionalConsumer, so you would be able to commit or roll back events by simply invoking consumer.commit() or consumer.rollback(), which will commit or roll back all of the unacknowledged events up to and including the last received event.

The Merge Engine and Event Deltas

In order to streamline publish/subscribe applications it is possible to deliver only the portion of an event's data that has changed rather than the entire event. These event deltas minimise the amount of data sent from the publisher and ultimately delivered to the subscribers.

The publisher simply registers an event and can then publish changes to individual keys within the event. Subscribers can be configured to get callbacks which contain either the entire event or just the changed key(s). Either way, only the key(s) that have changed are delivered to the subscribing client.

Publisher - Registered Events

In order to publish event deltas the publisher uses the Registered Event facility available on a Universal Messaging Channel. Please note that the channel must have been created with the Merge Engine and it must have a single Publish Key. The publish key represents the primary key for the channel and the registered events. So for example if you are publishing currency rates you would setup a channel as such

```
   nChannelAttributes cattr
            = new nChannelAttributes("RatesChannel", 0, 0,
nChannelAttributes.SIMPLE_TYPE);
//
// This next line tells the server to Merge incoming events based on the publish
//  key name and the name of the registered event
//
        cattr.useMergeEngine(true);
//
// Now create the Publish Key (See publish Keys for a full description
//
        nChannelPublishKeys[] pks = new nChannelPublishKeys[1];
        pks[0] = new nChannelPublishKeys("ccy", 1);
        cattr.setPublishKeys(pks);
//
// Now create the channel
//
        myChannel = mySession.createChannel(cattr);
```

At this point the server will have a channel created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent audEvent = myChannel.createRegisteredEvent("AUD");
        nEventProperties props = audEvent.getProperties();
        props.put("bid", 0.8999);
        props.put("offer", 0.9999);
        props.put("close", "0.8990");
        audEvent.commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to a ccy value of "AUD". We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props.put("bid", 0.9999);
        audEvent.commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

Subscriber - nEventListener

The subscriber implements nEventListener in the usual way and does not need to do anything different in order to receive either event deltas or snapshots containing the result of one or more merge operations. The standard nEventListener will receive a full event when the subscription is initiated. Thereafter it will receive only deltas. If at any time the user is disconnected then it will receive a fresh update of the full event on reconnection - followed by a resumption of delta delivery.

If you wish to differentiate between snapshot events and delta events then the nConsumeEvent attributes can be used as follows:

```
event.getAttributes().isDelta();
```

For more information on Universal Messaging publish / subscribe, please see the API documentation.

Consuming a JMS Map Message

In order to enable Universal Messaging to support JMS, message types for JMS are stored in a slightly different way from the normal nConsumeEvent.

When a Java client publishes a JMS Map Message, the map is serialised and stored in the payload of the message. For a C# subscriber to consume a JMS Map Message, this payload must be reconstructed as an nEventProperties using the getPayloadAsDictionary method.

Consuming a Map Message

A JMS map message will be received in the go callback in the same way as a normal nConsumeEvent. Once received, the Map Message can be handled as follows:

```
    go(nConsumeEvent evt){
      if(evt.getAttributes().getType()==nEventAttributes.MapMessageType){
        nEventProperties map = evt.getPayloadAsDictionary();
      }
    }
```

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same channel or queue. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages. The priority is a numeric value in the range 0 (lowest priority) to 9 (highest priority).

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent evt;
    ...
evt.getAttributes().setPriority(9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically, events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis, and the effects become more noticeable as load increases.

Note:
If events are stored for replay at a later stage, for example for a durable subscriber who is currently not consuming events, higher priority events will be delivered earlier than lower priority events when the durable subscriber starts consuming the events, even if the lower priority events were created much earlier .

Publish / Subscribe using Data Streams and Data Groups

Publish / Subscribe Using Data Streams and Data Groups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging Data Groups are lightweight structures designed to facilitate Publish/Subscribe . When using Data Groups, user subscriptions are managed remotely in a way that is transparent to subscribers. Universal Messaging Channels provide an alternative style of Publish/Subscribe where the subscribers manage their subscriptions directly.

This section demonstrates Universal Messaging pub / sub using Data Groups in C#, and provides example code snippets for all relevant concepts.

Enabling Data Groups and Receiving Event Callbacks

DataStreamListener

If an nSession is created with an nDataStreamListener then it will receive asynchronous callbacks via the onMessage implementation of the nDataStreamListener interface. The nDataStreamListener will receive events when:

- An event is published directly to this particular nDataStream
- An event is published to any nDataGroup which contains this nDataStream
- An event is published to an nDataGroup which contains a nested nDataGroup containing this nDataStream
- An example of how to create a session with an nDataStreamListener interface is shown below:

```
public class DataGroupClient : nDataStreamListener{

    nSession mySession;
    public DataGroupClient( string realmURLs){
        nSessionAttributes nsa = new nSessionAttributes(realmURLs);
        mySession = nSessionFactory.create(nsa, this);
        mySession.init(this);
    }

    ////
    // nDataStreamListener Implementation
    ////

    //Callback received when event is available
    public void onMessage(nConsumeEvent event){

            //some code to process the message

    }
}
```

Managing Data Groups

Creating and Deleting DataGroups

Creating Universal Messaging DataGroups

nDataGroups can be created programmatically as detailed below, or they can be created using the Universal Messaging enterprise manager.

In order to create a nDataGroup, first of all you must create an nSession object with an nDataStreamListener. This is effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using an RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
string[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa=new nSessionAttributes(RNAME);
nSession mySession=nSessionFactory.create(nsa);
mySession.init(this); // where this is an nDataStreamListener
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession object instance 'mySession', you can then create DataGroups. The create DataGroup methods will return the nDataGroup if it already exists.

The code snippets below demonstrate the creation of nDataGroups:

Create a Single nDataGroup

```
nDataGroup myGroup = mySession.createDataGroups("myGroup");
```

Create Multiple nDataGroups

```
string[] groups = {"myFirstGroup", "mySecondGroup"};
IEnumerable<nDataGroup> myGroups = mySession.createDataGroups(groups);
```

Creating DataGroups with DataGroupListeners and ConflationAttributes

It is also possible to specify additional properties when creating DataGroups:

- nDataGroupListener - To specify a listener for DataGroup membership changes
- nConflationAttributes - To specify attributes which control event merging and delivery throttling for the DataGroup

Now we have a reference to a Universal Messaging DataGroup it is possible to publish events

Deleting Universal Messaging DataGroups

There are various deleteDataGroup methods available on nSession which will delete DataGroups. It is possible to specify single nDataGroups or arrays of nDataGroups.

Managing DataGroup Membership

DataGroups are extremely lightweight from both client and server perspectives; a back-end process, such as a Complex Event Processing engine, can simply create DataGroups and then add or remove users (or even entire nested DataGroups) based on bespoke business logic. A user who is removed from one DataGroup and added to another will continue to receive events without any interruption to service, or indeed explicit awareness that any DataGroup change has occurred.

This page details some of the typical operations that DataGroup management process would carry out. Please see our C# sample apps for more detailed examples of DataGroup management.

Tracking Changes to DataGroup Membership (DataGroupListener)

The nDataGroupListener interface is used to provide asynchronous notifications when nDataGroup membership changes occur. Each time a user (nDataStream) or nDataGroup is added or removed from a nDataGroup a callback will be received.

```
public class datagroupListener : nDataGroupListener {
    nSession mySession;
    public datagroupListener(nSession session){
        mySession = session;
        //add this class as a listener for all nDataGroups on this Universal
        //Messaging realm
        mySession.getDataGroups(this);
    }

    ////
    //DataGroupListener Implementation
    ///
    public void addedGroup (nDataGroup to, nDataGroup group, int count){
        //Called when a group has been added to the 'to' data group.
        //count is the number of nDataStreams that will receive any events published
 to
        //this nDataGroup
    }

    public void addedStream (nDataGroup group, nDataStream stream, int count){
        //Called when a new stream has been added to the data group.
    }

    public void createdGroup (nDataGroup group){
        //Called when a group has been created.
    }

    public void deletedGroup (nDataGroup group){
        //Called when a group has been deleted.
    }

    public void deletedStream (nDataGroup group, nDataStream stream, int count,
                               boolean serverRemoved){
        //Called when a stream has been deleted from the data group.
       //serverRemoved is true if the nDataStream was removed because of flow control
    }

    public void removedGroup (nDataGroup from, nDataGroup group, int count){
        //Called when a group has been removed from the 'from' data group.
    }

}
```

There are three ways in which the nDataGroupListener can be used:

Listening to an individual DataGroup

Listeners can be added to individual DataGroups when they are created or at any time after creation. The code snippets illustrate both approaches:

```
mySession.createDataGroup(dataGroupName, datagroupListener);
```

```
myDataGroup.addListener(datagroupListener);
```

Listening to the Default DataGroup

The Default nDataGroup is a DataGroup to which all nDataStreams are added by default. If you add a DataGroupListener to the defaiult DataGroup then callbacks will be received when:

- a nDataStream is connected/disconnected
- a nDataGroup is created or deleted

Listening to all DataGroups on a Universal Messaging Realm

The code snippet below will listen on all nDataGroups (including the default DataGroup).

```
mySession.getDataGroups(datagroupListener);
```

Adding and Removing DataGroup Members

The nDataGroup class provides various methods for adding and removing nDataStreams and nDataGroups. Please see the nDataGroup API documentation for a full list of methods. Examples of some of these are provided below:

```
    //Add a nDataStream (user) to a nDataGroup
    public void addStreamToDataGroup(nDataGroup group, nDataStream user){
        group.add(user);
    }
    //Remove a nDataStream (user) from a nDataGroup
    public void removeStreamFromDataGroup(nDataGroup group, nDataStream user){
        group.remove(user);
    }
    //Add a nDataGroup to a nDataGroup
    public void addNestedDataGroup(nDataGroup parent, nDataGroup child){
        parent.add(child);
    }
    //Remove a nDataGroup from a nDataGroup
    public void removeNestedDataGroup(nDataGroup parent, nDataGroup child){
        parent.remove(child);
    }
```

DataGroup Conflation Attributes

Enabling Conflation on DataGroups

Universal Messaging DataGroups can be configured so that conflation (merging and throttling of events) occurs when messages are published. Conflation can be carried out in several ways and these are specified using an nConflationAttributes object. The ConflationAttributes object is passed in to the DataGroup when it is created initially.

The nConflationAttributes object has two properties action and interval. Both of these are passed into the constructor.

The action property specifies whether published events should replace previous events in the DataGroup or be merged with them. These properties are defined by static fields:

```
nConflationAttributes.sMergeEvents
nConflationAttributes.sDropEvents
```

The interval property specifies the interval in milliseconds between event fanout to subscribers. An interval of zero implies events will be fanned out immediately.

Creating a Conflation Attributes Object

```
//ConflationAttributes specifying merge events  and no throttled delivery
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 0);
//ConflationAttributes specifying merge events and throttled delivery at
// 1 second intervals
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 1000);
//ConflationAttributes specifying drop events and throttled delivery at
// 1 second intervals
nConflationAttributes confattribs =
   new nConflationAttributes(nConflationAttributes.sDropEvent, 1000);
```

Create a Single nDataGroup with Conflation Attributes

```
public class datagroupListener : nDataGroupListener {
    nSession mySession;
    nDataGroup myDataGroup;
    public datagroupListener(nSession session, nConflationAttributes confattribs,
           string dataGroupName){
        mySession = session;
        //create a DataGroup passing in this class as a nDataGroupListener and
        //a ConflationAttributes
        myDataGroup = mySession.createDataGroup(dataGroupName, this, confattribs);
    }
}
```

Create Multiple nDataGroups with Conflation Attributes

```
nConflationAttributes confattribs =
```

```
   new nConflationAttributes(nConflationAttributes.sMergeEvents, 1000);
string[] groups = {"myFirstGroup", "mySecondGroup"};
nDataGroup[] myGroups = mySession.createDataGroups(groups, confattribs);
```

Publishing Events to Conflated DataGroups With A Merge Policy

At this point the server will have a nDataGroup created with the ability to merge incoming events from Registered Events. The next step is to create the Registered events at the publisher.

```
        nRegisteredEvent audEvent = myDataGroup.createRegisteredEvent();
        nEventProperties props = audEvent.getProperties();
        props.put("bid", 0.8999);
        props.put("offer", 0.9999);
        props.put("close", "0.8990");
        audEvent.commitChanges();
```

You now have a nRegisteredEvent called audEvent which is bound to a data group that could be called 'aud'. We then set the properties relevant to the application, finally we call commitChanges(), this will send the event, as is, to the server. At this point if the bid was to change then that individual field can be published to the server as follows:

```
        props.put("bid", 0.9999);
        audEvent.commitChanges();
```

This code will send only the new "bid" change to the server. The server will modify the event internally so that any new client subscribing will receive all of the data, yet any existing subscribers will only receive the change.

When a data group has been created with Merge conflation, all registered events published to that data group will have their nEventProperties merged into the snapshot event, before the delta event is delivered to the consumers.

When using Merge conflation with an interval (i.e. throttling), all updates will be merged into a conflated event (as well as the snapshot event) that will be delivered within the chosen interval. For example, consider the following with a merge conflated group and an interval set to 100ms (ie maximum of 10 events a second):

```
Scenario 1
t0   - Publish Message1, Bid=1.234     (This message will be immediately
                                        delivered, and merged into the snapshot)
t10  - Publish Message2, Offer=1.234   (This message will be held as a
                                        conflation event, and merged into the
                                        snapshot)
t20  - Publish Message3, Bid=1.345     (This message will be merged with the
                                        conflated event, and with the snapshot)
t100 - Interval hit                    (Conflated event containing
                                        Offer=1.234,Bid=1.345
                                        is delivered to consumers)
                                        Interval timer reset to +100ms, ie t200
t101    - Publish Message4, Offer=1.345 (This message will be held as a
                                         conflation event,
                                         and merged into the snapshot)
Where t0...tn is the time frame in milliseconds from now.
Scenario 2
t0   - Publish Message1, Bid=1.234      (This message will be immediately
                                         delivered, and merged into the snapshot)
```

```
t100 - Interval hit                      (Nothing is sent as there has been no
                                          update since t0)
t101 - Publish Message2, Offer=1.234     (This message will be immediately
                                          delivered, and merged into the snapshot)
                                          Interval timer reset to +100ms, ie t201
```

Meanwhile, if any new consumers are added to the Data Group, they will always consume the most up to date snapshot and then begin consuming any conflated updates after that.

Publishing Events to Conflated DataGroups With A Drop Policy

If you have specified a "Drop" policy in your ConflationAttributes then events are published in the normal way rather than using nRegisteredEvent.

Consuming Conflated Events from a DataGroup

The subscriber doesn't need to do anything different to receive events from a DataGroup with conflation enabled. If nRegisteredEvents are being delivered then the events will contain only the fields that have changed will be delivered. In all other circumstances an entire event is delivered to all consumers.

Publishing to Data Groups

DataGroups Event Publishing

You can get references to any DataGroup from the nSession object. There are various writeDataGroup methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataGroups.

```
myDataGroup = mySession.getDataGroup("myGroup");
nEventProperties props = new nEventProperties();
//You can add other types in a dictionary object
props.put("key0string"+x, "1"+x);
props.put("key1int", (int) 1);
props.put("key2long", (long) -11);
nConsumeEvent evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession.writeDataGroup(evt1, myDataGroup);
```

DataStream Event Publishing

You can get references to any nDataStream (user) from the nSession object if you call getDefaultDataGroup(). You can also access nDataStreams by implementing the nDataGroupListener interface. Please see DataGroup management for more information. This will deliver callbacks as users are connected/disconnected. There are various writeDataStream methods available. These methods also support batching of multiple events to a single group or batching of writes to multiple DataStreams.

```
nEventProperties props = new nEventProperties();
//You can add other types in a dictionary object
props.put("key0string"+x, "1"+x);
props.put("key1int", (int) 1);
```

```
props.put("key2long", (long) -11);
nConsumeEvent evt1 = new nConsumeEvent(props, buffer);
//Publish the event
mySession.writeDataStream(evt1, myDataStream)
```

Using Priority Messaging

For general information about how priority messaging works in Universal Messaging, see the section "Commonly Used Features" > "Priority Messaging" in the *Concepts* guide.

In certain scenarios it may be desirable to deliver messages with differing levels of priority over the same datagroup. Universal Messaging provides the ability to expedite messages based on a priority level. Messages with higher levels of priority are able to be delivered to clients ahead of lower priority messages. The priority is a numeric value in the range 0 (lowest priority) to 9 (highest priority).

Universal Messaging achieves this capability through a highly concurrent and scalable implementation of a priority queue. Where in a typical queue events are first in first out, in a priority queue the message with the highest priority is the first element to be removed from the queue. In Universal Messaging each client has its own priority queue for message delivery.

The following code snippet demonstrates how to set priority on a message:

```
nConsumeEvent evt;
    ...
evt.getAttributes().setPriority(9);
```

Priority Messaging allows for a high priority message to be delivered ahead of a backlog of lower priority messages. Ordering of delivery is done dynamically on a per client basis.

Priority messaging is enabled by default, there are no configuration options for this feature.

As Priority Messaging is done dynamically, events may not appear in strict order of priority. Higher priority events are expedited on a best effort basis, and the effects become more noticeable as load increases.

Note:
If events are stored for replay at a later stage, for example for a durable subscriber who is currently not consuming events, higher priority events will be delivered earlier than lower priority events when the durable subscriber starts consuming the events, even if the lower priority events were created much earlier .

Message Queues

Message Queues

Universal Messaging provides message queue functionality through the use of queue objects. Queues are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Message queues differ from publish / subscribe channels in the way that events are delivered to consumers. Whilst queues may have multiple consumers, each event is typically only delivered to one consumer, and once consumed (popped) it is removed from the queue.

Universal Messaging also supports non destructive reads (peeks) from queues, which enable consumers to see what events are on a queue without removing them from the queue. Any event which has been peeked will still be queued for popping in the normal way. The Universal Messaging Enterprise Manager also supports the ability to visually peek a queue using its snoop capability.

This section demonstrates how Universal Messaging message queues work, and provide example code snippets for all relevant concepts.

Creating a Queue

In order to create a queue, first of all you must create your nSession object, which is your effectively your logical and physical connection to a Universal Messaging Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME={"nsp://127.0.0.1:9000"};
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nSession mySession = nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

We can use the nSession object instance mySession to create the queue object. Queues have an associated set of attributes that define their behaviour within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them,

To create a queue, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setChannelMode(nChannelAttributes.QUEUE_MODE);
cattrib.setMaxEvents(0);
cattrib.setTTL(0);
cattrib.setType(nChannelAttributes.PERSISTENT_TYPE);
cattrib.setName("myqueue");
nQueue myQueue = mySession.createQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Note:
For information about the set of permissible characters you can use to name a queue, see the section *Creating Queues* in the Enterprise Manager section of the *Administration Guide*.

Finding a Queue

In order to find a queue, first of all the queue must be created. This can be achieved through the Universal Messaging Enterprise Manager, or programmatically. First of all you must create your nSession object, which is effectively your logical and physical connection to a Universal Messaging

Realm. This is achieved by using the correct RNAME for your Universal Messaging Realm when constructing the nSessionAttributes object, as shown below:

```
String[] RNAME = ({"nsp://127.0.0.1:9000"});
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nSession mySession = nSessionFactory.create(nsa);
mySession.init();
```

Once the nSession.init() method is successfully called, your connection to the realm will be established.

Using the nSession objects instance 'mySession', we can then try to find the queue object. Queues have an associated set of attributes, that define their behavior within the Universal Messaging Realm Server. As well as the name of the queue, the attributes determine the availability of the events published to a queue to any consumers wishing to consume them,

To find a queue previously created, we do the following:

```
nChannelAttributes cattrib = new nChannelAttributes();
cattrib.setName("myqueue");
nQueue myQueue = mySession.findQueue(cattrib);
```

Now we have a reference to a Universal Messaging queue within the realm.

Behavior if a queue has been deleted

Whenever a store (i.e. a channel or a queue) is deleted, all clients will be advised that the store has changed, and will mark the object as invalidated, thus stopping any further use of the object. At this point all producing and consuming to the server will have ceased. If the store has been recreated, you must go and find the store and create any durable subscriptions as if it was a fresh start.

See the section *Deleting Channels and Queues* in the *Administration Guide* for further details.

Note:
Since *editing* a store involves deleting the old store and creating a new store, the behavior described for deleting a store applies also to editing a store.

There will be exceptions raised whenever an invalidated object is used to attempt to undertake any function, since the public signatures of methods have not changed, and thus not all methods will raise an exception but most will.

Any asynchronous consumer will receive a special event upon deletion of a store. The event will have an event identifier of -2, an event tag containing "QUEUE DELETED" and an event data component equal to "The queue has been deleted".

Publishing events to a Queue

There are 2 types of publish available in Universal Messaging for queues:

Reliable Publish is simply a one way push to the Universal Messaging Server. This means that the server does not send a response to the client to indicate whether the event was successfully received by the server from the publish call.

Transactional Publish involves creating a transaction object to which events are published, and then committing the transaction. The server responds to the transaction commit call indicating if it was successful. There are also means for transactions to be checked for status after application crashes or disconnects.

Reliable Publish

Once the session has been established with the Universal Messaging realm server and the queue has been located, an event must be constructed prior to a publish call being made to the queue.

The following code snippet shows how to reliably publish events to a queue. Further examples can be found in the API documentation.

```
// Publishing a simple byte array message
myChannel.publish(new nConsumeEvent("TAG", (new UTF8Encoding()).GetBytes(message)));
// Publishing multiple messages in one publish call
List Messages = new List();
Messages.Add(message1);
Messages.Add(message2);
Messages.Add(message3);
myChannel.publish(Messages);
```

Transactional Publish

Transactional publishing provides us with a method of verifying that the server receives the events from the publisher, and provides guaranteed delivery.

There are similar prototypes available to the developer for transaction publishing. Once we have established our session and our queue, we then need to construct our events and our transaction, then publish these events to the transaction. The transaction will then be committed and the events available to consumers to the queue.

Below is a code snippet demonstrating transactional publishing:

```
List Messages = new List();
Messages.Add(message1);
nTransactionAttributes tattrib = new nTransactionAttributes(myChannel);
nTransaction myTransaction = nTransactionFactory.create(tattrib);
myTransaction.publish(Messages);
myTransaction.commit();
```

If during the transaction commit your Universal Messaging session becomes disconnected, and the commit call throws an exception, the state of the transaction may be unclear. To verify whether a transaction has been committed or aborted, the transaction can be queried to determine whether the events within the transactional were successfully received by the Universal Messaging Realm Server:

```
bool committed = myTransaction.isCommitted(true);
```

Examples

For more information on Universal Messaging Message Queues, please see the API documentation.

Asynchronously Consuming a Queue

Asynchronous queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called *go* which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

An example of an asynchronous queue reader is shown below:

```
public class myAsyncQueueReader : nEventListener {
        nQueue myQueue = null;
        public myAsyncQueueReader() {
                // construct your session and queue objects here
                // begin consuming events from the queue
                nQueueReaderContext ctx = new nQueueReaderContext(this, 10);
                nQueueAsyncReader reader = myQueue.createAsyncReader(ctx);
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                new myAsyncQueueReader();
        }
}
```

Subscription with a Filtering Selector

Asynchronous queue consumers can also be created using a selector, which allows the subscription to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the constructor for the nQueueReaderContext object shown in the example code will ensure that the subscriber will only consume messages that contain the correct value for the event property BONDNAME.

Synchronously Consuming a Queue

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

An example of a synchronous queue reader is shown below:

```
public class mySyncQueueReader {
        nQueueSyncReader reader = null;
        nQueue myQueue = null;
```

```
        public mySyncQueueReader() {
                // construct your session and queue objects here
                // construct the queue reader
                nQueueReaderContext ctx = new nQueueReaderContext(this, 10);
                reader = myQueue.createReader(ctx);
        }
        public void start() {
                while (true) {
                        // pop events from the queue
                        nConsumeEvent event = reader.pop();
                        go(event);
                }
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                mySyncQueueReader sqr = new mySyncQueueReader();
                sqr.start();
        }
}
```

Subscription with a Filtering Selector

Synchronous queue consumers can also be created using a selector, which allows the subscription to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the constructor for the nQueueReaderContext object shown in the example code will ensure that the subscriber will only consume messages that contain the correct value for the event property BONDNAME.

Asynchronous Transactional Queue Consumption

Asynchronous transactional queue consumers consume events from a callback on an interface that all asynchronous consumers must implement. We call this interface an nEventListener. The listener interface defines one method called go which when called will pass events to the consumer as they are delivered from the Universal Messaging Realm Server.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional asynchronous queue reader is shown below:

```
public class myAsyncTxQueueReader : nEventListener {
```

```
        nQueueAsyncTransactionalReader reader = null;
        nQueue myQueue = null;
        public myAsyncTxQueueReader() {
                // construct your session and queue objects here
                // begin consuming events from the queue
                nQueueReaderContext ctx = new nQueueReaderContext(this, 10);
                reader = myQueue.createAsyncTransactionalReader(ctx);
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
                reader.commit();
        }
        public static void Main(String[] args) {
                new myAsyncTxQueueReader();
        }
}
```

As previously mentioned, the big difference between a transactional asynchronous reader and a standard asynchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with Event IDs 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Subscription with a Filtering Selector

Asynchronous queue consumers can also be created using a selector, which allows the subscription to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the constructor for the nQueueReaderContext object shown in the example code will ensure that the subscriber will only consume messages that contain the correct value for the event property BONDNAME.

Synchronous Transactional Queue Consumption

Synchronous queue consumers consume events by calling pop() on the Universal Messaging queue reader object. Each pop call made on the queue reader will synchronously retrieve the next event from the queue.

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

An example of a transactional synchronous queue reader is shown below:

```
public class mySyncTxQueueReader {
        nQueueSyncTransactionReader reader = null;
        nQueue myQueue = null;
        public mySyncTxQueueReader() {
                // construct your session and queue objects here
                // construct the transactional queue reader
                nQueueReaderContext ctx = new nQueueReaderContext(this, 10);
                reader = myQueue.createTransactionalReader(ctx);
        }
        public void start() {
                while (true) {
                        // pop events from the queue
                        nConsumeEvent event = reader.pop();
                        go(event);
                        // commit each event consumed
                        reader.commit(event.getEventID());
                }
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                mySyncTxQueueReadersqr = new mySyncTxQueueReader();
                sqr.start();
        }
}
```

As previously mentioned, the big difference between a transactional synchronous reader and a standard synchronous queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

Developers can also call the rollback() method on a transactional reader that will notify the server that any events delivered to the reader that have not been committed, will be rolled back and redelivered to other queue consumers. Transactional queue readers can also commit or rollback any specific event by passing the event id of the event into the commit or rollback calls. For example, if a reader consumes 10 events, with Event IDs 0 to 9, you can commit event 4, which will only commit events 0 to 4 and rollback events 5 to 9.

Subscription with a Filtering Selector

Synchronous queue consumers can also be created using a selector, which allows the subscription to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the constructor for the nQueueReaderContext object shown in the example code will ensure that the subscriber will only consume messages that contain the correct value for the event property BONDNAME.

Browse (Peek) a Universal Messaging Queue

Universal Messaging provides a mechanism for browsing (peeking) queues. Queue browsing is a non-destructive read of events from a queue. The queue reader used by the peek will return an array of events, the size of the array being dependent on how many events are in the queue, and the window size defined when your reader context is created. For more information, please see the Universal Messaging Client API documentation.

An example of a queue browser is shown below:

```
public class myQueueBrowser {
        nQueueReader reader = null;
        nQueuePeekContext ctx = null;
        nQueue myQueue = null;
        public myQueueBrowser() {
                // construct your session and queue objects here
                // create the queue reader
                reader = myQueue.createReader(new nQueueReaderContext());
                ctx = nQueueReader.createContext(10);
        }
        public void start() {
                bool more = true;
                long eid =0;
                while (more) {
                        // browse (peek) the queue
                        nConsumeEvent[] evts = reader.peek(ctx);
                        for (int x=0; x < evts.Length; x++) {
                                go(evts[x]);
                        }
                        more = ctx.hasMore();
                }
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                myQueueBrowser qbrowse = new myQueueBrowser();
                qbrowse.start();
        }
}
```

Subscription with a Filtering Selector

Queue browsers can also be created using a selector, which allows the peek to be filtered based on event properties and their values.

For example, assume some events are being published with the following event properties:

```
nEventProperties props = new nEventProperties();
props.put("BONDNAME", "bond1");
```

A developer can create a message selector string such as:

```
String selector = "BONDNAME='bond1'";
```

Passing this string into the constructor for the nQueuePeekContext object shown in the example code will ensure that the browser will only receive messages that contain the correct value for the event property BONDNAME.

For more information on Universal Messaging Message Queues, please see the API documentation.

Request Response

Universal Messaging can easily be used to issue request/response message exchanges. To accomplish this, the requester simply publishes an event to a request queue and then listens for a response to be issued on a response queue. The responder tags this response with the username of the requester, and this ensures that only the requester will see the response event.

Requester

The requester publishes an event to a request queue and then listens for a response to be issued on a response queue. The response will be tagged with the username of the requester. This is specified during the initial configuration of the session, as shown below:

```
mySession = nSessionFactory.create(nsa, this,"subscriber tag");
```

After setting this, the requester simply publishes an event to the request queue and listens for a response on the response queue. An example C# .NET requester is available in the examples section.

Responder

The responder listens to the request channel and responds to each request event. To ensure the message is only delivered to the correct recipient, the Subscriber Name must be set on the response event. The response event's data can contain the relevant information the user needs.

```
//Having received a request event req,
//and established a connection to a response queue respQueue.
Console.WriteLine("Received request");

//Retrieve username of request sender.
String requester = req.getPublishUser();
//Construct repsponse message.
String text = "Response: " + new String(req.getEventData());
//Construct response event
nEventProperties atr = new nEventProperties();
nConsumeEvent resp = new nConsumeEvent(atr, text.getBytes());
//Set recipient of the event to the requester's tag to response.
resp.setSubscriberName(requester.getBytes());
respQueue.push(resp);
```

An example C# .NET responder is available in the examples section.

Basic Authentication

Overview

The entire set of session creation methods of the Universal Messaging client, admin APIs and reactive extension in .NET have overloaded variants that accept username/password credentials which are then supplied to the Universal Messaging server.

To use these overloaded variants, the external SASL library (`S22_SASL.dll`) must be loaded and its assembly must be available to the Universal Messaging API, as described in the "Prerequisites for Basic Authentication" on page 178 page.

Note that authentication does not supplant the traditional Universal Messaging ACLs and is merely an additional security step performed before the relevant ACLs are evaluated and applied.

The configuration for .NET authentication is controlled by either a set of environment variables or directly in the API via the com.pcbsys.nirvana.client.nSessionAttributes structure.

Prerequisites for Basic Authentication

In order to enable SASL authentication in .NET, the library `S22_SASL.dll` and its dependency `BouncyCastle.Crypto.dll` must be made available in the same application domain as the Universal Messaging library. The easiest away to do this is to add a reference to the `S22_SASL.dll` library whenever you have a reference to any Universal Messaging .NET libraries. Alternatively the assembly can be loaded into the current application domain using `System.Reflection.Assembly.Load()` and supplying the path to the `S22_SASL.dll` file.

The files `S22_SASL.dll` and `BouncyCastle.Crypto.dll` are located in the same `dotnet/bin` directory as the other DLL files for .NET.

Once the assembly is available, it will be automatically picked up when creating a session (whether via a client session, realm node or RX session) and used to connect to the server. If the assembly is unavailable and authentication credentials are supplied, an exception will be thrown when attempting to connect to the server, stating that no SASL implementations are available.

If authentication is not enabled on the server, the client will default to a standard connection without authentication.

Client-side Authentication

Authentication methods will only be used if a password is supplied when creating a session to the server. The SASL implementation for Universal Messaging in .NET supports the following mechanisms: plain (plain text username/password authentication), CRAM-MD5 and Digest-MD5 (cryptographically encoded credential authentication). The preferred mechanism can be set either via an API call or an environment variable as detailed below.

Setting the preferred authentication mechanism via Environment Configuration

There are a number of environment variables which may be used to control the authentication behaviour of the .NET API:

- *Nirvana.sasl.client.mech*

  This specifies which SASL mechanism to use, and the supported options are PLAIN, CRAM-MD5 and DIGEST-MD5.

  The mechanism defaults to PLAIN if this system property is not set, and the usual SASL trade-offs apply. PLAIN transmits the user password in plain text, so it is advisable to only use it over an SSL connection. On the other hand, CRAM-MD5 and DIGEST-MD5 do not transmit the password in plain text so are more appropriate for general connections.

  Note that if the preferred mechanism is set via nSessionAttributes, the API-set value will be preferred over this one.

- *Nirvana.sasl.client.enablePrehash*

  This specifies whether to prehash the supplied password when using the CRAM-MD5 or DIGEST-MD5 mechanisms. It may be set to "true" or "false". This should be set to "true" only when the server is using the fSAGInternalUserRepositoryAdapter to store client credentials, otherwise CRAM-MD5 and DIGEST-MD5 authentication will fail. If `Nirvana.sasl.client.enablePrehash` is not set, then the value defaults to "false" and prehashing is not enabled.

Setting the preferred authentication mechanism via API

For the client and admin APIs, the preferred authentication mechanisms can be set via the nSessionAttributes class used to create a session as follows:

nSessionAttributes:

```
public void setSASLMechPrefs(nSaslMechanism[] mechPrefs)
```

Here, `nSaslMechanism` is an enum with possible values `PLAIN`, `CRAM_MD5` or `DIGEST_MD5`. The array passed in should be an array of any number of these `nSaslMechanisms` in order of preference. Preferences set here will take precedence over any preferences set via environment variables. If this is unset, Universal Messaging will use the mechanism preference set via the environment variable `Nirvana.sasl.client.mechanism`. If this environment variable is unset, the default mechanism will be `PLAIN`. Note that this method is unavailable to clients using a reactive session. In this case, the mechanism preferences can only be set via the environment variable `Nirvana.sasl.client.mechanism`.

In order to supply credentials to the API, Universal Messaging offers a number of additions to the standard constructors and factory methods. Either the username and password can be supplied independently as a String and a SecureString (inbuilt in .NET in System.Security) respectively or in some cases both can be supplied together inside a NetworkCredentials object (inbuilt in .NET in System.Net).

Thus we have the following API additions:

Client Sessions:

nSessionFactory:

```
public static nSession create(nSessionAttributes sAttr,
    String username, SecureString password)
public static nSession create(nSessionAttributes sAttr,
    NetworkCredentials creds)
public static nSession create(nSessionAttributes sAttr,
    nReconnectHandler handler, String username, SecureString password)
public static nSession create(nSessionAttributes sAttr,
    nReconnectHandler handler, NetworkCredentials creds)
public static nSession createMultiplexed(nSessionAttributes sAttr,
    String username, SecureString password)
public static nSession createMultiplexed(nSessionAttributes sAttr,
    NetworkCredentials creds)
public static nSession createMultiplexed(nSession session,
    String username, SecureString password)
public static nSession createMultiplexed(nSession session,
    NetworkCredentials creds)
```

Admin Sessions:

nRealmNode:

```
Constructor -
nRealmNode(nSessionAttributes sAttr, String username, SecureString passwd);
Constructor -
nRealmNode(nSessionAttributes sAttr, NetworkCredentials creds);
```

nRealmAdmin:

```
Constructor -
nRealmAdmin(nSessionAttributes sAttr, String username,
    SecureString password)
Constructor -
nRealmAdmin(nSessionAttributes sAttr, String username,
    SecureString password, bool followTheMaster)
```

Reactive Sessions:

ISessionAttributes:

```
string Username { get; set; }
SecureString Password { get; set; }
NetworkCredentials Credentials { get; set; }
```

Server-side Authentication

For information about setting up and using server-side authentication, see the section "Server-side Authentication" in the *Concepts* guide. The information described there applies equally to basic authentication when using the C# client.

Google Protocol Buffers

Overview

Google Protocol Buffers are a way of efficiently serializing structured data. They are language and platform neutral and have been designed to be easily extensible. The structure of your data is defined once, and then specific serialization and deserialization code is produced specifically to handle your data format efficiently.

For general details on using Google protocol buffers, see the section *Google Protocol Buffers* in the *Concepts* guide.

Using Google Protocol Buffers with Universal Messaging

Google supplies libraries for Protocol Buffer in Java, C++ and Python, and third party libraries provide support for many other languages including .NET, Perl, PHP etc. Universal Messaging's client APIs provide support for the construction of Google Protocol Buffer event through which the serialized messages can be passed.

To create a nProtobuf event, simply build your protocol buffer as normal and pass it into the nProtobuf constructor along with the message type used.

```
nProtobufEvent evt = new nProtobufEvent(buffer,"example");
myChannel.publish(evt);
```

nProtobuf events are received by subscribers in the normal way.

The Enterprise Manager can be used to view, edit and republish protocol buffer events, even if the Enterprise Manager is not running on the same machine as the server. The Enterprise Manager is able to parse the protocol buffer message and display the contents, rather than the binary data.

All descriptors will be automatically synced across the cluster if the channel is cluster-wide.

Examples

Publish / Subscribe using Channel Topics

Publish / Subscribe

Publish / Subscribe is one of several messaging paradigms available in Universal Messaging. Universal Messaging Channels are a logical rendezvous point for publishers (producers) and subscribers (consumers) or data (events).

Universal Messaging Data Streams and Data Groups provide an alternative style of Publish/Subscribe where user subscriptions can be managed remotely on behalf of clients.

Universal Messaging Channels equate to Topics if you are using the Universal Messaging Provider for JMS.

Under the publish / subscribe paradigm, each event is delivered to each subscriber once and only once per subscription, and is not typically removed from the channel as a result of the message being consumed by an individual client.

This section demonstrates how Universal Messaging pub / sub works in C#, and provides example code snippets for all relevant concepts:

Channel Publisher

This example publishes events onto a Universal Messaging Channel.

Usage

```
publish <rname> <channel name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
```

Application Source Code

See the online documentation for a code example.

Transactional Channel Publisher

This example publishes events transactionally to a Universal Messaging Channel. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been committed successfully.

Usage

```
txpublish <rname>  <channel name> [count] [size] [tx size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[tx size] - The number of events per transaction (default: 1)
```

Application Source Code

See the online documentation for a code example.

Asynchronous Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel. See also: " Synchronous Subscription" on page 183

Usage

```
subscriber <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

Synchronous Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel. See also: “ Asynchronous Subscription” on page 182

Usage

```
channeliterator <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

C# Client: Asynchronous Durable Channel Consumer

This example shows how to asynchronously consumer events on a Universal Messaging Channel using a durable subscription.

Usage

```
namedsubscriber <rname> <channel name> [name] [start eid] [debug] [count] [auto ack]
                                   [cluster wide] [persistent] [selector] [shared]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
```

```
[name]          - Specifies the unique name to be used for a durable subscription
                  (default: OS username)
[start eid]     - The Event ID to start subscribing from if the durable subscription
                  needs to be created (doesn't exist)
[debug]         - The level of output from each event,
                  0 - none, 1 - summary, 2 - EIDs, 3 - All
[count]        - The number of events to wait before printing out summary information
                  (default: 1000)
[auto ack]      - Specifies whether each event will be automatically acknowledged by
                  the api (default: true)
[cluster wide] - Specifies whether the durable subscription is to be used across a
cluster
                  (default: false)
[persistent  ] - Specifies whether the durable subscription state is to be stored to
 disk or
                  held in server memory (default: false)
[selector]      - The event filter string to use
[shared]        - Whether the durable subscriber is shared (default: false)
```

Application Source Code

See the online documentation for a code example.

C# Client: Synchronous Durable Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging channel using a durable subscription and a channel iterator.

Usage

```
namedchanneliterator <rname> <channel name> [name] [start eid] [debug] [count]
                                        [cluster wide] [persistent] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name]          - Specifies the unique name to be used for a durable subscription
                  (default: OS username)
[start eid]     - The Event ID to start subscribing from if name subscriber is to be
                  created (doesn't already exist)
[debug]         - The level of output from each event,
                  0 - none, 1 - summary, 2 - EIDs, 3 - All
[count]         - The number of events to wait for before printing out summary
                  information (default: 1000)
[cluster wide] - Specifies whether the durable subscription is to be used across a
cluster
                 (default: false)
[persistent]    - Specifies whether the durable subscription state is to be stored
                  to disk or held in server memory (default: false)
[selector]      - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

Batching Server Calls

This example shows how to find multiple channels and queues in one call to the server.

Usage

```
findChannelsAndQueues <RNAME> <name> <name> <name>.....
<Required Arguments>
<RNAME> - The RNAME of the realm you wish to connect to
<name> - The name(s) of the channels to find
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Batching Subscribe Calls

This example of batching shows how to subscribe to multiple Universal Messaging Channels in one server call.

Usage

```
sessionsubscriber <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Folder name parameter for the location of the channels to subscribe
 to
[Optional Arguments]
[start eid] - The Event ID to start subscribing from
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
[selector] - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

Publish / Subscribe using Data Streams and Data Groups

DataStream Listener

This example shows how to initialise a session with a DataStream listener and start receiving data.

Usage

```
DataGroupListener <rname> [debug] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
```

```
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

DataGroup Publishing with Conflation

This example shows how to publish to DataGroups, with optional conflation.

Usage

```
DataGroupPublish <rname> <group name> <conflate> [count] [size]
                [conflation merge or drop] [conflation interval]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<group name> - Data group name parameter to publish to
<conflate> - enable conflation true or false
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[conflation merge or drop] - merge to enable merge or drop to enable drop
                            (default: merge)
[conflation interval] - the interval for conflation to publish(default: 500
```

Application Source Code

See the online documentation for a code example.

DataGroup Manager

This is an example of how to run a DataGroup manager application

Usage

```
dataGroupsManager <rname> <Properties File Location>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<Properties File Location Data Groups> - The location of the property file to use for
    mapping data groups to data groups
<Properties File Location Data Streams> - The location of the property file to use
for
    mapping data streams to data groups
<Auto Recreate Data Groups> - True or False to auto recreate data groups takes the
    data group property file and creates channels
    a group for every name mentioned on the left of equals sign
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Delete DataGroup

This is a simple example of how to delete a DataGroup

Usage

```
deleteDataGroups <RNAME> <data group name> [delete type]
<Required Arguments>
<RNAME> - RNAME for the realm to connect to
<data group name> - Data group name parameter to delete
<Optional Arguments>
[Delete Type] - Data group delete by string(1) or object(2) default:1
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

DataGroup Delta Delivery

This example shows how to use delta delivery with DataGroups.

Usage

```
DataGroupDeltaDelivery <rname> [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<Optional Arguments>
<count> - the number of times to commit the registered events - default : 10
```

Application Source Code

See the online documentation for a code example.

Message Queues

Queue Publisher

This example publishes events onto a Universal Messaging Queue.

Usage

```
pushq <rname> <queue name> [count] [size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
```

```
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
```

Application Source Code

See the online documentation for a code example.

Transactional Queue Publisher

This example publishes events transactionally to a Universal Messaging Queue. A Universal Messaging transaction can contain one or more events. The events which make up the transaction are only made available by the Universal Messaging server if the entire transaction has been committed successfully.

Usage

```
txpushq <rname> <queue name> [count] [size] [tx size]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name parameter for the queue to publish to
[Optional Arguments]
[count] -The number of events to publish (default: 10)
[size] - The size (bytes) of the event to publish (default: 100)
[tx size] - The number of events per transaction (default: 1)
```

Application Source Code

See the online documentation for a code example.

Asynchronous Queue Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Queue. See also: " Synchronous Queue Subscription" on page 189

Usage

```
qsubscriber <rname> <queue name> [debug] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name parameter for the queue to pop from
[Optional Arguments]
[debug] - The level of output from each event,
          0 - none, 1 - summary, 2 - EIDs, 3 - All
[transactional] - true / false whether the subscriber is transactional, if true,
                  each event consumed will be ack'd to confirm receipt
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

Synchronous Queue Consumer

This example shows how to synchronously consume events from a Universal Messaging Queue.
See also: " Asynchronous Queue Subscription" on page 188

Usage

```
qreader <rname> <queue name> [debug] [timeout] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name to pop from
[Optional Arguments]
[debug] - The level of output from each event,
          0 - none, 1 - summary, 2 - EIDs, 3 - All
[timeout] - The timeout for the synchronous pop call
[transactional] - true / false whether the subscriber is transactional,
                  if true, each event consumed will be ack'd to confirm receipt
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

Peek Events on a Queue

Consume events from a Universal Messaging Queue in a non-destructive manner

Usage

```
qpeek <rname> <queue name> [debug] [selecter] [count]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - Queue name on which to peek
[Optional Arguments]
[debug] - The level of output from each event, 0 - none, 1 - summary, 2 - EIDs, 3 -
All
[selector] - The event filter string to use
[count] - The number of events to wait before printing out summary information
```

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana API

MyChannels.Nirvana DataGroup Publisher

This example shows how to create a DataGroup Publisher using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana Queue Publisher

This example shows how to create a Queue Publisher using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana Topic Publisher

This example shows how to create a Topic Subscriber using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana DataGroup Listener

This example shows how to create a DataGroup Listener using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana Queue Consumer

This example shows how to create a Queue Consumer using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

MyChannels.Nirvana Topic Subscriber

This example shows how to create a Topic Subscriber using the MyChannels.Nirvana API.

Application Source Code

See the online documentation for a code example.

RX Topic Subscriber

This example shows how to create a Topic Subscriber using the Universal Messaging Reactive library.

Application Source Code

See the online documentation for a code example.

RX Queue Consumer

This example shows how to create a Queue Consumer using the Universal Messaging Reactive library.

Application Source Code

See the online documentation for a code example.

RX DataGroup Listener

This example shows how to create a DataGroup Listener using the Universal Messaging Reactive library.

Application Source Code

See the online documentation for a code example.

Administration API

Add a Queue ACL Entry

This example demonstrates how to add an ACL entry to a Universal Messaging Queue.

Usage

```
naddqueueacl <rname> <user> <host> <queue name> [list_acl] [modify_acl] [full] [peek]
                                                [push] [purge] [pop]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter for the queue to add the ACL entry to
<host> - Host name parameter for the queue to add the ACL entry to
<queue name> - Queue name parameter for the queue to add the ACL entry to
[Optional Arguments]
[list_acl] - Specifies that the list acl permission should be added
[modify_acl] - Specifies that the modify acl permission should be added
[full] - Specifies that the full permission should be added
[peek] - Specifies that the peak permission should be added
[push] - Specifies that the push permission should be added
[purge] - Specifies that the purge permission should be added
[pop] - Specifies that the pop permission should be added
```

Application Source Code

See the online documentation for a code example.

Modify a Channel ACL Entry

This example demonstrates how to modify the permissions of an ACL entry on a Universal Messaging Channel.

Usage

```
nchangechanacl <rname> <user> <host> <channel name> [+/-list_acl] [+/-modify_acl]
 [+/-full] [+/-last_eid] [+/-read] [+/-write] [+/-purge] [+/-named] [+/-all_perms]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter for the channel to change the ACL entry for
<host> - Host name parameter for the channel to change the ACL entry for
<channel name> - Channel name parameter for the channel to change the ACL entry for
[Optional Arguments]
[+/-] - Prepending + or - specifies whether to add or remove a permission
[list_acl] - Specifies that the list acl permission should be added/removed
[modify_acl] - Specifies that the modify acl permission should be added/removed
[full] - Specifies that the full permission should be added/removed
[last_eid] - Specifies that the get last EID permission should be added/removed
[read] - Specifies that the read permission should be added/removed
[write] - Specifies that the write permission should be added/removed
[purge] - Specifies that the purge permission should be added/removed
[named] - Specifies that the used named subscriber permission should be added/removed
[all_perms] - Specifies that all permissions should be added/removed
```

Application Source Code

See the online documentation for a code example.

Delete a Realm ACL Entry

This example demonstrates how to delete an ACL entry from a realm on a Universal Messaging Channel.

Usage

```
ndelrealmacl <rname> <user> <host> [-r]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<user> - User name parameter to delete the realm ACL entry from
<host> - Host name parameter to delete the realm ACL entry from
[Optional Arguments]
[-r] - Specifies whether recursive traversal of the namespace should be done
```

Application Source Code

See the online documentation for a code example.

Monitor realms for client connections coming and going

This example demonstrates how to monitor for connections to the realm and its channels.

Usage

```
nconnectionwatch <rname>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
```

Application Source Code

See the online documentation for a code example.

Export a realm to XML

This example demonstrates how to export a realm's cluster, joins, security, channels / queues, scheduling, interfaces / plugins and configuration information to an XML file so that it can be imported into any other realm.

Usage

```
nexportrealmxml <rname> <export_file_location>
<Optional Arguments> -all -realms -cluster -realmacl -realmcfg -channels -channeacls
                     -joins -queues -queueacls -interfaces -plugins -via
```

Application Source Code

See the online documentation for a code example.

Import a realm's configuration information

This example demonstrates how to import a realm's cluster, joins, security, channels / queues, scheduling, interfaces / plugins and configuration information from an XML file.

Usage

```
nimportrealmxml <rname> <file_name>
<Optional Arguments> -all -realmacl -realmcfg -channels -channeacls -queues
                     -queueacls -interfaces
```

Application Source Code

See the online documentation for a code example.

Console-based Realm Monitor

This example demonstrates how to monitor live realm status.

Usage

```
nTop <rname> [refreshRate]
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
[Optional Arguments]
[refreshRate] - the rate at which the information is reloaded on screen (milliseconds)
```

Application Source Code

See the online documentation for a code example.

Set Container ACL

Set the ACL of a container to that currently applied to a specified channel.

Usage

```
nsetcontaineracl <channel name> <container name>
<Required Arguments>
<rname> - name of the realm to connect to.
<channel name> - channel name parameter used to obtaine the ACL to set the container
                 nodes to" );
<container name> - Container name parameter for the container to set the ACL to" );
Note: -? provides help on environment variables
```

Application Source Code

```
/**
 *
 *  --------------------------------------------------------------------------------
 *
 *  PCB Systems Limited License Version 1.1
 *  Copyright  PCB Systems Limited. All rights reserved
 *
 *  In the event that you should download or otherwise use this software
 *  ( the "Software" ) you hereby acknowledge and agree that:
 *
 *  1. The Software is the property of PCB Systems Limited: Title, Copyright and all
 *  other proprietary rights, interest and benefit in and to the Software is and
 *  shall be owned by PCB Systems Limited;
 *
 *  2. You will not make copies of the Software whatsoever other than, if you should
 *  so wish, a single copy for archival purposes only;
 *
 *  3. You will not modify, reverse assemble, decompile, reverse engineer or otherwise
 *  translate the Software;
 *
 *  4. You will not redistribute, copy, forward electronically or circulate the Software
 *  to any person for any purpose whatsoever without the prior written consent of
 *  PCB Systems Limited;
 *
 *  5. You will not charge for, market or provide any managed service or product that
 *  is based upon or includes the Software or any variant of it; and
 *
 *  6. You will not use the Software for any purpose apart from your own personal,
```

```
 *  noncommercial and lawful use;
 *
 *  You hereby agree that the software is used by you on an "as is" basis, without
 *  warranty of any kind. PCB Systems Limited hereby expressly disclaim all warranties
 *  and conditions, either expressed or implied, including but not limited to any
 *  implied warranties or conditions or merchantability and fitness for a particular
 *  purpose.
 *
 *  You agree that you are solely responsible for determining the appropriateness of
 *  using the Software and assume all risks associated with it including but not
 *  limited to the risks of program errors, damage to or loss of of data, programs or
 *  equipment and unavailability or interruption of operations.
 *
 *  PCB Systems Limited will not be liable for any direct damages or for any, special,
 *  incidental or indirect damages or for any economic consequential damages ( including
 *  lost profits or savings ), or any damage howsoever arising.
 */
namespace com.pcbsys.nirvana.nAdminAPI
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading;
    using System.Collections;
    using com.pcbsys.nirvana.client;
    class setContainerACL
    {
   /**
    * Private variables used in this application
    */
   private String name = null;
   private String host = null;
   private nSessionAttributes attr = null;
   private String containerName = null;
   private String channelName = null;
   private nRealmNode node = null;
   private nLeafNode leaf = null;
   private nACL acl = null;
   private String rname = null;

   /**
    * Consruct and instance of this class using the command line arguments passed
    * when it is executed.
    */
   public setContainerACL(String[] args) {

     //Process command line arguments
     processArgs(args);

     try {
       Console.WriteLine( "Connecting to " + rname );
       // construct the session attributes from the realm
       attr = new nSessionAttributes( rname );
       // get the root realm node from the realm admin
       node = new nRealmNode(attr);
       if(!node.isAuthorised()){
         Console.WriteLine("User not authorised on this node "+attr);
         return;
       }
```

```
      // wait for the entire node namespace to be constructed
      Console.WriteLine( "waiting for namepsace construction..... " );
      node.waitForEntireNameSpace();
      Console.WriteLine( "finished" );
      leaf = (nLeafNode)node.findNode(channelName);
      if (leaf != null) {
       acl = leaf.getACLs();
        searchNode(node);
      } else {
       Console.WriteLine("Cannot find leaf node "+channelName);
      }
      node.close();
    } catch (Exception e) {
      Console.WriteLine(e.StackTrace);
    }
  }

  /**
   * recursively search through the realm node looking for channel nodes
   */
  public void setContainer(nContainer p_node) {
    try {
      // set the acl for the container nodes
      Console.WriteLine( "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" );
     Console.WriteLine( "Applying acl to container node " + p_node.getAbsolutePath()
);
      // set the acl on the container
        p_node.setACL(acl);
      Console.WriteLine( "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" );
    } catch (Exception e) {
      Console.WriteLine(e.StackTrace);
    }
  }

  /**
   * search the enumeration of child nodes for other realms and containers
   */
  private void searchNodes( nContainer p_node, System.Collections.IEnumerator enum1
) {
    while ( enum1.MoveNext() ) {
      Object obj = enum1.Current;
     if ( obj is nRealmNode ) {
      searchNode( (nRealmNode)obj );
     } else if ( obj is nContainer ) {
        nContainer cont = (nContainer)obj;
        String fullyQualifiedName = cont.getAbsolutePath();
        if (fullyQualifiedName.Equals(containerName)) {
          Console.WriteLine("Found container "+fullyQualifiedName);
          setContainer( cont );
       } else {
         searchNodes(cont, cont.getNodes());
       }
      }
    }
  }

  /**
   * Search the children of the realm passed as a paremeter
   */
```

```
  private void searchNode( nRealmNode p_node ) {
    try {
      searchNodes( p_node, p_node.getNodes() );
    }
    catch ( Exception ex ) {
      Console.WriteLine(ex.StackTrace);
    }
  }

  /**
   * If you construct an instance of this class from another class, you can set the
name
   * and host for the subject.
   */
  public void setSubject(String p_name, String p_host) {
    name = p_name;
    host = p_host;
  }

  /**
   * Set the program variables and permissions flags based on command line args
   */

  private void processArgs(String[] args){
    if (args.Length != 3) {
      Usage();
      Environment.Exit(1);
    }
    switch (args.Length){
      case 3:
            channelName = args[2];
            goto case 2;
      case 2:
            containerName = args[1];
            goto case 1;
      case 1:
            rname = args[0];
        break;
    }
  }

  /**
   * Run this as a command line program passing the command line args.
   *
   * Or construct one of these classes from another class ensuring you have added :
   *
   *    RNAME
   *    CHANNEL
   *    CONTAINER
   *
   * as system properties, and pass in a list of permissions in the constructor
   *
   */
  public static void Main( String[] args ) {

    setContainerACL setAcl = new setContainerACL(args);
    Environment.Exit(0);
  }

  /**
```

```
     * Prints the usage message for this class
     */
    private static void Usage() {

      Console.WriteLine( "Usage ...\n" );
      Console.WriteLine("nsetcontaineracl <channel name> <container name> \n");
      Console.WriteLine(
        "<Required Arguments> \n");
      Console.WriteLine(
        "<rname> - name of the realm to connect to.");
      Console.WriteLine("<channel name> - channel name parameter used to obtain " );
      Console.WriteLine("    the ACL to set the container nodes to" );
      Console.WriteLine("<container name> - Container name parameter for the " );
      Console.WriteLine("    container to set the ACL to" );
      Console.WriteLine(
        "\n\nNote: -? provides help on environment variables \n");
    }

    private static void UsageEnv() {
      Console.WriteLine(
        "\n\n(Environment Variables) \n");

      Console.WriteLine(
        "(RNAME) - One or more RNAME entries in the form protocol://host:port" );
      Console.WriteLine(
        "   protocol - Can be one of nsp, nhp, nsps, or nhps, where:" );
      Console.WriteLine(
        "   nsp - Specifies Universal Messaging Socket Protocol (nsp)" );
      Console.WriteLine(
        "   nhp - Specifies Universal Messaging HTTP Protocol (nhp)" );
      Console.WriteLine("   nsps - Specifies Universal Messaging Socket Protocol " );
      Console.WriteLine("          Secure (nsps), i.e. using SSL/TLS" );
      Console.WriteLine("   nhps - Specifies Universal Messaging HTTP Protocol " );
      Console.WriteLine("          Secure (nhps), i.e. using SSL/TLS" );
      Console.WriteLine(
        "   port - The port number of the server" );
      Console.WriteLine("\nHint: - For multiple RNAME entries, use comma separated "
);
      Console.WriteLine("          values which will be attempted in connection " );
      Console.WriteLine("          weight order\n" );

      Console.WriteLine("(LOGLEVEL) - This determines how much information the nirvana
 " );
      Console.WriteLine("             api will output 0 = verbose 7 = quiet\n" );

      Console.WriteLine("(CKEYSTORE) - If using SSL, the location of the keystore ");
      Console.WriteLine("              containing the client cert\n");
      Console.WriteLine("(CKEYSTOREPASSWD) - If using SSL, the password for the");
      Console.WriteLine("                    keystore containing the client cert\n");
      Console.WriteLine(
        "(CAKEYSTORE) - If using SSL, the location of the ca truststore\n");
      Console.WriteLine(
        "(CAKEYSTOREPASSWD) - If using SSL, the password for the ca truststore\n");

      Console.WriteLine(
        "(HPROXY) - HTTP Proxy details in the form proxyhost:proxyport, where:" );
      Console.WriteLine(
        "   proxyhost - The HTTP proxy host" );
      Console.WriteLine(
        "   proxyport - The HTTP proxy port\n" );
```

```
      Console.WriteLine(
        "(HAUTH) - HTTP Proxy authentication details in the form user:pass, where:" );
      Console.WriteLine(
        "   user - The HTTP proxy authentication username" );
      Console.WriteLine(
        "   pass - The HTTP proxy authentication password\n" );
      Environment.Exit(1);
     }
  }
}
```

Channel / Queue / Realm Management

Creating a Channel

Output all the differences between two realms.

Usage

```
makechan <rname> <channel name> [time to live] [capacity] [type] [cluster wide]
                                [start eid]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to be created
[Optional Arguments]
[time to live] - The Time To Live parameter for the new channel (default: 0)
[capacity] - The Capacity parameter for the new channel (default: 0)
[type] - The type parameter for the new channel (default: S)
R - For a reliable (stored in memory) channel with persistent eids
P - For a persistent (stored on disk) channel
M - For a Mixed (allows both memory and persistent events) channel
[cluster wide] - Whether the channel is cluster wide. Will only work if the realm
                is part of a cluster (default: false)
[start eid] - The initial start event id for the new channel (default: 0)
```

Application Source Code

See the online documentation for a code example.

Deleting a Channel

Output all the differences between two realms.

Usage

```
deletechan <rname> <channel name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<channel name> - Channel name parameter for the channel to delete
```

Application Source Code

See the online documentation for a code example.

Creating a Queue

This example demonstrates how to create a Universal Messaging queue programmatically.

Usage

```
makequeue <rname> <queue name> [time to live] [capacity] [type] [cluster wide] [start
 eid]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - queue name parameter for the queue to be created
[Optional Arguments]
[time to live] - The Time To Live parameter for the new queue (default: 0)
[capacity] - The Capacity parameter for the new queue (default: 0)
[type] - The type parameter for the new queue (default: S)
R - For a reliable (stored in memory) queue with persistent eids
P - For a persistent (stored on disk) queue
M - For a Mixed (allows both memory and persistent events) queue
[cluster wide] - Whether the queue is cluster wide. Will only work if the realm is
part
                 of a cluster (default: false)
[start eid] - The initial start event id for the new queue (default: 0)
```

Application Source Code

See the online documentation for a code example.

Deleting a Queue

This example demonstrates how to delete a Universal Messaging queue programmatically.

Usage

```
deletequeue <rname> <queue name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<queue name> - queue name parameter for the queue to delete
```

Application Source Code

See the online documentation for a code example.

Create Channel Join

Create a join between two Universal Messaging Channels.

Usage

```
makechanneljoin <rname> <source channel name> <destination channel name> [max hops]
                                                           [selector] [allow purge]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination channel name> - Channel name parameter of the remote channel name to join
[Optional Arguments]
[max hops] - The maximum number of join hops a message can travel through
[selector] - The event filter string to use on messages travelling through this join
[allow purge] - boolean to specify whether purging is allowed (default : true)
```

Application Source Code

See the online documentation for a code example.

Delete a Channel Join

Create a join between two Universal Messaging Channels

Usage

```
deletechanneljoin <rname> <source channel name> <destination channel name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination channel name> - Channel name parameter of the remote channel name to join
```

Application Source Code

See the online documentation for a code example.

Purge Events From a Channel

Delete all events from a Universal Messaging Channel

Usage

```
purgeevents <rname> <channel name> <start eid> <end eid> [filter]
<Required Arguments>
<rname> - The realm to retrieve channels from
<channel name> - Channel name parameter for the channel to be purged
<start eid> - The start eid of the range of events to be purged
<end eid> - The end eid of the range of events to be purged
[Optional Arguments]
[filter] - The filter string to use for the purge
```

Application Source Code

See the online documentation for a code example.

Create Queue Join

Create a join between a Universal Messaging Queue and a Universal Messaging Channel

Usage

```
makequeuejoin <rname> <source channel name> <destination queue name> [max hops]
[selector]
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination queue name> - Queue name parameter of the remote queue name to join
[Optional Arguments]
[max hops] - The maximum number of join hops a message can travel through
[selector] - The event filter string to use on messages travelling through this join
```

Application Source Code

See the online documentation for a code example.

Delete Queue Join

Delete a join between a Universal Messaging Queue and a Universal Messaging Channel

Usage

```
deletequeuejoin <rname> <source channel name> <destination queue name>
<Required Arguments>
<rname> - the custom URL to access the realm. Example: nhp://localhost:9000
<source channel name> - Channel name parameter of the local channel name to join
<destination queue name> - Queue name parameter of the remote queue name to join
```

Application Source Code

See the online documentation for a code example.

Prerequisites

C# Prerequisites

This section gives information on what is required to get started using the C# API for Universal Messaging. The C# API for Universal Messaging is available in a DLL distribution for developing native Microsoft Windows applications ("Nirvana DotNet.dll").

Universal Messaging .NET

Universal Messaging .Net requires Microsoft .Net Framework, which you can download from the Microsoft Download website at http://www.microsoft.com/downloads/. The .NET installer will automatically set up the environment such that C# applications can be compiled and run natively on Microsoft Windows. Please see the Environment Setup section below for information on how to compile and run applications using the Universal Messaging C#.NET API.

SSL

To subscribe to a channel using an SSL interface, extra requirements must be met. Universal Messaging C# supports client certificate authentication as well as anonymous SSL. For client certificate authentication, the location of the client certificate and private key password, as well as the trust store must be known to the application. For instructions on how to run Universal Messaging C# applications using an SSL enabled interface, please see Client SSL.

Environment Setup

Compilation

It is recommended that you use Microsoft Visual Studio to compile Universal Messaging C# applications. Visual Studio will set up the required environment for compiling C# applications. However to make use of the Universal Messaging APIs, the location of the Universal Messaging libraries will need to be referenced such that they can be found by the compiler.

The libraries can be found in the `<InstallDir>`\UniversalMessaging`\dotnet\bin` directory. For native Windows applications the "Nirvana DotNet.dll" library is required.

Runtime

The Universal Messaging DLLs used to compile C# applications are unlike C++ in that these libraries are used both at compile time and at runtime. At compile time, the location of the library is specified as a reference such that it can be used by the compiler. At runtime this library is looked for in the same directory as the executable. For information on how to run an application without the DLL in the same directory, see Globally Accessible DLLs.

Sample Applications

The `<InstallDir>`\UniversalMessaging`\dotnet\bin` directory contains precompiled sample applications for Universal Messaging C#.Net. These applications can be run on a PC running Microsoft Windows which has .NET installed as described above.

The source code for each application can be found in `<InstallDir>`\UniversalMessaging`\dotnet\examples` along with a batch file which can be used to compile the application:

```
> cd <InstallDir>\UniversalMessaging\dotnet\examples\channeliterator
> builddotnetsampleapp.bat channeliterator
```

This will compile the channeliterator sample application and place the executable in the `dotnet\bin` directory.

C# Client SSL Configuration

Universal Messaging fully supports SSL Encryption. This section describes how to use SSL in your Universal Messaging C# client applications.

Once you have created an SSL enabled interface you will need to create certificates for the server and client (if using client certificate authentication). The Universal Messaging download contains a generator to create some example Java key store files to be used by the Universal Messaging server but may also be converted to Public-Key Cryptography Standards (PKCS) files for use with a Universal Messaging C# client. To convert from .jks to .p12 you can use keytool.exe (supplied with java). The command to do so is shown below:

```
keytool -importkeystore -srckeystore client.jks -destkeystore client.p12
        -srcstoretype JKS -deststoretype PKCS12
```

Please refer to this guide to create your own client certificates. However please remember that in order to run a Universal Messaging C# client, the certificate provided must be in PKCS format.

Running a Universal Messaging C# Client

A client can use anonymous SSL, but when the Universal Messaging SSL interface is configured for client validation, only trusted clients can connect with a valid certificate. To enable or disable client certificate validation at the realm server, you can use the Universal Messaging Enterprise Manager. Highlight the SSL enabled interface in the "Interface" tab for your realm then open the "Certificates" tab and check or uncheck the box labelled "Enable Client Cert Validation". Hit the Apply button, and restart the interface.

When client certificate validation is enabled, the client is required to have a certificate so that the server can validate the client. If the server certificate is self signed (as the certificates created using the generator are), the client must also have a trust store to validate the server certificate.

The location of the key stores and the relevant passwords need to be specified in nConstants. This can be done by adding the client certificate and trust store to the windows certificate store. The location of the client certificate can also be set by setting the certificate property (defined in nConstants) in the application code or by setting CERTPATH (the location of the certificate) and CERTPASS (the private key password) as environment variables. For more information, see SSL Concepts.

Adding Certificates to the Windows Certificate Store

if you wish to add certificates to the Windows certificate store, follow these instructions.

The default password for the certificates created using the generator is "nirvana".

To add the client certificate:

- Open the Start menu, click on Run and enter "certmgr.msc".
- In the new window, expand the "Personal" folder and right click on the "Certificates" folder.
- Select "All Tasks->Import..."
- Follow the Instructions and import the client certificate (client.p12)

To add the trust store:

- Open the Start menu, click on Run and enter "certmgr.msc".
- In the new window, expand the "Trusted Root Certification Authorities" folder and right click on the "Certificates" folder.
- Select "All Tasks->Import..."
- Follow the Instructions and import the trust store (nirvanacacerts.p12)

You will now be able to connect to a realm using nsps and nhps.

Globally Accessible DLLs

By default, C# applications require any user created DLLs to be present in the same directory as the application. As DLLs are typically shared by multiple applications, it may be necessary for the DLL to be placed in a globally accessible location. To do this in C# you need to add the DLL file to the Global Assembly Cache (GAC).

Strong-Named Assemblies

Before a DLL can be added to the GAC, it must be given a strong-same. This procedure aims to protect the user from corrupted DLLs. As DLLs are linked at runtime, it would be possible for someone to build a new version of the DLL but add malicious code. The user application would have no way of telling that this is not the correct DLL and would run the malicious code. GAC and strong-named assemblies protect against this, for more information see Strong-Named Assemblies on the Microsoft website.

Creating a Strong-Named Assembly

The C# DLLs in the Universal Messaging download have already been given strong-names so this section is not required to make the Universal Messaging DLLs globally accessible.

1. Either open a .NET command prompt or open a standard command prompt and run vsvars32.bat which is located in "C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools". Which will set up the required environment.
2. Navigate to a directory where you want to store the keyfile and run the following command:

   ```
   C:\myarea\folder\> sn -k keyfile.snk
   ```

   This will create a keyfile which contains a pair of private and public keys which can be used to protect your DLLs.
3. Now you need to edit the AssemblyInfo.cs file for the project used to create the DLL by adding the following code:

   ```
   [assembly:AssemblyKeyFile(@"C:\myarea\folder\keyfile.snk")]
   ```

4. Now when you build the DLL as usual it will be given a strong-name but will not be globally accessible until added to GAC.

Adding Strong-Named Assembly to GAC

1. Either open a .NET command prompt or open a standard command prompt and run vsvars32.bat which is located in "C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools". Which will set up the required environment.

2. In this prompt execute gacutil as shown below:

```
C:\myarea\folder\> gacutil /i mylib.DLL
```

The DLL will now be globally accessible on the system. The C#.NET sample applications in the download use the "Nirvana Dotnet.DLL" library and "nSampleApp.DLL", both have been given strong-names so can be added to GAC using gacutil as described above.

NOTE: to remove an assembly from the cache execute "gacutil /u mylib", the file extension is not required.

Using UM API libraries for .NET in .NET Standard projects

The Universal Messaging API for .NET is built and tested against .NET Framework 4.5, which implements .NET Standard 1.1. However it is verified that the `Nirvana DotNet.dll`, `Nirvana nAdminAPI.dll` and `Nirvana.Reactive.dll` APIs are compatible with .NET Standard 2.0. This means that they can be used in .NET Standard 2.0+ compatible runtimes, such as .NET Core 2.0. For the verification, the ".NET Portability Analyzer" tool is used (https://docs.microsoft.com/en-us/dotnet/standard/analyzers/portability-analyzer).

Limitations

The SASL authentication available with the Universal Messaging API for .NET cannot be used in a .NET Standard 2 project. SASL is implicitly used when a password is passed on session creation using the `nSessionFactory.create(..., NetworkCredential creds)` API.

An exception such as the following will be thrown on session init in this case:

```
com.pcbsys.nirvana.client.nRealmUnreachableException,
Message=Realm is currently not reachable:
Requested SASL mechanisms not supported by chosen library.
```

Messaging API

MyChannels.Nirvana API: Creating and Disposing of a Session

Creating a session is extremely simple with the C# .NET MyChannels.Nirvana API. Simply create a new Session object with the desired RNAME, then call the Session.Initialize() method.

```
String RNAME = "nsp://127.0.0.1:9000";
Session session = new Session(RNAME);
session.Initialize();
```

DataGroups can be enabled on a session by setting the DataGroups.Enable flag, as shown below, before the call to Initialize() is made.

```
session.DataGroups.Enable = true;
```

To end a session, call the Session.Dispose() method.

```
session.Dispose();
```

Session Events

- AsynchronousExceptionRaised - fired when an asynchronous exception is thrown by the session
- ConnectionStatusChanged - fired when the connection status changes, for example when the connection is lost.

MyChannels.Nirvana API: Producers

The sending of messages is exposed via the Producers feature, simplifying the message sending process across Topics, Queues and DataGroups by using an identical procedure for each.

Firstly, a Producer is created of the appropriate type, passing in the name of the DataGroup, Topic or Queue. Examples are included below for each of the three mechanisms. Obviously, in order to use DataGroups, they must first be enabled by setting the Session.DataGroups.Enable flag to true before initializing the session.

```
IProducer producer = session.DataGroups.CreateProducer("Group1");
```

```
IProducer producer = session.Queues.CreateProducer("Queue1");
```

```
IProducer producer = session.Topics.CreateProducer("Topic1");
```

In order to send a message, a Message object is first created, as shown below, then is passed into the Producer's Send() method. The Message constructor has various overloads to allow the specification of properties, tags and data.

```
// Creating a Message
string msgContents = "Hello World!";
Message msg = new Message(msgContents, new byte[] { }));
producer.Send(msg);
```

MyChannels.Nirvana API: Consumers

Consumers are the main means of consuming messages when using the MyChannels.Nirvana API. They allow simple consumption of messages from both Topics and Queues. A Consumer is created using the CreateConsumer() method in either Session.Queues or Session.Topics, depending upon which type of Consumer is desired.

The Consumer's MessageReceived event is fired whenever a message is received by the Topic or Queue being consumed. By attaching an appropriate handler, the message can be dealt with in whatever way is desired.

```
IConsumer consumer = session.Queues.CreateConsumer("Queue1");
consumer.MessageReceived += (s, e) => ProcessMessage(e.Message);
```

```
IConsumer consumer = session.Topics.CreateConsumer("Topic1");
```

```
consumer.MessageReceived += (s, e) => ProcessMessage(e.Message);
```

DataGroups

Consuming messages when using DataGroups is even simpler than when using Topics or Queues. The Session.DataGroups object itself has a MessageReceived event, which can be used in the same manner as above to handle incoming messages.

```
session.DataGroups += (s, e) => ProcessMessage(e.Message);
```

MyChannels.Nirvana API: Reactive Extensions

Reactive Extensions for .NET (commonly referred to as "Rx") is a new library currently under development by Microsoft that aims to allow the development of so-called "reactive" applications, by exposing the Observer pattern (as seen in C# Multicast delegates and Events), but in a simpler, more intuitive manner.

Nirvana.Reactive

The Universal Messaging Reactive library for .NET aims to make use of the capabilities offered by Rx, by allowing the conversion from Universal Messaging objects to Observable sequences and vice versa.

Currently, the library only supports the conversion from Universal Messaging objects to Observable sequences, and is designed to work with the MyChannels.Nirvana API. One main method is included: ToObservable(), which converts the messages from either a IConsumer (Topics and Queues) or a IDataGroupSession. This means that consuming messages on a Topic or Queue looks distinctly different from the more conventional Consumer method.

```
var consumer = session.Topics.CreateConsumer("Topic1");
var query = from e in consumer.ToObservable()
            select e.Message;
            // Subscribe
query.Subscribe(ProcessMessage);
//...
public void ProcessMessage(object m)
{
    Console.WriteLine("Message: {0}", ((Message)m).Id);
}
```

This looks somewhat confusing at first glance, but is simple enough when broken down. The ToObservable() call on the Topic Consumer returns an Observable sequence of MessageEventArgs, as returned when the MessageReceived event is fired in the MyChannels.Nirvana API on the Consumer. The query simply filters that sequence to obtain the Messages from each MessageEventArgs. The Subscribe() method allows a handling method to be attached to the Observable sequence, just as one would attach an event handler to an typical event. In this case, the ProcessMessage() method simply writes the Id of the message received to the console.

DataGroups work in a similar fashion. As DataGroups do not have Consumers in the manner of Topics and Queues, the ToObservable() method is instead called on the IDataGroupSession object, returning an Observable sequence which can be manipulated in an identical fashion.

```
var query = from e in session.DataGroups.ToObservable()
            select e.Message;
            // Subscribe
query.Subscribe(ProcessMessage);
```

Enterprise Developer's Guide for VBA

This guide describes how to develop Microsoft Excel spreadsheets which receive data in real time and publish events to Universal Messaging Channels using Visual Basic for Applications (VBA).

Universal Messaging Enterprise Client Development in VBA

- "Universal Messaging Publish/Subscribe" on page 209
- "Prerequisites" on page 216

Publish / Subscribe

Publish/Subscribe

The Universal Messaging VBA API allows you to publish and subscribe to Universal Messaging channels using Microsoft Officeproducts such as Excel. Channels are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Subscribing Tasks

Subscribing to a Channel

Once you have installed the Universal Messaging (Nirvana) RTD server, the server will be available for use in any Microsoft Excel spreadsheet on the system. To start subscribing you need to use the RTD function in Excel. The RTD function is used in the same way as any other Excel function. By entering the function with the correct parameters into a cell, you will immediately subscribe to the specified channel and receive the value associated with the specified property contained in the event.

RTD Function

The RTD function is a built in Excel function but the parameters are specific to the Universal Messaging RTD server. To subscribe to a Universal Messaging channel you need to use the following structure:

```
=RTD("NirvanaRTD",,RNAME,Channel,Property,Key,Value,Key2,Value2 ...)
```

The parameters are explained below:

"NirvanaRTD"

This is the CLSID which has been registered for the Universal Messaging RTD server. By specifying this ID, Excel will look up the Universal Messaging RTD server in the Windows Registry.

Second Parameter

The second parameter is left blank because the Universal Messaging RTD server is installed on the local machine. If it were installed remotely, the server would be specified here.

RNAME

The RNAME of the realm which the cell should connect to. You may also specify certain configurations for the session in this field. The RNAME is of the form:

```
protocol://host:port?property=value&property2=value2...
```

The following properties are available:

- `user` - this is the username that will be used to connect to the realm

Channel

The name of the Universal Messaging Channel which you wish to connect to. You may also specify channel specific configuration properties in this field. The Channel field has the form:

```
/folder/channelname?property=value&property2=value2...
```

The following properties are available:

- `eid` - the eid for which to start subscribing. This value is -1 by default which means subscription starts from the last eid of the channel (will not receive any events currently on the channel). -2 will mean the last event published on the channel is consumed as well as any further events published and hence -3 will mean the last 2 are consumed etc. A positive value will cause mean events from that eid onward are consumed so 0 means all events on the channel will be consumed.
- `hwmark` - the high water mark for the event queue of the channel. This ensures that the event queues do not grow too large without dropping any events. For more information see "Queue watermarks" on page 215.
- `lwmark` - once the event queue has reached high watermark, no more events will be added to the event queue. Once the queue length reaches lwmark (low watermark) the listener is notified to continue receiving events.

Key, Value

The Universal Messaging RTD server allows you to filter events based on key-value pairs. Here the value of Property is only shown if the event properties contains each key and the value associated with that key.

A Universal Messaging Event can contain nEventProperties which themselves can contain nested nEventProperties. These nested properties are accessed by a key in the same way as the values are accessed. In order to access the key-value pairs contained within the inner properties using the RTD server, you should use the syntax shown below:

```
...,propsA.Key,value,propsA.propsB.key,value,...
```

Here propsA is found inside the main nEventProperties for the nConsumeEvent. Inside propsA is a set of key-value pairs but also another nEventProperties object called propsB which itself contains key-value pairs and possibly further nEventProperties.

Publishing Tasks

Creating a Session

To interact with a Universal Messaging Server, the first thing to do is create a Universal Messaging Session object, which is effectively your logical and physical connection to a Universal Messaging Realm.

Creating a Universal Messaging Session Object

The VBA code snippet below demonstrates the creation and initialisation of an nSession object:

```
Dim nsa As New nSessionAttributes
Call nsa.init("nsp://127.0.0.1:9000")
Dim fact As New nSessionFactory
Set sess = fact.Create(nsa)
Call sess.init
```

Finding a Channel

Once the session has been established with the Universal Messaging realm server, the session object can be used to locate an existing Universal Messaging Channel by specifying the channel's name.

Note that you can use the Enterprise Manager GUI to create a Universal Messaging Channel.

This VBA code snippet demonstrates how to find a channel (for example */eur/rates*):

```
Dim nca As New nChannelAttributes
Call nca.setName("/eur/rates")
Set chan = sess.findChannel(nca)
```

Universal Messaging Events

A Universal Messaging Event (*nConsumeEvent*) is the object that is published to a Universal Messaging channel or queue. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an Universal Messaging Event Dictionary (*nEventProperties*).

Constructing an Event

In this VBA code snippet, we construct our Universal Messaging Event object, as well as a Universal Messaging Event Dictionary object (*nEventProperties*) for our Universal Messaging Event:

```
Dim props As New nEventProperties
Call props.put("examplekey", "hello world")
Dim evt As New nConsumeEvent
Call evt.init_2(props)
```

Here the function evt.init_2() is used. The nConsumeEvent class currently has 3 initialise methods but Excel does not support overloading so renames these methods to init_1 init_2 etc.

Publishing Events to a Channel

Once the session has been established with the Universal Messaging realm server, and the channel has been located, the channel's publish function can be invoked.

```
Call chan.publish(evt)
```

Learn More

Event Properties

A Universal Messaging Event (*nConsumeEvent*) can contain nEventProperties. This object contains key-value pairs in a similar way to a hash table and can also support nested nEventProperties.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's properties.

In this code snippet, we assume we want to publish an event containing a key called "myKey" with value "myValue"

```
Dim props As New nEventProperties
Call props.put("myKey", "myValue")

Dim evt As New nConsumeEvent
Call evt.init_2(props)

Call myChannel.Publish(evt)
```

The highlighted code shows the creation of the event properties.

Now say we want to add another set of properties within the properties we have just created. The code below highlight the extra code required to add a nested nEventProperties.

```
Dim props As New nEventProperties
Call props.put("myKey", "myValue")
Dim innerProps As New nEventProperties
Call innerProps.put("myInnerKey", "myInnerValue")
Call props.put_4("myDictName", innerProps)

Dim evt As New nConsumeEvent
```

```
Call evt.init_2(props)

Call myChannel.Publish(evt)
```

Here you see that the inner nEventProperties is created in exactly the same way and is then added to the outer nEventProperties in the same way that you would add a key-value pair with the value being the nEventProperties.

How the RTD Server Works

Excel is a single threaded application which means that asynchronous behavior is limited. Most asynchronous systems make use of either *push* or *pull* methods of receiving data.

Both of these methods have limitations. Pushing data to Excel when Excel is busy (see note* below) will mean that any events pushed will be dropped as Excel cannot deal with them. If Excel is required to *pull* from the server, then because it does not know when the data is available it will have to continually send requests to the server.

For this reason Excel uses a hybrid of both mechanisms. Once events are received, the Universal Messaging RTD server will send a notification to Excel to say that data is available. Excel will then respond to this notification by requesting the RTD server to send the data. This does however mean that if Excel is busy, although no events will be dropped, the notification sent to Excel may be ignored. The Universal Messaging RTD Server deals with this by queueing events internally.

*Excel is said to be busy whenever it is recalculating but also when the user responds to dialog prompts or enters data into a cell.

Setting the RTD Throttle Interval

Excel Throttle Interval

When Microsoft Excel receives a notification that new data is available it will only respond if it is not busy* and if the throttle interval has passed. By default Excel sets a throttle interval of 2 seconds which means that updates cannot be received faster than every 2 seconds. A high throttle value does not mean that events will be missed. The Universal Messaging RTD server queues events and will process the entire queue internally before returning data to Excel.

*Excel is said to be busy whenever it is recalculating but also when the user responds to dialog prompts or enters data into a cell.

Changing the Excel Throttle Interval

The throttle interval is stored in the Windows registry but you may wish to set a different throttle interval for different spreadsheets. In order to do this you need to use VBA.

- Open Excel and switch to the VBA window
- In the Project Explorer panel double click on "ThisWorkbook"
- This will bring up a new code window. In this window enter the following code

```
Private Sub Workbook_Open()
Application.RTD.ThrottleInterval = 0
End Sub
```

By setting a throttle interval of 0, Excel will try to respond to update notifications whenever it is not busy. A value of -1 will set the RTD server to manual mode which means Excel will not respond to any update notifications. Instead the user must manually call the RTD server to request new data.

Internal Event Processing

Microsoft Excel is a single threaded application, therefore it cannot process events when it is in a busy state. Every time an event is received by the RTD server, a notification that new data is available is sent to Excel. As soon as Excel receives this notification it will request data from the RTD server. During this request, Excel enters a busy state and will therefore drop any further notifications that more data is available. If Excel responded to every notification it would be appropriate to simply allow excel to pop an event off the internal event queue and return this data (then deal with the next request) but as this is not the case, a different solution needs to be approached.

The Universal Messaging RTD server approaches this scenario in two different ways:

Processing Historical Data

If a user specifies an eid previous to that of the last published event (0 or less than -1) it is assumed that every event up to the last published event is required by Excel. In this case, the Universal Messaging RTD server will continue to notify Excel that new data is available until the internal event queue is empty and the last published event on the channel has been consumed. Every time Excel requests data it will pop one event off the internal event queue for that channel and update its cells. This ensures that every value is returned to Excel however quickly the events are received.

Once the last published event has been consumed, the RTD server returns to its normal state as described below.

Normal Processing State

Every time Excel requests data, the entire internal event queue is consumed internally and the most recent value required for each cell is returned to Excel.

Every event is processed internally, however only the most recent value that a cell requires is returned. For example if a cell is subscribed to a channel and requests events with property "name". If 50 events are queued internally, each event will be processed but only the most recent value of name would be returned to Excel. This saves Excel from making 50 separate requests for data when it may be that only one of the 50 events contains the property "name". If all 50 events contained the property "name" then returning the value 50 times would cause the value of the cell to rapidly change which is not generally required for an Excel application.

Universal Messaging RTD Server Internal Queues

High/Low Watermark

As mentioned in "How RTD Works" on page 213, if Microsoft Excel is in a busy state it will not request any data from the Universal Messaging RTD Server. Rather than drop events, the Universal Messaging RTD server will continue to push all events onto internal event queues.

If events are rapidly published onto a channel or Excel remains in a busy state indefinitely (if a dialog box is not responded to), without the high/low watermark mechanism, the queues would continue to grow and use system resources.

The watermarks refer to the queue length and can be set per channel using the "RTD function" on page 209. Once the event queue length for a particular channel reaches the high watermark, any incoming events will be caused to wait which will trigger flow control handled by Universal Messaging. Once events are popped off the queue and the queue length reaches the low watermark, the incoming events will be notified to continue and then event queue will begin to refill.

OnChange() Event Using RTD

When cells are updated using the RTD function, the onChange() event for that cell is not triggered. It is not possible to fully recreate this functionality but there are several methods to produce a similar result.

Alternative Solutions

User Defined Function (UDF)

Microsoft Excel functions are recalculated whenever the value of that functions parameters change. This means that a function can be created in cell A1 with a parameter reference of cell A2. When the value of cell A2 changes, the function in cell A1 will recalculate and give a similar functionality to that of the onChange() event.

There are several limitations to what actions can be performed using this method. For example Excel 2003 will not allow any formatting of cells inside a function and Excel 2007 also places certain restrictions. For more information please see limitations with user defined functions on the Microsoft website.

onCalculate()

The onCalculate() event is called whenever a calculation takes place on the worksheet. When an RTD Server is used, this event is triggered whenever new data is sent to Excel. This means the event is potentially triggered very often if a low throttle interval (see "Setting the RTD Throttle Interval" on page 213) is used so it is advised to keep any code in this section to a minimum. This event does not have any parameters so it is up to the user to determine which cells have changed during the calculation.

Prerequisites

Pub/Sub in VBA uses libraries written using the C# API. Please refer to the C# Prerequisites for C# specific requirements.

.NET Framework

Because Universal Messaging VBA makes use of Universal Messaging C# libraries, it requires Microsoft .Net Framework, which you can download from the Microsoft Download website at http://www.microsoft.com/downloads/.

Subscribing

To access the Universal Messaging RTD server installer please contact support. The installer will register the RTD server in the windows registry so that it can be found by the RTD function in Microsoft Excel.

Microsoft Excel Versions

Universal Messaging VBA has been tested on Excel version 2003 and 2007. The Universal Messaging RTD Server has been compiled using Excel 2003 Primary Interop Assemblies (PIA). Due to backwards compatibility, Excel 2007 is able to run with this version of PIA which means that the same version of the Universal Messaging RTD server can be run on both versions of Microsoft Excel.

Publishing

To publish from Excel, you must set a reference to the NirvanaExcel.tlb type library. To access this library please contact support. This library will allow you to create and publish events from within VBA.

The type library is essentially a wrapper for the Universal Messaging C# API to make it visible from Excel.

Macro Security

Publishing events requires code to be written in VBA. If macros are not enabled you will not see any events published as the VBA code is not allowed to run.

Enterprise Developer's Guide for Python

This guide describes how to develop and deploy Enterprise-class Python applications using Universal Messaging, and assumes you already have Universal Messaging installed.

> **Note:**
> The Client API for Python is deprecated in Universal Messaging v10.2 and will be removed in a future version of the product.

Enterprise Client Development

Environment Configuration

The Python API for Universal Messaging uses a C++ wrapper library to expose functionality from the C++ API of Universal Messaging in Python. Therefore the Python API has the same dependencies as the C++ API, some of which do not ship with the product.

OpenSSL

The C++ Client for Universal Messaging uses OpenSSL for secure connections. The OpenSSL libraries required by the C++ libraries are delivered with the Universal Messaging installation.

Running the Sample Applications

Once you have installed Universal Messaging, the sample applications can be found in `<InstallDir>\UniversalMessaging\python\examples`.

To run the applications you first need to set up the required paths. The path setting should include the following directories:

```
<InstallDir>/UniversalMessaging/cplus/lib/x86_64
<InstallDir>/common/security/openssl/lib
<InstallDir>/common/security/openssl/bin
```

Python Version

The Python libraries for Universal Messaging are compatible with the 2.7 version of Python.

Creating a Session

To interact with a Universal Messaging Server, the first thing to do is initialize a Universal Messaging Session object, which is effectively your logical and physical connection to one or more Universal Messaging Realms.

Creating a Universal Messaging Session Object

A Universal Messaging Session object (called NirvanaSession) is contained within the `NirvanaPython` library so you must first include the library and then initiate a new NirvanaSession.

```
    from NirvanaPython import *
    NirvanaModule = NirvanaSession()
```

If the NirvanaPython library is not in the same directory as the application you are writing then you will need to make sure the directory containing the library is in the Python sys path:

```
    import sys
    sys.path.append('..\\bin\\Win32\\Python26\\')
```

If you have problems importing the NirvanaPython library then it may be that one of the other dependencies are missing. Please make sure you have dealt with the prerequisites (see "Environment Configuration" on page 217)

Connecting to a Universal Messaging Realm

Once the NirvanaSession object has been initialised, you can connect to a Universal Messaging Realm as follows:

```
    rname="nsp://localhost:9000"
    NirvanaModule.connect(rname)
```

For information of the parameters you can pass to the connect() function e.g a user name, you can enter:

```
    help(NirvanaModule.connect)
```

Subscribing to a Channel/Topic or Queue

In the NirvanaPython API there is no object which represents a Universal Messaging Channel or Queue. In order to subscribe you simply pass the name of the destination to the NirvanaSession.subscribe method along with the NirvanaCallback object which will receive the asynchronous events.

Creating a NirvanaCallback Object

Asynchronously receiving events requires an object which implements the NirvanaPython.NirvanaCallback interface. The interface has one method, onMessage which is passed a nConsumeEvent object (see "Universal Messaging Events" on page 223).

```
class NirvanaCallback(NirvanaPython.NirvanaCallback):
    def onMessage(self,message):
        print "received an event"
listener = NirvanaCallback()
```

Registering the NirvanaCallback Object to Receive Events

Once the NirvanaCallback object is created you need to register that object as a listener on the Universal Messaging Channel or Queue. First of all you need to construct a NirvanaSession (see "Creating a Session" on page 217). Then you can call the NirvanaSession.subscribe method where the first parameter is the name of the Universal Messaging Channel or Queue that you wish to subscribe to and the second parameter is the Universal Messaging Callback object.

```
mySession = NirvanaSession()
mySession.connect("nsp://localhost:9000")
chanName="demochannel"
mySession.subscribe(chanName,listener)
```

Once the subscription has been registered, the onMessage method of the NirvanaCallback object will be invoked whenever a message is published onto the channel named "demochannel".

DataStream - Receiving DataGroup Events

Python clients can (optionally) act as a DataStream, which allows them to receive events from DataGroups of which they are made members.

The NirvanaSession can be initialised to receive DataGroup events by passing a NirvanaCallback object into the connect method.

Creating a NirvanaCallback Object

Asynchronously receiving events requires an object which implements the NirvanaPython.NirvanaCallback interface. The interface has one method, onMessage which is passed a nConsumeEvent object (see "Universal Messaging Events" on page 223).

```
class NirvanaCallback(NirvanaPython.NirvanaCallback):
    def onMessage(self,message):
        print "received an event"
listener = NirvanaCallback()
```

Registering as a DataStream

In order to register the NirvanaSession as a DataStream, you simply need to pass the NirvanaCallback object into the connect method of NirvanaSession along with the RNAME (see "Creating a Session" on page 217).

```
mySession = NirvanaSession()
mySession.connect("nsp://localhost:9000",listener)
```

Publishing Events to a Channel or Queue

Once the NirvanaSession has been established with the Universal Messaging realm server, a new Universal Messaging Event object (nConsumeEvent) must be constructed prior to use in the publish call being made to the channel.

Note that in this example code, we also create a Universal Messaging Event Dictionary object (nEventDictionary) for our Universal Messaging Event before publishing it:

```
chanName = "demoChannel"
props = nEventProperties()
props.put("exampleKey", "Hello World")
event = nConsumeEvent(props,"aTag")
mySession.publish(chanName,event)
```

Note that there is no Universal Messaging Channel or Queue object, you simply pass the name of the destination (channel or queue) to the publish method.

The underlying library (written using the Universal Messaging C++ API) will find the Channel or Queue object the first time the destination is accessed. So if you pass the name of a Channel which does not exist then you will receive an exception.

Other than initially finding the channel, publish calls are asynchronous so the publish call will immediately return allowing the client to continue. This means that if there is an exception on the server e.g. the client does not have permission to publish to the destination, there will be no client side exception unless you use an asynchronous exception listener.

Writing an Event to a DataGroup

Once the NirvanaSession has been established with the Universal Messaging realm server, a new Universal Messaging Event object (nConsumeEvent) must be constructed.

Note that in this example code, we also create a Universal Messaging Event Dictionary object (nEventDictionary) for our Universal Messaging Event before publishing it:

```
datagroupname = "myDataGroup"
props = nEventProperties()
props.put("exampleKey", "Hello World")
event = nConsumeEvent(props,"aTag")
mySession.writeDataGroup(datagroupname,event)
```

Note that there is no Universal Messaging DataGroup object, you simply pass the name of the DataGroup you wish to publish to.

The underlying library (written using the Universal Messaging C++ API) will create the DataGroup if it does not exist on the Universal Messaging Realm Server.

Asynchronous Exception Listener

Certain methods within the Universal Messaging Python Client API require synchronous calls to the server. For example the NirvanaSession.getLastEID method will request the most recent event ID that was published onto a Universal Messaging Channel. This method is required to be synchronous i.e. must block until a response is received. Other methods such as NirvanaSession.publish do not require a response so to make these methods as fast as possible, they are asynchronous.

With synchronous calls, if an exception is thrown on the server e.g. the user does not have permission to get the last event ID then the exception is passed back in the response and thrown on the client.

With asynchronous calls, the client does not wait for a response so if an exception is thrown on the server e.g. the user does not have permission to publish, the client will not know that the event was not successfully published. This is where it is useful to have an *Asynchronous Exception Listener*.

The Asynchronous Exception Listener will receive notification of exceptions that occurred on the server for asynchronous calls. So if the user was not allowed to publish, the listener will be notified with a message indicating this.

Creating a Asynchronous Exception Listener

Asynchronously receiving exceptions requires an object which implements the NirvanaPython.AsyncExceptionListener interface. The interface has one method, onException which is passed a string describing the exception.

```
class AsyncExceptionListener(NirvanaPython.AsyncExceptionListener):
```

```
    def onException(self,message):
        print "Received an exception -> "+message
exceptionListener = AsyncExceptionListener()
```

Registering the listener for events

In order to register the NirvanaPython.AsyncExceptionListener to receive notification of exceptions, you can call the addAsyncExceptionListener method of NirvanaSession (see "Creating a Session" on page 217).

```
mySession = NirvanaSession()
mySession.connect("nsp://localhost:9000")
mySession.addAsyncExceptionListener(exceptionListener)
```

Synchronously Requesting Events

Although in most circumstances it is more efficient to consume events asynchronously. The Universal Messaging Python API also provides the ability to request events one by one from the server.

Once you have created a session you can create an iterator for the channel or queue that you wish to consume from.

```
iter = NirvanaModule.getIterator(channame,startEid, selector,
       timeout)
for evt in iter:
    doSomething(evt)
```

On each iteration the Python client will request an event from the server and receive the event back as a response. Once the client has consumed all of the events on the channel/queue, it will wait for *timeout* milliseconds for another event to be received. When the client times out it will stop iterating.

Alternatively you can manually request events from the server:

```
evt = iter.next()
```

Once all events are consumed the `next()` method will time out and return `None`.

Sample Applications

Publish / Subscribe using Channel Topics

Channel Publisher

This example shows how publish events onto a Universal Messaging Channel

Application Source Code

See the online documentation for a code example.

Asynchronous Channel Subscriber

This examples show how to connect to a Universal Messaging Channel and asynchronously receive messages.

Application Source Code

See the online documentation for a code example.

Channel Iterator

This example shows how to iterate over events stored on a Universal Messaging Channel

Application Source Code

See the online documentation for a code example.

Publish / Subscribe using Data Streams and Data Groups

DataGroup Publisher

This is a simple example of how to delete a DataGroup

Application Source Code

See the online documentation for a code example.

DataStream Listener

This example shows how to initialise a session ready to asynchronously receive events via DataGroups.

Application Source Code

See the online documentation for a code example.

Message Queues

Queue Publisher

This example shows how publish events onto a Universal Messaging Queue

Application Source Code

See the online documentation for a code example.

Asynchronous Queue Consumer

This examples show how to connect to a Universal Messaging Queue and asynchronously receive messages.

Application Source Code

See the online documentation for a code example.

Synchronous Queue Reader

This example shows how to synchronously pop messages off a Universal Messaging Queue.

Application Source Code

See the online documentation for a code example.

Python Objects

Universal Messaging Events

A Universal Messaging Event (nConsumeEvent) is the object that is published to a Universal Messaging Channel, Queue or Data Group. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an Universal Messaging Event Dictionary (nEventProperties).

Constructing an Event

In this Python code snippet, we construct our Universal Messaging Event object (nConsumeEvent), as well as a Universal Messaging Event Dictionary object (nEventProperties) for our Universal Messaging Event:

```
props = nEventProperties()
props.put("bondname", "bond1")
props.put("price", 100.00)
event = nConsumeEvent(props,"Tag")
```

Handling a Received Event

When a client subscribes to a channel and specifies a callback function to handle received events, the callback function will be invoked with the event as its parameter whenever an event is received.

In this Python code snippet, we demonstrate a simple implementation of such a callback function. In this example, we assume that the event contains an Event Dictionary with two keys: *bondname* and *price*.

```
class myCallback(NirvanaPython.NirvanaCallback):
```

```
    def onMessage(self,event):
        props = event.getProperties()
        name = props.get("bondname")
        price = props.get("price")
        //do something with name and price
```

Event Dictionaries

Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within a Universal Messaging Event.

Event Dictionaries are quite similar to a hash table, supporting primitive types, arrays, and nested dictionaries.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's dictionary.

In this code snippet, we assume we want to publish an event containing the definition of a bond, say, with a name of "bond1":

```
props = nEventProperties()
props.put("bondname", "bond1")
props.put("price", 100.00)
event = nConsumeEvent(props,"Tag")
NirvanaModule.publish("Channelname",evt);
```

Note that in this example code, we also create a new Universal Messaging Event object (nConsumeEvent) to make use of our Event Dictionary (nEventProperties).

3 Mobile Clients

- Overview of Mobile Clients 226
- iPhone Applications 226
- Android Applications 226

Overview of Mobile Clients

Our mobile messaging solution allows developers to implement real-time publish/subscribe functionality within mobile phone applications on a range of devices including Apple iPhone and Google Android:

- **Apple iPhone**

  Our *Universal Messaging API for iPhone* is implemented natively in C++, and through Objective-C and C++ code offers a core subset of Universal Messaging client functionality which allows iPhone to publish and subscribe to Universal Messaging channels, and to asynchronously receive events in realtime:

  Please contact us for a live demonstration, or for access to our Universal Messaging for Apple iPhone API.

- **Android**

  Google Android devices are able to make use of our *Universal Messaging Enterprise API for Java* to subscribe to Universal Messaging channels and utilize message queues.

See Universal Messaging's Language API Comparison Grid for an overview of basic differences between the APIs.

iPhone Applications

This guide describes how to develop and deploy Apple iPhone applications using Universal Messaging, and assumes you already have Universal Messaging installed.

Universal Messaging iPhone Client Development

Universal Messaging for the iPhone is provided through a port of our Universal Messaging C++ library. The iPhone development environment supports both Objective-C and C++ and allows resources from either environment to coexist and be accessible from the other.

Universal Messaging for the iPhone is delivered as a suite of static libraries built for the platform along with their associated header files. Dragging the libraries into XCODE automatically includes them in your project.

Android Applications

Universal Messaging for Google Android is supported through our *Universal Messaging Enterprise API for Java*.

Using the Enterprise Client API

To use Universal Messaging within your Android project, you must reference the JAR file `nClient.jar` for the Enterprise Client API for Java, found in your Universal Messaging installation. References may typically be made by simply dragging the JAR into your IDE.

Documentation

The Universal Messaging Enterprise Developer's Guide for Java provides information on how to use *pub/sub* and *message queues* in your Android application.

Portions of this page are modifications based on work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.

4 Web Client APIs

- Overview of Web Client APIs 230
- Web Developer's Guide for Javascript 230
- Web Developer's Guide for Java 272

Overview of Web Client APIs

Our web-based messaging solution allows developers to implement real-time publish/subscribe functionality into browser applications or RIAs (Rich Internet Applications) using *JavaScript* or *Java*:

- **JavaScript**

  The *Universal Messaging JavaScript API* is a *pure JavaScript solution*. This allows developers to use JavaScript and HTML to build Ajax/Comet clients which can publish and subscribe to Universal Messaging channels, and asynchronously receive events in realtime:

  - "JavaScript Developer's Guide" on page 230

  Our JavaScript API is popular because it works without plugins or infrastructure workarounds, using only the browser's built-in JavaScript engine.

- **Java**

  The *Universal Messaging Java* client APIs can be used for standalone Java applications, but can also be used in the browser as either Applets or Java Webstart applications.

  - "Java Developers Guide for Web Developers" on page 272

  Note that the above Universal Messaging Java links are for web-based applications. Universal Messaging Java APIs can also be used for enterprise clients and servers, as well as mobile applications.

See Universal Messaging's Language API Comparison Grid for an overview of basic differences between each API.

Web Developer's Guide for Javascript

Overview

The following sections describe how to develop and deploy JavaScript applications using Universal Messaging, and assumes you already have Universal Messaging installed.

Universal Messaging supports both WebSocket and Ajax / Comet streaming through our Javascript API. Universal Messaging streams events to web clients asynchronously, without the requirement for any additional technology components at clients' browsers. The API will automatically detect client capabilities and make use of the optimum underlying transport in each case.

Server Configuration for JavaScript

Server Configuration for HTTP Delivery

Universal Messaging can serve web content over both HTTP and HTTPS communication modes. This section discusses the steps necessary to configure a realm server to deliver web content over HTTP.

Creating a Universal Messaging HTTP (nhp) Interface

> **Note:**
> Universal Messaging ships with an HTTP Interface enabled by default.

Universal Messaging provides its own protocol, the Universal Messaging HTTP Protocol (nhp) for the delivery of web content over HTTP. For web communication to take place an interface using this protocol must be created. Creating an interface can be done through the enterprise manager.

Serving Content through File Plugins

A Universal Messaging nhps interface delivers content to connected browsers through file plugins. Generally at least two file plugins will need to be configured to serve a page using the Universal Messaging JavaScript API. The first will be a pointer to the Universal Messaging JavaScript client libraries. The second will be a plugin pointing to the base directory of the web pages which use these libraries.

The Universal Messaging JavaScript client libraries are located in `<InstallDir>/UniversalMessaging/javascript/lib`. To use these libraries in any content served from an interface a file plugin with a `BasePath` which points to this directory is necessary. The `URL Path` of the file plugin may be anything you wish, though it must be referenced the same in the include in your JavaScript code. For example, if you set the `URL Path` to `/lib/js` then the following code must be included in your pages:

```
<script language="JavaScript" src="/lib/js/nirvana.js"></script>
```

> **Note:**
> Universal Messaging ships with a file plugin with the base path `/lib/js` and the above configuration.

The file plugin which points to your web content is configured in a similar way. The `BasePath` should point to the fully qualified path of your web directory. The `URL Path` is the resource location relative to your address. For example, serving content from the root of the website can be done by setting a `URL Path` of `/`.

If you prefer, you can host your web application on a different web server entirely. In addition, nirvana.js could be served from such a web server. The Universal Messaging realm server's interface's file plugin (/lib/js in this case) will only be required if you opt to use any of the JavaScript drivers that use postMessage for cross domain communication (see JavaScript driver details for more information).

JavaScript Interface Properties

The behaviour of nhp interfaces when serving web content can be changed through the enterprise manager. These settings can be changed by editing configuration properties available in the JavaScript panel accessed through the interface tab.

Comet Configuration Properties

The Universal Messaging enterprise manager also provides realm wide configuration settings for Comet. These are available in the enterprise manager from the Comet Config panel.

Server Configuration for HTTPS Delivery

Universal Messaging can serve web content over both HTTP and HTTPS communication modes. This section discusses the steps necessary to configure a realm server to deliver web content over HTTPS.

Creating a Universal Messaging HTTPS (nhps) Interface

Universal Messaging provides its own protocol, the Universal Messaging HTTPS Protocol (nhps) for the secure delivery of web content over HTTPS. For web communication to take place an interface using this protocol must be created. Creating an interface can be done through the enterprise manager.

Enabling SSL on the Interface

When the interface is created using the enterprise manager default values are placed into the Certificates tab in the interface panel. To communicate using HTTPS over the interface additional configuration in this panel is required.

Other Configuration Options

Once the interface is created and SSL is enabled and correctly set up on the interface configuration can be completed by using the same steps which apply to configuring a HTTP interface.

Serving From Another Webserver

The Universal Messaging JavaScript API consists of two files:

- `nirvana.js` (which can be served from any webserver)
- `crossDomainProxy.html` (needed only if using one of the postMessage drivers, and which must be served from a file plugin on the Universal Messaging realm server)

Universal Messaging Realm Servers provide the option of exposing an HTTP web server interface for serving files to clients, removing the need to install a third party web server for hosting applications. Of course, it is possible to use a third party web server to host applications too.

Here we will explain how to deploy applications in both scenarios.

Web Applications on a Realm File Plugin

Your application source code, and the Universal Messaging library files shown above, need to be deployed to one or more directories on the Realm Server, and File Plugins configured to provide access to these directories.

> **Note:**
> Universal Messaging ships with an HTTP Interface enabled by default. This HTTP Interface is pre-configured with a file plugin with the base path `/lib/js` which points to the directory containing the above library files.

As a result, both files are accessible via a browser at the following paths on the realm:

- `/lib/js/nirvana.js`
- `/lib/js/crossDomainProxy.html`

To use Universal Messaging, applications then simply need to include nirvana.js as follows:

```
<script src="/lib/js/nirvana.js"></script>
```

There is no need to reference the crossDomainProxy.html file directly (the nirvana.js library will load it automatically if it is required).

Your Universal Messaging session can be started with a relatively simple configuration, as follows:

```
var mySession = Nirvana.createSession({
        applicationName : "myExampleApplication",
        sessionName     : "myExampleSession",
        username        : "testuser"
});
mySession.start();
```

Web Applications on a Third Party Web Server

Your application source code and HTML files, and optionally the `nirvana.js` library (which may in fact be served from any server, including a CDN), are deployed to a third party web server, such as Apache.

If there is any chance that your client will use a postMessage drivers, then you must ensure that the `crossDomainProxy.html` file is accessible on the realm via a file plugin.

If you use the default file plugin configuration mentioned above, then no further configuration is required. If instead you decide to make the `crossDomainProxy.html` file available at a different path by using a different file plugin, then you will need to specify this path as a `crossDomainPath` key in the session configuration object passed to `Nirvana.createSession()`.

For any driver other than WEBSOCKET, the third party web server must be using the *same protocol* (i.e. http or https) as the Universal Messaging Realm interface file plugin, and running on the *same port*. The WEBSOCKET driver does not have this restriction.

To use Universal Messaging:

1. Applications need to include nirvana.js as follows:

```
<script src="/front/end/server/lib/nirvana.js"></script>
```

2. The `Nirvana.createSession()` call must use a configuration object that includes the following key/value pair:

   - `realms` : *An array of URLs of the realm servers in use, e.g.*

     `["http://node1.um.softwareag.com:80", "http://node2.um.softwareag.com:80"]`

3. Your Universal Messaging session can then be started with a configuration such as:

```
var mySession = Nirvana.createSession({
        realms : ["http://node1.um.softwareag.com:80",
                  "http://node2.um.softwareag.com:80"],
        applicationName : "myExampleApplication",
        sessionName     : "myExampleSession",
        username        : "testuser"
});
mySession.start();
```

For more information, please see Universal Messaging Sessions in JavaScript, which describes in more detail the options that can be set using the Universal Messaging session configuration object.

Web Client Development in JavaScript

Creating a Session

To interact with a Universal Messaging Server, the first thing to do is configure and start a Universal Messaging Session object, which is effectively your logical and physical connection to one or more Universal Messaging Realms.

Configuring a Universal Messaging Session Object

A Universal Messaging session object is created using the `Nirvana.createSession()` function. Once created, a session can be *started* at will:

```
var mySession = Nirvana.createSession();
```

The `Nirvana.createSession()` function may be passed an optional configuration object as a parameter, as follows:

```
var mySession = Nirvana.createSession({
  realms            : [ "http://showcase.um.softwareag.com:80" ],
                      // this can be an array of realms
  debugLevel        : 4, // 1-9 (1 = noisy, 8 = severe, 9 = default = off)
  sessionTimeoutMs  : 10000,
  enableDataStreams : false,
  drivers           : [ // an array of transport drivers in preferred order:
    Nirvana.Driver.WEBSOCKET,
    Nirvana.Driver.XHR_STREAMING_CORS,
    Nirvana.Driver.XDR_STREAMING,
    Nirvana.Driver.JSONP_LONGPOLL
```

```
  ]
});
```

For a full list of the parameters used in this configuration object, please see the JavaScript API Documentation for Nirvana.createSession().

Starting your Session

Once a session object has been created, the session may be started as follows:

```
var mySession = Nirvana.createSession();
mySession.start();
```

Session Start Callbacks

When a Universal Messaging Session is successfully started following a call to start(), an asynchronous callback will be fired. You can assign a function to be fired as follows:

```
function sessionStarted(s) {
        console.log("Session started with ID " + s.getSessionID());
}
var mySession = Nirvana.createSession();
mySession.on(Nirvana.Observe.START, sessionStarted);
mySession.start();
```

For more details on methods available on a session object, please see the JavaScript API Documentation for the Universal Messaging Session object.

Publish/Subscribe Tasks

Overview of using Publish/Subscribe

The Universal Messaging JavaScript API provides publish subscribe functionality through the use of channel objects. Channels are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Under the publish / subscribe paradigm, each event is delivered to each subscriber once and only once per subscription, and is not removed from the channel after being consumed.

This section demonstrates how Universal Messaging pub / sub works, and provides example code snippets for all relevant concepts.

Using a Universal Messaging Channel

This JavaScript code snippet demonstrates how to create a channel object, which allows you to publish or subscribe to a Universal Messaging channel:

```
var myChannel = mySession.getChannel("/fxdemo/prices");
```

Note that unlike the Enterprise APIs, the JavaScript API does not support programmatic creation of channels; instead, you can use the Enterprise Manager GUI to create a Universal Messaging Channel.

A channel object offers several methods. Three of the more important ones are:

- `myChannel.subscribe()`
- `myChannel.unsubscribe()`
- `myChannel.publish(Event event)`

Please see JavaScript API Documentation for Channels for information on all available methods on a channel.

Each of the above methods can invoke one or more optional user-specified callback functions which you can (and probably should) implement and assign as follows:

```
var myChannel = mySession.getChannel("/fxdemo/prices");
// Assign a handler function for Universal Messaging Events received on the Channel,

// then subscribe:
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + " " + dictionary.get("bid"));
}
myChannel.on(Nirvana.Observe.DATA, myEventHandler);
myChannel.subscribe();
```

See “Subscribing to a Channel” on page 236 and “Publishing Events to a Channel” on page 237.

Subscribing to a Channel

Once a Universal Messaging Channel object has been created, you can subscribe to the channel, and receive Universal Messaging Events published on the channel.

Simple Subscription

This JavaScript code snippet demonstrates how to subscribe to a channel:

```
var myChannel = mySession.getChannel("/fxdemo/prices");
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + " " + dictionary.get("bid"));
}
myChannel.on(Nirvana.Observe.DATA, myEventHandler);
myChannel.subscribe();
```

Note that the `subscribe()` call is asynchronous; it returns immediately, allowing single-threaded JavaScript clients to continue processing. Whenever an event is received on the channel, however, any user function assigned as a callback for the observable event `Nirvana.Observe.DATA` will be invoked, with the appropriate Event as its parameter.

Subscription with a Filtering Selector

It is also possible to subscribe to a channel with a user-specified *selector* (a type of filter), ensuring that your client receives only events that match the selector. Selectors are SQL-like statements such as:

- name LIKE '%bank%' AND description IS NOT NULL
- (vol > 0.5 OR price = 0) AND delta < 1

This JavaScript code snippet demonstrates how to subscribe to a channel and receive only events which have a key named "volatility" and a value greater than 0.5:

```
var myChannel = mySession.getChannel("/fxdemo/prices");
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + " " + dictionary.get("bid"));
}
myChannel.on(Nirvana.Observe.DATA, myEventHandler);
myChannel.setFilter("name like '%EUR%'");
myChannel.subscribe();
```

Handling Errors

You may optionally specify an error handler to be notified of subscription or publishing errors:

```
function myErrorHandler(error) {
        console.log(error.message);
}
myChannel.on(Nirvana.Observe.ERROR, myErrorHandler);
```

If you do not implement an error handler in this way, errors will be silently ignored.

Publishing Events to a Channel

Once the session has been established with the Universal Messaging realm server, and a Universal Messaging Channel object has been created, a new Universal Messaging Event object must be constructed to use in the publish call being made on the channel.

Note that in this example code, we also create a Universal Messaging Event Dictionary object for our Universal Messaging Event before publishing it:

```
var mySession = Nirvana.createSession();
var myChannel = mySession.getChannel("/tutorial/sandbox");
var myEvent = Nirvana.createEvent();
var myDict = myEvent.getDictionary();
myDict.putString("demoMessage", "Hello World");
myChannel.publish(myEvent);
```

Note that the publish call is asynchronous; it returns immediately, allowing single-threaded JavaScript clients to continue processing.

To enable the developer to know when a publish call has completed, any user function assigned as a callback for the channel's observable event `Nirvana.Observe.PUBLISH` will be invoked, with the a string value of *"OK"* (which indicates the publish was successful):

```
function publishCB(responseString) {
        console.log("Publish attempt: " + responseString);
}
myChannel.on(Nirvana.PUBLISH, publishCB);
myChannel.publish(myEvent);
```

Transactional Publish

Transactional publishing provides a means of verifying that the server has received events from the publisher, and therefore provides guaranteed delivery. Clients can publish using transactions to both channels and queues in Javascript.

Transactions can be created by the user from a "Universal Messaging Event" on page 102 object and a Universal Messaging Transaction object. The event can then be published through the transaction object to the server.

```
var demoSession = Nirvana.createSession();
var myChannel = demoSession.getChannel("/example/txChannel");
var dataListener = function(event) {
        console.log("Received Event from Channel");
};
myChannel.on(Nirvana.Observe.DATA, dataListener);
myChannel.subscribe();

var myTransaction = myChannel.createTransaction();
var commitListener = function() {
        console.log("Received Commit Callback from Publish");
};

myTransaction.on(Nirvana.Observe.COMMIT, commitListener);
var myEvent = Nirvana.createEvent();
myEvent.setData("Hello World");
myTransaction.setEvent(myEvent);
myTransaction.publishAndCommit();
```

The transaction's observable event `Nirvana.Observe.COMMIT` is fired after a successful publish request once the client receives confirmation from the server that the event has been published. This will result in the invocation of any user-assigned listener functions, as in the example code above.

Similarly, a transaction's observable event `Nirvana.Observe.ERROR` is fired after when the client receives confirmation from the server that a problem occurred resulting in the event not being published.

```
var errorListener = function(error) {
        console.log(error.message);
};
myTransaction.on(Nirvana.Observe.ERROR, errorListener);
```

In scenarios where a problem occurs, the client may not receive either of these callbacks. This may be either due to server side or client side failure. In these scenarios the state of the transaction from the clients perspective is ambiguous.

By invoking the `myTransaction.checkCommitStatus(queryServer)` method the client will attempt to resolve the state of the server. It will first attempt to do this locally; if it can do this the method will instantaneously invoke the callback method with the transaction status. If it cannot do this and `queryServer` is set to `true`, it will contact the server for confirmation and pass this confirmation to any callback method associated with the transaction's observable event `Nirvana.Observe.COMMIT`. If `queryServer` is set to `false` it will immediately invoke the callback with the failure status.

DataStream - Receiving DataGroup Events

JavaScript clients can (optionally) act as a DataStream, which allows them to receive events from DataGroups of which they are made members.

The process for enabling DataStream functionality is quite simple:

1. Pass a configuration object to the Nirvana.createSession() call with a suitable configuration parameter (`enableDataStreams`).
2. Implement the Session.on() callback function.

Processing events received as a DataStream is also very simple:

```
var mySession = Nirvana.createSession({ enableDataStreams : true });
function myDGEventHandler(event) {
        console.log("Received a DataGroup Event");
}
mySession.on(Nirvana.Observe.DATA, myDGEventHandler);
mySession.start();
```

Note that JavaScript clients can only act as DataStreams (consumers of DataGroup events). The JavaScript API does not currently support publishing to DataGroups or remote management of DataGroup members; DataGroup management is instead supported by Universal Messaging's Enterprise APIs.

Message Queue Tasks

Overview of using Message Queues

The Universal Messaging JavaScript API provides message queue functionality through the use of queue objects. Queues are the logical rendezvous point for publishers (producers) and subscribers (consumers) of data (events).

Message queues differ from publish / subscribe channels in the way that events are delivered to consumers. Whilst queues may have multiple consumers, each event is typically only delivered to one consumer, and once consumed (popped) it is removed from the queue.

This section demonstrates how Universal Messaging message queues work in JavaScript, and provides example code snippets for all relevant concepts.

Using a Queue

This JavaScript code snippet demonstrates how to create a queue object, which allows you to publish or subscribe to a Universal Messaging queue:

```
var myQueue = mySession.getQueue("/demo/prices");
```

Note that unlike the Enterprise APIs, the JavaScript API does not support programmatic creation of queues; instead, you can use the Enterprise Manager GUI to create a Universal Messaging Queue.

A queue object offers several methods. Three of the more important ones are:

- `myQueue.subscribe()`
- `myQueue.unsubscribe()`
- `myQueue.publish(Event event)`

Please see JavaScript API Documentation for Queues for information on all available methods on a queue.

Each of the above methods can invoke one or more optional user-specified callback functions which you can (and probably should) implement and assign as follows:

```
var myQueue = mySession.getQueue("/demo/prices");
// Assign a handler function for Universal Messaging Events received on the Queue,
// then subscribe:
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + " " + dictionary.get("bid"));
}
myQueue.on(Nirvana.Observe.DATA, myEventHandler);
myQueue.subscribe();
```

See “Subscribing to a Queue” on page 240 and “Publishing Events to a Queue” on page 241.

Subscribing to a Queue

Once a Universal Messaging Queue object has been created, you can subscribe to the queue, and receive Universal Messaging Events published on the queue. JavaScript supports two kinds of queue subscribers. An asynchronous non-transactional consumer and a asynchronous transactional consumer.

Simple Subscription

Once a Universal Messaging Queue object has been created, you can subscribe to the channel, and receive Universal Messaging Events published on the queue.

This JavaScript code snippet demonstrates how to subscribe to a queue:

```
var myQueue = mySession.getQueue("/demo/prices");
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + " " + dictionary.get("bid"));
}
myQueue.on(Nirvana.Observe.DATA, myEventHandler);
myQueue.subscribe();
```

Note that the `subscribe()` call is asynchronous; it returns immediately, allowing single-threaded JavaScript clients to continue processing. Whenever an event is received on the queue, however, any user function assigned as a callback for the observable event `Nirvana.Observe.DATA` will be invoked, with the appropriate Event as its parameter.

Handling Errors

You may optionally specify an error handler to be notified of subscription or publishing errors:

```
function myErrorHandler(error) {
        console.log(error.message);
}
myQueue.on(Nirvana.Observe.ERROR, myErrorHandler);
```

If you do not implement an error handler in this way, errors will be silently ignored.

Publishing Events to a Queue

Once the session has been established with the Universal Messaging realm server, and a Universal Messaging Queue object has been created, a new Universal Messaging Event object must be constructed to use in the publish call being made on the queue.

Note that in this example code, we also create a Universal Messaging Event Dictionary object for our Universal Messaging Event before publishing it:

```
var mySession = Nirvana.createSession();
var myQueue = mySession.getQueue("/tutorial/somequeue");
var myEvent = Nirvana.createEvent();
var myDict = myEvent.getDictionary();
myDict.putString("demoMessage", "Hello World");
myQueue.publish(myEvent);
```

Note that the publish call is asynchronous; it returns immediately, allowing single-threaded JavaScript clients to continue processing.

To enable the developer to know when a publish call has completed, any user function assigned as a callback for the queue's observable event `Nirvana.Observe.PUBLISH` will be invoked, with the a string value of *"OK"* (which indicates the publish was successful):

```
function publishCB(responseString) {
        console.log("Publish attempt: " + responseString);
}
myQueue.on(Nirvana.PUBLISH, publishCB);
myQueue.publish(myEvent);
```

Asynchronous Transactional Queue Consuming

Transactional queue consumers have the ability to notify the server when events have been consumed (committed) or when they have been discarded (rolled back). This ensures that the server does not remove events from the queue unless notified by the consumer with a commit or rollback.

Subscribing as a Transactional Reader

This JavaScript code snippet demonstrates how to subscribe to a queue as a transactional queue reader:

```
var demoSession = Nirvana.createSession();
var demoQueue = demoSession.getQueue ("/some/demo/queue");
demoQueue.on(Nirvana.Observe.DATA,
          function(event) {
              // define what to do when we receive an event
          });
```

You can specify the transaction flag and the window size as follows:

```
var demoQueue = mySession.getQueue ("/some/demo/queue", true);
// The true flag specifies that we are a transactional reader
demoQueue.setWindowSize(10); // 10 is the windowSize
demoQueue.subscribe();
```

Performing a Commit

As previously mentioned, the big difference between a transactional reader and a standard queue reader is that once events are consumed by the reader, the consumers need to commit the events consumed. Events will only be removed from the queue once the commit has been called.

The server will only deliver up to the specified windowSize number of events. After this the server will not deliver any more events to the client until commit has been called. The default windowSize is 5.

The JavaScript libraries provide two methods for committing events which have been consumed. `demoQueue.commitAll()` will commit every event which this consumer has received thus far, but has not previously committed. When the server receives this message, all these events will be removed. `demoQueue.commit(event)` will commit the given event and any uncommitted events occurring before.

```
demoQueue.on(Nirvana.Observe.DATA,
          function(event) {
             // process the event
             demoQueue.commit(event); // Commit the event
          });
```

Performing a Rollback

Developers can also roll back events received by the transactional reader. Uncommitted events will be redelivered by the server (possibly to other queue consumers if they exist).

The JavaScript libraries provide two methods for performing a rollback. `demoQueue.rollbackAll()` will roll back all previously uncommitted events which the consumer has received. `demoQueue.rollback(event)` will perform a rollback on all events starting from the given event.

JavaScript Objects

Universal Messaging Events

A Universal Messaging Event is the object that is published to a Universal Messaging channel or queue. It is stored by the server and then passed to consumers as and when required.

Events can contain simple byte array data, or more complex data structures such as an "Universal Messaging Event Dictionary" on page 244.

Constructing an Event

In this JavaScript code snippet, we construct our Universal Messaging Event object, as well as a "Universal Messaging Event Dictionary" on page 244 object for our Universal Messaging Event:

```
var myEvent = Nirvana.createEvent();
var myDict = myEvent.getDictionary();
myDict.putString("demoMessage", "Hello World");
```

Handling a Received Event

When a client subscribes to a channel and specifies a callback function to handle received events, the callback function will be invoked with the event as its parameter whenever an event is received.

In this JavaScript code snippet, we demonstrate a simple implementation of such a callback function. In this example, we assume that the event contains an Event Dictionary with two keys: *name* and *price*.

```
var myChannel = mySession.getChannel("/fxdemo/prices");
function myEventHandler(event) {
        var dictionary = event.getDictionary();
        console.log(dictionary.get("name") + ": " + dictionary.get("price"));
}
myChannel.on(Nirvana.Observe.DATA, myEventHandler);
myChannel.subscribe();
```

Universal Messaging Event Attributes

Universal Messaging Event Attributes are objects which contain meta-data about a "Universal Messaging Event" on page 243. A Client can query the attributes of a delivered event to provide specific handling for different types of event depending on its attributes.

Some header information contained in an Event Attributes object can be set by the client prior to publishing the Event. This information will subsequently be delivered to any clients who receive it.

Obtaining an nEventAttributes object

If an Event does not explicitly have any Event Attributes, then a new, empty Event Attributes object is initialized for the Event with a call to `getEventAttributes()`:

```
var attribs = someEvent.getEventAttributes();
attribs.setPublisherName("John Doe");
myChannel.publish(someEvent);
```

For more detailed usage information, please see the JavaScript API Documentation for Event Attributes.

Merging and Registered Events

"Registered Events" on page 245 facilitate delivery of partial events containing only the data which has changed to a client. A publisher can publish and subsequently update a registered event through JavaScript by using the `setAllowMerge` method of nEventAttributes.

```
myEvent.putDictionary(aDictionary);
myEvent.getEventAttributes().setAllowMerge(true);
myChannel.publish(someEvent);
```

A JavaScript client can query if a received event is a registered event by checking the `isRegistered` flag on the nEventAttributes object

```
if(aEvent.getAttributes().isRegistered()) {
        // Some registered event specific code here...
}
```

Similarly, a JavaScript client which receives an event can query if it is a full or partial event by checking if the `isDelta` flag is set on an nEventAttributes object.

```
if(aEvent.getAttributes().isDelta()) {
        // Some registered delta-specific code here...
}
```

Obtaining the Event TimeStamp

If configured to do so, the realm server will stamp each event it delivers to the client with the time it was received. This timestamp can be accessed through the nEventAttributes object.

```
var age = now - aEvent.getAttributes().getTimeStamp();
```

Other Headers

The nEventAttributes object also contains methods for setting and getting other attributes associated with the object. These include JMS specific headers and details on the origin, destination and join channels. To see the full list of operations available on this object consult the API documentation.

Event Dictionaries

Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within a "Universal Messaging Event" on page 243.

Event Dictionaries are quite similar to a hash table, supporting primitive types, arrays, and nested dictionaries.

Universal Messaging filtering allows subscribers to receive only specific subsets of a channel's events by applying the server's advanced filtering capabilities to the contents of each event's dictionary.

In this code snippet, we assume we want to publish an event containing a string, say, with a name of "demoMessage":

```
var mySession = Nirvana.createSession();
var myChannel = mySession.getChannel("/tutorial/sandbox");
var myEvent = Nirvana.createEvent();
var myDict = myEvent.getDictionary();
myDict.putString("demoMessage", "Hello World");
myChannel.publish(myEvent);
```

Note that in this example code, we also create a new "Universal Messaging Event" on page 243 object to make use of our Event Dictionary (nEventProperties).

Optimizing Throughput

The Merge Engine and Event Deltas

In order to streamline web-based Publish/Subscribe applications, it is possible to deliver only the differences between consecutive events, as opposed to the entire event each time. These *event deltas* minimize the amount of data that needs to be sent from the publisher, as well as the amount of data ultimately delivered to subscribers.

Event Deltas and Publishers

Imagine a channel that is used to deliver foreign-exchange currency prices. Let us assume that the channel has a publish-key named *pair*, of depth 1, representing the currency pair. This means that a maximum of one event for each currency pair will exist on the channel at any time.

An event representing a foreign-exchange currency price might therefore be published as follows:

```
var event = Nirvana.createEvent();
var priceDictionary = myEvent.getDictionary();
priceDictionary.putString("pair", "EURUSD");
priceDictionary.putFloat("bid", 1.2261);
priceDictionary.putFloat("offer", 1.2263);
priceDictionary.putFloat("close", 1.2317);
priceDictionary.putFloat("open", 1.2342);
demoChannel.publish(event);
```

Let us now imagine that the spread on this price has tightened: while the *bid* value remains the same, the *offer* is lowered from 1.2263 to 1.2262.

Under normal circumstances, an entire new event would be created and published:

```
var event = Nirvana.createEvent();
var priceDictionary = myEvent.getDictionary();
priceDictionary.putString("pair", "EURUSD");
priceDictionary.putFloat("bid", 1.2261);
priceDictionary.putFloat("offer", 1.2262);
```

```
priceDictionary.putFloat("close", 1.2317);
priceDictionary.putFloat("open", 1.2342);
demoChannel.publish(event);
```

Notice that the majority of the information in this new event is no different to that in the previously sent event.

Event deltas allow us to *publish only the information that has changed*:

```
var event = Nirvana.createEvent();
var priceDictionary = myEvent.getDictionary();
// we need to specify the publish-key too, of course
priceDictionary.putString("pair", "EURUSD");
priceDictionary.putFloat("offer", 1.2262);
event.getAttributes().setAllowMerge(true);
demoChannel.publish(event);
```

It is clear from the above example that using event delta functionality through `setAllowMerge(true)` in the Event Attributes object is especially useful when publishing events with many dictionary keys that have unchanged values.

Event Deltas and Subscribers

In the above example, where the channel had a publish-key named *pair* with a depth of 1, only one event for each currency will exist on the channel at any one time. Given that the last published event was a mere delta, how can we guarantee that a new subscriber will receive an event with a fully populated dictionary containing all expected keys?

Universal Messaging's Merge Engine will process and merge events with all event deltas, maintaining internal representations of *merged event snapshots*, keyed on the channel's publish-key. A merged event snapshot for each unique publish-key value is delivered to subscribers when they initially subscribe, or when they reconnect after a period of disconnection.

Web clients built using the Universal Messaging JavaScript API can receive any combination of standard events, event deltas and merged event snapshots.

New Subscribers: Merged Events

A client that subscribed to the channel some time after the above example's event delta was published would receive a server-generated *merged event snapshot* with a dictionary containing the following key/value pairs:

- pair : "EURUSD"
- bid : 1.2261
- offer : 1.2262
- close : 1.2317
- open : 1.2342

Note how the *offer* value of 1.2262 has been merged into the older event's dictionary.

Existing Subscribers: Events and Event Deltas

A client that was subscribed before the initial example event was published would receive two events. The first event would have a dictionary containing the following key/value pairs:

- pair : "EURUSD"
- bid : 1.2261
- offer : 1.2263
- close : 1.2317
- open : 1.2342

The second event received by the client (the delta) would be marked as a delta, and have a dictionary containing only the following key/value pairs:

- pair : "EURUSD"
- offer : 1.2262

In summary, therefore, any new client subscribing will receive all of the fields in the merged event for EURUSD, while any existing subscribers will only receive the *offer* change for EURUSD.

Important:
Note that only the event delta is passed to the developer-implemented `Channel.on()` callback; it is the developer's responsibility to make use of the deltas in this case.

Further Notes

- In order for a channel to be capable of delivering deltas and merging events it must be created with the Merge Engine enabled, and it must have a single publish-key. The publish-key represents the primary key for the channel.
- If a publisher of an event does not make a call to `setAllowMerge(true)` then the merged event snapshot for that publish-key value would be replaced in its entirety by the newly published event.
- If a subscriber disconnects and then reconnects it will again receive the latest snapshot before receiving only the deltas that are subsequently published.

JavaScript Communication Drivers and Protocols

Communication Drivers

Overview

JavaScript communication drivers use streaming techniques or long polling, as required.

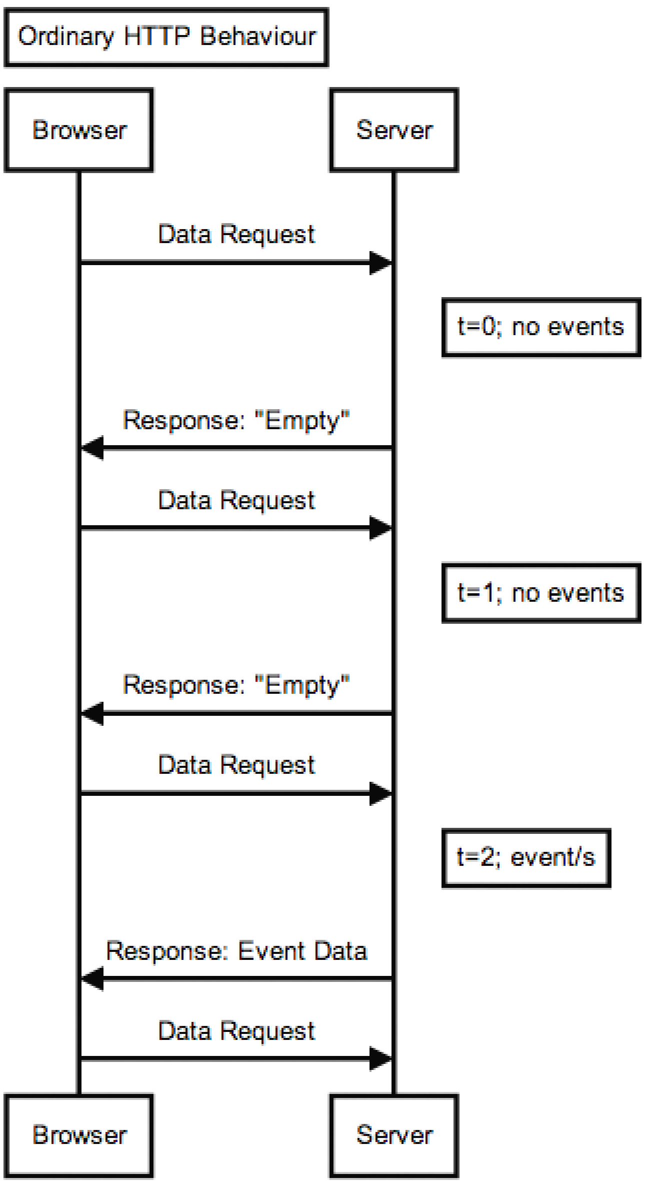
For a full list, please see the JavaScript API Documentation for Drivers.

The following links provide a basic description of the main techniques employed by these drivers:

- "HTML5 WebSocket" on page 262
- " Comet Streaming" on page 268
- "Comet LongPolling" on page 268

Standard HTTP Polling

Most non-Universal Messaging web applications make use of repeated, standard HTTP polling requests. Such application requests/responses look like this:



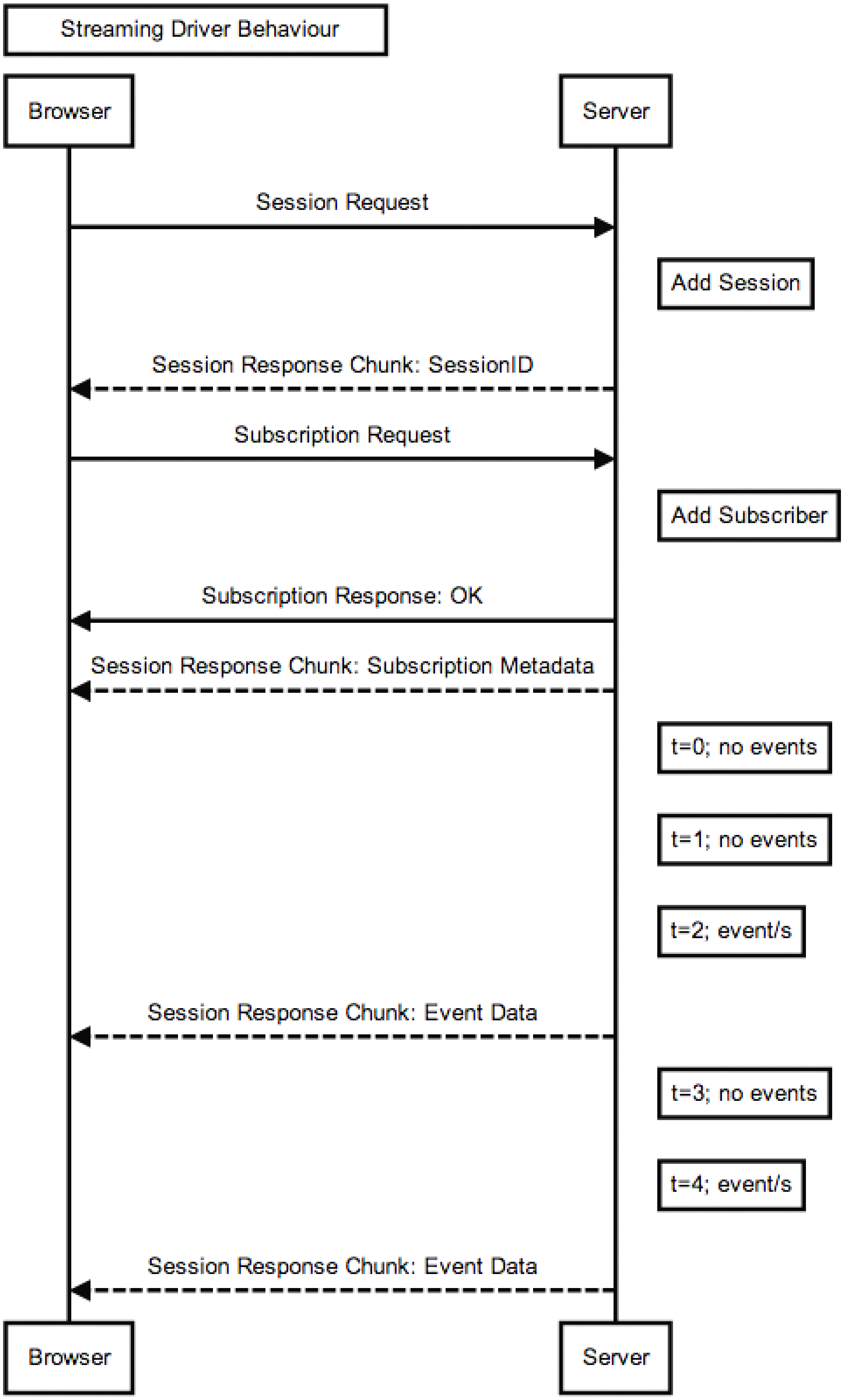
The Universal Messaging JavaScript API is more efficient than this. It implements several underlying drivers for communication between a Universal Messaging JavaScript client and a Universal Messaging realm server. These drivers can be conceptually divided into:

- streaming drivers
- long polling drivers

Streaming Drivers

The Streaming drivers implemented in Universal Messaging take advantage of various technologies implemented in different web browsers, and various mechanisms to achieve HTTP server push or HTTP streaming. These technologies and mechanisms include HTML5 Web Sockets, chunked XMLHTTPRequest and XDomainRequest responses, EventSource/SSE, iFrame Comet Streaming and more.

The fundamental difference between Universal Messaging JavaScript API's Streaming drivers and standard HTTP polling is that the Universal Messaging realm server will not terminate the HTTP connection after it sends response data to the client. The connection will remain open so that if additional data (such as Universal Messaging events) becomes available for the client, it can immediately be delivered to the client without having to be queued and without having to wait for the client to make a subsequent request. The client can interpret the "partial" response chunks as they arrive from the server.

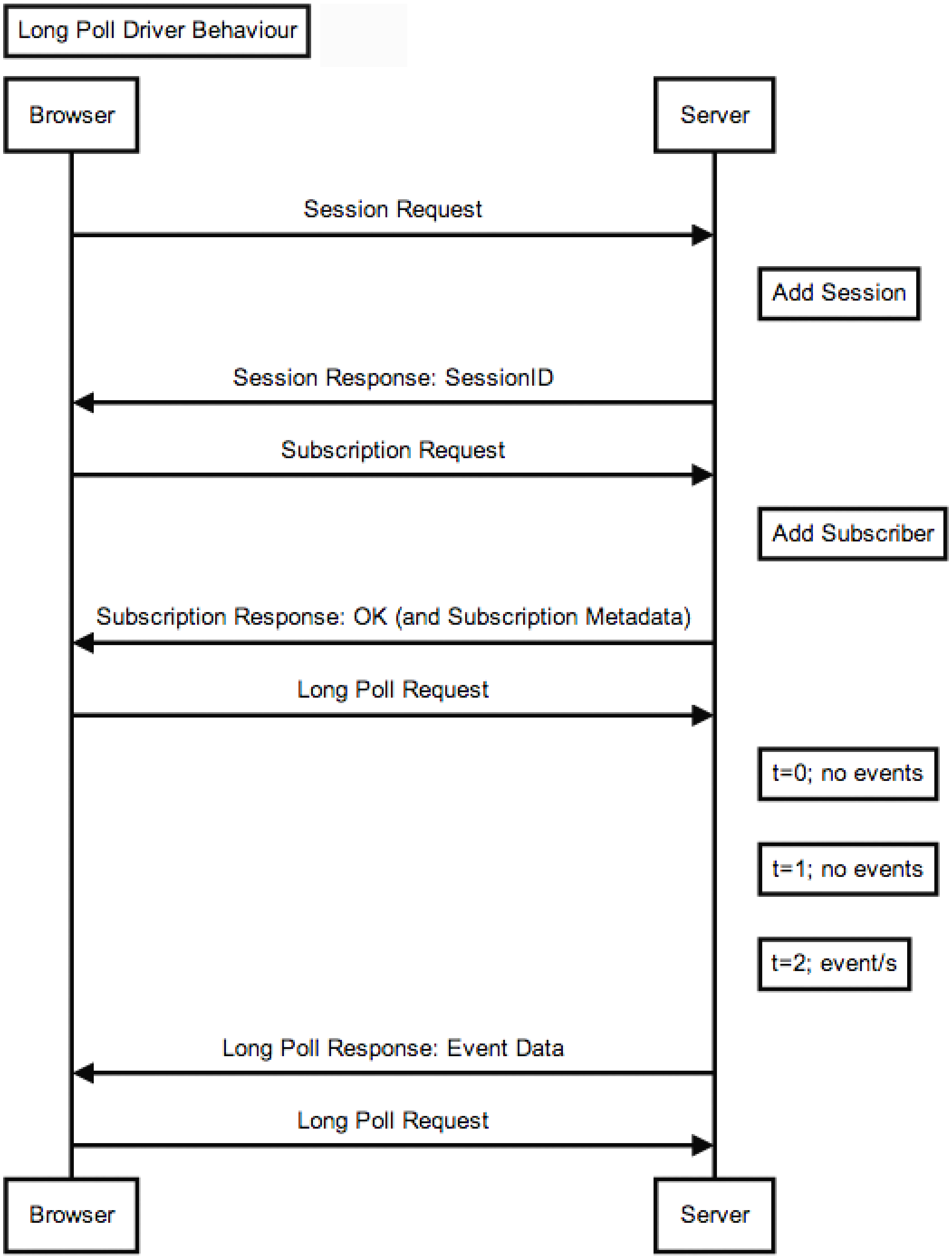


This is much more efficient than standard HTTP polling, since the client need only make a single HTTP request, yet receive ongoing data in a single, long lived response.

Streaming drivers are the preferred drivers to use in a Universal Messaging JavaScript application. Do note, however, that some environments may limit the successful use of streaming drivers (such as intermediate infrastructure with poorly configured client-side proxy servers or reverse proxy servers). In these instances, clients can be configured to fall back to a Long Polling driver (which can be considered a driver "of last resort").

Long Polling Drivers

When using a Long Polling driver, a Universal Messaging client requests information from the realm server in a similar way to a normal HTTP poll. The primary difference is that if the server does not have any information for the client at that time, then instead of sending an empty response and closing the connection, the server will instead hold the request and wait for information (for example, Universal Messaging events) to become available. Once information is available for the client, the server completes its response and closes the connection. The Universal Messaging client will then immediately make a new Long Poll request for the next batch of information:



Clearly, if information is constantly being provided, then a client will end up making very frequent Long Poll requests (potentially as frequently as it would with a normal HTTP poll approach). The Universal Messaging realm server can be configured to delay responding and closing a connection for as long as desired (thereby allowing administrators the option of making requests be fulfilled as soon as data is available, or waiting for a time period, allowing information to accumulate further, before responding - this latter technique is recommended if a client is likely to receive many events per second).

Long Polling drivers are therefore not true "push" drivers, and should only be used under circumstances where real push is not possible (owing, for instance, to intermediate infrastructure such as poorly configured client-side proxy servers or reverse proxy servers).

Infrastructural Issues and Workarounds

Universal Messaging JavaScript clients are intended to receive data in near real-time. To facilitate this, such clients will tend to use one of the API's Streaming drivers (as opposed to a basic, repetitive HTTP polling).

In most environments, a Streaming driver - which works over HTTP - can be used without problem. In some environments, however, infrastructure components can interfere with the real-time HTTP streams. Here we will look at some possible causes of interference, and discuss how to work around them.

Client-Side Proxy Servers

Most client side proxy servers will permit a client to make a long lived connection to a Universal Messaging realm server (to see why a long-lived connection is important, please see the discussion of Streaming drivers above).

In some environments, however, a proxy server might interrupt these connections. This might be, for example, because:

- the proxy server wishes to "virus check" all content it receives on behalf of its clients. In this case the proxy server may buffer the streaming response in its entirety before "checking" it and delivering it to the client. Since the response is, essentially, unending, the proxy will never deliver any such content to the client and the client will be forced to reset and try again. If the client has been configured to try other drivers, it will eventually fall back to a long-polling driver (which should work perfectly well in such an environment).
- some companies may limit the size of responses which the proxy server will handle. In this case, the proxy server will forcefully close the connection. As a result, a Universal Messaging JavaScript client will automatically attempt to re-initialize the session, and gracefully continue. This should not affect usability too much (though clients will experience an unnecessary disconnect/reconnect cycle).
- some companies may limit the time for which a connection can remain open through the proxy. Again, a Universal Messaging JavaScript client will automatically work around this as above.

It is strongly recommended that you use SSL-encrypted HTTP for your Universal Messaging applications. Many proxy servers will allow SSL-encrypted HTTP traffic to pass unhindered. This will ensure that the greatest number of clients can use an efficient streaming driver.

Reverse Proxy Servers and Load Balancers

If your infrastructure includes a reverse proxy server or a load balancer, it is important to ensure that "stickiness" is enabled if you are using this infrastructure to front more than one Universal Messaging realm server. That is, requests from an existing client should always be made to the same back end Universal Messaging realm server.

Most load balancers offer stickiness based either on client IP address or using HTTP cookies. Client IP based stickiness can work well (though be aware that some clients may be making requests from different IP addresses, if, for instance, they are behind a bank of standard proxy servers in their own environment). Cookie-based stickiness will work well for most drivers (though note that some drivers, notably the "XDR" drivers which are based on the XDomainRequest object from Microsoft, do not support cookies - please see the JavaScript API Documentation for Drivers for more information).

Choosing Appropriate Drivers for Your Environment

While the default set of drivers work well in a simple environment where browsers connect directly to the server without intermediate infrastructure such as ill-configured proxy servers or overly-aggressive antivirus products, in some cases you may wish to customise the driver set to minimise issues with clients behind such infrastructure.

In summary, to minimise rare but lengthy delays where present-day client infrastructure interferes with session initialization, use the following set:

- XHR_STREAMING_CORS
- XDR_STREAMING
- IFRAME_STREAMING_POSTMESSAGE
- EVENTSOURCE_STREAMING_POSTMESSAGE
- XDR_LONGPOLL
- XHR_LONGPOLL_CORS
- XHR_LONGPOLL_POSTMESSAGE
- JSONP_LONGPOLL

AND use HTTPS with SSL certificates for the servers on which you deploy your HTML/JS and for all the UM servers.

More Details

Unless configured otherwise, Universal Messaging JavaScript clients will attempt to use the following drivers, in decreasing order of preference:

- WEBSOCKET: Streaming driver for browsers supporting HTML5 Web Sockets.
- XHR_STREAMING_CORS: Streaming driver for browsers supporting XMLHTTPRequest with CORS (Cross-Origin Resource Sharing). Intended for Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Internet Explorer 10+.
- XDR_STREAMING: Streaming driver for browsers supporting XDomainRequest (Internet Explorer 8+). Intended for Internet Explorer 8 and Internet Explorer 9. Note that XDomainRequest and hence the XDR_STREAMING driver can not send client cookies to the server.

- IFRAME_STREAMING_POSTMESSAGE: Streaming driver for browsers supporting the cross-window postMessage API (per https://developer.mozilla.org/en/DOM/window.postMessage). Intended for Chrome, Firefox, Safari, Internet Explorer 8+ and Microsoft Edge.
- EVENTSOURCE_STREAMING_POSTMESSAGE: Streaming driver for browsers supporting both Server-Sent-Events and the cross-window postMessage API.
- XDR_LONGPOLL: Longpoll driver for browsers supporting XDomainRequest (Internet Explorer 8+). Intended for Internet Explorer 8 and Internet Explorer 9. Note that XDomainRequest and hence the XDR_STREAMING driver can not send client cookies to the server.
- XHR_LONGPOLL_CORS: Longpoll driver for browsers supporting XMLHTTPRequest with CORS (Cross-Origin Resource Sharing). Intended for Chrome, Firefox, Safari, Internet Explorer 10+ and Microsoft Edge.
- XHR_LONGPOLL_POSTMESSAGE: Longpoll driver for browsers supporting the cross-window postMessage API. Intended for Chrome, Firefox, Safari, Internet Explorer 8+ and Microsoft Edge.
- NOXD_IFRAME_STREAMING: Legacy non-cross domain streaming driver for older clients requiring streaming from the server that serves the application itself. Intended for Chrome, Firefox, Safari, Internet Explorer 6+ and Microsoft Edge.
- JSONP_LONGPOLL: Longpoll driver for older browsers relying on DOM manipulation only (no XHR or similar required). Intended for Chrome, Firefox, Safari, Internet Explorer 6+ and Microsoft Edge.

The vast majority of clients settle on one of the first three streaming drivers.

As outlined in the API documentation, the developer can override the driver set and preference order. This is rarely recommended, however, unless a significant proportion of clients are located behind infrastructure which interrupt communication based on the typically preferred drivers. We shall explain how such interruptions can manifest themselves for each of these driver types.

Firstly, a little more detail on how driver failover works for the JavaScript API.

We will first look at how a client communicates with a single UM server:

A client browser first checks whether it supports the underlying technologies on which the current driver is based. If it does not, then it removes the driver from its list of possible drivers and will never attempt to use it again.

If the browser successfully initializes a session with the server, then it will always try to use the same driver thereafter, and will NOT fail over to a different driver (unless the user reloads the application).

If a browser does support the driver, then the driver gets 3 consecutive attempts to initialize a session. The first attempt happens immediately. The second connection attempt has a delay of 1 second. The third attempt has a delay of 2 seconds. This is to avoid compounding problems on a server that may be undergoing maintenance or other load issues (this is of particular importance if an environment supports many users, or has been configured to use only a single realm server).

After the third failure, if the client's session object has been configured with more than one driver (or if it is using the default set), then it will switch to the next driver, and will once again immediately try to connect and initialize a session. Such drivers' second and third attempts are subject to the same introduced delays as described above.

Next, let us look at how a client communicates with a cluster of UM servers:

When attempting to initialize a session with a server within a cluster, then a client will go through all drivers until one works, as described above. A side effect of this is that if the first server with which the client attempts to communicate is unresponsive, then one can expect a delay of at least (number_of_supported_drivers * 3 seconds) plus any underlying request timeouts before the client switches to the next UM server. In a worst-case scenario this could lead to a delay of 20 seconds or more, which is far from desirable.

This delay would be this considerable only when the first UM server the client attempted to use was unavailable. Had the first UM server been available, and subsequently become unavailable (thus disconnecting the browser), the browser will switch to a second realm server considerably more quickly, because:

- If the browser gets a confirmed session to server X, then - as explained earlier - it will always try to use that driver thereafter.

- If, having had a confirmed session to server X it gets disconnected from server X, then it will continue retrying to connect to server X with the same driver for a maximum of 5 consecutive failed attempts (any successful connection will reset the failure count to 0). If the 5th attempt fails, the browser will consider this server to be unavailable, and will switch to the next server, re-enable *all* drivers, and start cycling through them again (giving each one 3 chances to connect as usual). On the assumption that the second realm server is indeed available, a client would, at this point, reconnect immediately.

So one key here is to avoid including realms servers in a client's session configuration object if they are known to be not available pre-session-initialization. This can be done by dynamically generating session configuration objects based upon the availability of back-end servers (though this is beyond the scope of the UM API).

An alternative approach to avoid such a delay would be to lower the number of potential drivers a browser could attempt - but this will doubtlessly lead to fewer clients being able to connect, so is not recommended.

Finally, let's look at the individual drivers themselves. All notes below are for traffic over HTTP (not HTTPS):

- WEBSOCKET Streaming driver for browsers supporting HTML5 Web Sockets.

  In a typical production deployment, one would expect the majority of external client browsers to not be able to communicate using WebSockets. There are several reasons for this:

  - Firstly, a significant proportion of end users are using web browsers that do not yet support the WebSocket protocol (e.g. around 30% of users are on Internet Explorer 9 or earlier, with no WebSocket support).

- Secondly, users behind corporate proxy servers will, at present, find that the proxy server will in all likelihood not support the WebSocket protocol, even if their browser does; this is expected to change, but it will take years, not months.
- Thirdly, many companies wish to deploy their UM servers behind standard reverse proxy servers or load balancers; again, WebSocket support will appear in such products, but it is simply not there today.
- Fourthly, a client-side antivirus product can interfere with the browser's receipt of real-time data sent down a WebSocket response. This is common, for example, for clients using Avast's WebShield functionality - these clients usually have to be configured to use a long polling driver unless they configure their antivirus product to ignore traffic from specific hosts. The issue is that the antivirus product attempts to buffer the entire response so that it can process it for viruses and the like before giving it to the client. Avast upgraded their product to better handle WebSocket last year, and as part of that process they have whitelisted a number of well-known sites that use WebSocket, but may be still being a little overzealous with WebSocket connections to hosts they don't know.

These issues in combination make it more likely than not that a corporate web user will be unable to use WebSockets.

That said, if a browser doesn't support WebSocket, it will fail over to the next driver immediately. On the other hand, if the browser does support WebSockets, but an intermediate proxy or reverse proxy server doesn't, then the WebSocket HTTP handshake would, in some circumstances, result in the proxy immediately returning an error response to the browser. Under these conditions, the browser will retry the WebSocket driver a maximum of three times (incurring the "3 second" delay as per the "back-off" rules described above) before failing over to the next driver. The worse case scenario is a client using a proxy that simply ignores the WebSocket HTTP Protocol Upgrade handshake; here you are at the mercy of the client timing out, as the UM server would never receive the WebSocket upgrade request. Although you can set these timeouts to whatever values you wish, the default values are set to support clients on slow connections - lowering them will cause clients on slow connections to frequently fail to initialize sessions.

For client environments that support it, WebSocket is an excellent choice of driver. Whether you wish to risk a 3-second delay (or, in the rare cases described above, much higher) for those client environments that don't support it is down to your distribution of customers. If you are building applications targeting customers whose infrastructure you understand, then it is worth using the WebSocket driver. If the audience is more generally distributed, then a good proportion of the clients that are unable to use WebSockets will incur a session initialization delay. You may therefore wish to exclude the WEBSOCKET driver from your application's session configuration.

- XHR_STREAMING_CORS Streaming driver for browsers supporting XMLHTTPRequest with CORS (Cross-Origin Resource Sharing). Intended for Chrome, Firefox, Safari, Internet Explorer 10+.

  Unlike WebSocket, this driver does not rely on anything other than standard HTTP requests and a streaming HTTP response. If the client supports the driver, the only issues here are intermediate infrastructure components interfering with the real-time stream of data sent from the UM server to the client:

  - An intermediate proxy server may choose to buffer the streamed response rather than streaming it directly to the client in real-time. Some have a habit of buffering all un-encrypted traffic for virus-scanning purposes, thus preventing clients from using streaming drivers of any kind. This is not very common, but it does happen.
  - More common is the interruption of a streamed response by client-side antivirus products like Avast's WebShield (see above, in the WebSocket driver discussion). In these cases, the browser would not receive the session initialization response (as it is being buffered by the antivirus product), and would eventually time out. This is much slower than failing immediately.

- XDR_STREAMING Streaming driver for browsers supporting XDomainRequest (Internet Explorer 8+). Intended for Internet Explorer 8 and Internet Explorer 9. Note that XDomainRequest and hence the XDR_STREAMING driver can not send client cookies to the server.

  Although Internet Explorer 8 and Internet Explorer 9 support the XHR object, they cannot use the XHR object to make cross-domain requests. This driver therefore instead uses a Microsoft-specific object called the XDomainRequest object which can support streaming responses to fully cross-domain requests.

  As mentioned in its description, this driver can not send client cookies to the server. This is because Microsoft chose to prevent cookies being sent via the XDR object. As a result, any intermediate DMZ infrastructure (such as a load balancer) that relies on client-side HTTP cookies to maintain, for example, load balancer server "stickiness", will be unable to maintain stickiness (since no cookies that the load balancer might set will appear in subsequent client requests when using an XDR based driver). If the load-balancer is fronting more than one UM server, then this setup can result in load balancers sending a client's post-session-initialization requests, such as a subscription request, to a random back-end UM server rather than to the server with which a client had initialized a (load-balancer-proxied) session. This will cause a client to reinitialize their session, and repeat ad infinitum. In these cases, if load balancers are indeed using cookie-based stickiness, then you have two options: either explicitly configure JavaScript sessions with a set of drivers that exclude all XDR variants, or change the load balancer configuration to use client IP-based stickiness instead of cookie-based stickiness instead

- IFRAME_STREAMING_POSTMESSAGE Streaming driver for browsers supporting the cross-window postMessage API (per https://developer.mozilla.org/en/DOM/window.postMessage). Intended for Chrome, Firefox, Safari, Internet Explorer 8+ and Microsoft Edge.

  In all likelihood, a client browser will settle on one of the three drivers discussed above before it would fail over to this driver. This driver is really only of use in environments where, for some reason, a browser is unable to create an instance of an XHR object. This used to be the the case in some older versions of Internet Explorer if, for example, a Windows Administrator's policy prevented Internet Explorer from being able to invoke ActiveX objects (such as the XMLHTTPRequest object). In modern versions of Internet Explorer, however, XMLHTTPRequest is a native non-ActiveX object so this is less of an issue.

  The downsides of this driver are the same as those for the XHR_STREAMING_CORS driver: proxy and antivirus product interference.

- EVENTSOURCE_STREAMING_POSTMESSAGE Streaming driver for browsers supporting both Server-Sent-Events and the cross-window postMessage API.

  This driver is really only useful to a small subset of modern browsers such as Opera. It does not rely on any unusual HTTP behaviour, and is therefore only subject to the same negatives as the other streaming drivers: proxy and antivirus product interference.

- XDR_LONGPOLL Longpoll driver for browsers supporting XDomainRequest (Internet Explorer 8+). Intended for Internet Explorer 8 and Internet Explorer 9. Note that XDomainRequest and hence the XDR_STREAMING driver can not send client cookies to the server.

  This is the most efficient long-polling driver for Internet Explorer 8 and Internet Explorer 9. Its downside is the lack of cookie support. This is only an issue if dealing with certain load balancer configurations (see discussion of XDR_STREAMING).

- XHR_LONGPOLL_CORS Longpoll driver for browsers supporting XMLHTTPRequest with CORS (Cross-Origin Resource Sharing). Intended for Chrome, Firefox, Safari, Internet Explorer 10+ and Microsoft Edge.

  This is the most efficient long-polling driver for non-Internet Explorer browsers and for Internet Explorer 10+.

  As with all long polling drivers (including XHR_LONGPOLL_POSTMESSAGE and JSONP_LONGPOLL discussed below), the browser will maintain an open connection to the server until data is available for transport to the browser. As soon as data arrives for delivery to the browser, the server will wait for a short period of time (known as the server-side "Long Poll Active Delay" timeout, which defaults to 100ms) before sending the data to the browser, and closing the connection. The browser will then immediately make a new long-poll connection to the server, in preparation for more data. Since the server closes the connection immediately after sending any data, any intermediary proxy servers or antivirus products will not be buffering the response in the same was as they might buffer a streaming response, but will instead immediately relay the response to the browser.

  Note that if no data is available to be sent to the browser (if, say, the client is subscribed to a channel which rarely contains events) then the long-poll connection will remain open for a limited period before being closed by the server (at which point the browser will automatically create a new long-poll request and wait for data once again). The length of time for which such "quiet" connections stay open is defined by the "Long Poll Idle Delay" value, which can be set using Enterprise Manager (see the JavaScript tab for the relevant Interface). It is important that this value is lower than the value of any intermediary proxy servers' own timeouts (which are often 60 seconds, but sometimes as low as 30 seconds). A suitable production value for "Long Poll Idle Delay" might therefore be 25000ms (25 seconds).

  This driver, like all long polling drivers, is good for browsers that want low-latency but infrequent updates. Long polling drivers are not a good choice for browsers that receive a continual stream of many messages per second, however (as the browser may end up making several requests per second to the server, depending upon the "Long Poll Active Delay" value). In such cases it would be prudent to increase the "Long Poll Active Delay" value significantly from its default value of 100ms, to perhaps 1000ms or more (while acknowledging that the browsers using a long polling driver in such a scenario would no longer be receiving "near-real-time" data). For browsers that are instead subscribed to channels (or members of datagroups) that have relatively infrequent updates, the "Long Poll Active Delay" can potentially be lowered to 0, resulting in real-time delivery despite the use of a long-polling driver.

- XHR_LONGPOLL_POSTMESSAGE Longpoll driver for browsers supporting the cross-window postMessage API. Intended for Chrome, Firefox, Safari, Internet Explorer 8+ and Microsoft Edge.

  This is the most efficient long-polling driver for non-Internet Explorer browsers that do not support CORS, and is a cookie-supporting alternative to XDR_LONGPOLL for Internet Explorer 8 and Internet Explorer 9. See the XHR_LONGPOLL_CORS discussion for details on how long polling functions.

- NOXD_IFRAME_STREAMING Legacy non-cross domain streaming driver for older clients requiring streaming from the realm that serves the application itself. Intended for Chrome, Firefox, Safari, Internet Explorer 6+ and Microsoft Edge.

  This is a streaming driver that is subject to the same issues as the other streaming drivers: proxy and antivirus product interference.

  In addition, unlike all the other drivers, which are fully cross-domain (allowing communication to a UM server on a different domain to the server that serves the applications HTML/JS), this driver only permits communication back to the same server that served the HTML/JS. This implies that the application's HTML/JS must be served from a file plugin on the UM server in question.

- JSONP_LONGPOLL Longpoll driver for older browsers relying on DOM manipulation only. Browser will show "busy indicator/throbber" when in use. Intended for Chrome, Firefox, Safari, Internet Explorer 6+ and Microsoft Edge.

  This is the least efficient long-poll driver, but the most likely one to work in all circumstances. See the XHR_LONGPOLL_CORS discussion for details on how long polling functions.

Dealing with Intermediate Infrastructure Issues

As you can see, proxy servers are a particular bane for streaming HTTP traffic. Happily, practically all of these issues can be mitigated simply by using HTTPS instead of HTTP. In these cases, a proxy server will blindly proxy the HTTPS traffic without attempting to understand or modify the HTTP conversation contained within, and without interfering with it. If, therefore, you deploy your application on an HTTPS server, and install SSL certificates on your UM servers, you should be able to use the default set of drivers with minimal problems.

We would also point out that in our experience all production deployments at our customers' sites are secured with HTTPS (SSL or TLS). Besides helping browsers behind certain proxy servers maintain persistent connections that would otherwise be interrupted by naively configured proxy servers (perhaps buffering for virus checking), or by misconfigured or old proxies (that cannot understand WebSocket Protocol Upgrade handshakes), HTTPS offers clear security benefits and improves usability.

Working around client-side antivirus products is a little more difficult, unfortunately. One option might be to offer a link to a "long-poll-only" version of the application, and to show this link on screen only when a client is attempting to initialize a session for the first time. For clients which successfully initialize a session using a streaming driver, the link will be replaced with actual application content in no time, but for clients having problems, the ability to click a "click here if you're having problems connecting" is probably sufficient.

Other "workarounds", such as remembering the driver that worked for a browser as a value within a persistent cookie (so that it can be reused automatically thereafter), are risky; a client can be a laptop which subsequently connects via a proxy with a completely different configuration, for example. Such approaches are best not relied upon.

It is a shame to abandon WebSocket given its clear benefits, but in a production deployment we would, for the sake of simplicity, suggest the following as a useful custom set of drivers:

- XHR_STREAMING_CORS
- XDR_STREAMING
- IFRAME_STREAMING_POSTMESSAGE
- EVENTSOURCE_STREAMING_POSTMESSAGE
- XDR_LONGPOLL
- XHR_LONGPOLL_CORS
- XHR_LONGPOLL_POSTMESSAGE
- JSONP_LONGPOLL

This basically leaves out WEBSOCKET because of the risk of a prolonged "failure" time for a minority of clients behind incompatible proxies, and also leaves out the "legacy" NOXD_IFRAME_STREAMING driver since this is the only non-cross-domain driver, and it is likely that any production application's HTML/JS would benefit from being deployed on a static WebServer rather than on a file plugin within the messaging server.

If your DMZ infrastructure includes load balancer components that rely on HTTP cookies for "stickiness", then the XDR-based drivers should not be used (see the discussion of XDR_STREAMING above). In this case, the set of custom drivers would be best reduced to:

- XHR_STREAMING_CORS
- IFRAME_STREAMING_POSTMESSAGE
- EVENTSOURCE_STREAMING_POSTMESSAGE
- XHR_LONGPOLL_CORS
- XHR_LONGPOLL_POSTMESSAGE
- JSONP_LONGPOLL

What Drivers Should I Use?

This example can help you choose a suitable configuration:

Are your application's clients typically behind proxy servers?	Yes
Is your Universal Messaging server behind a Load Balancer?	No, we do not use a Load Balancer
Is your Universal Messaging server behind a Reverse Proxy Server?	No, we do not use a Reverse Proxy Server
Where is your application's HTML/JS served from?	It is served from a different web server (e.g. Apache)

Suggested Session Driver Configuration:

```
var session = Nirvana.createSession({
    // The WEBSOCKET driver is best not used if clients are behind
    // traditional proxy servers.
    // NOXD_IFRAME_STREAMING excluded as it won't work when
    // hosting application HTML on a 3rd-party webserver.
    drivers : [
        Nirvana.Driver.XHR_STREAMING_CORS,
        Nirvana.Driver.XDR_STREAMING,
        Nirvana.Driver.IFRAME_STREAMING_POSTMESSAGE,
        Nirvana.Driver.EVENTSOURCE_STREAMING_POSTMESSAGE,
        Nirvana.Driver.XDR_LONGPOLL,
        Nirvana.Driver.XHR_LONGPOLL_CORS,
        Nirvana.Driver.XHR_LONGPOLL_POSTMESSAGE,
        Nirvana.Driver.JSONP_LONGPOLL
    ]
});
```

Corresponding Server-Side Configuration:

Remember to include the hostname(s) of all servers hosting your application HTML in the CORS Allowed Origins field.

Ensure that you have configured a file plugin on the Universal Messaging interface to which JavaScript clients will connect, and that you have configured it to serve /lib/js/crossDomainProxy.html correctly.

Ensure that the Long Poll Idle Delay value is set to be lower than your Load Balancer or Reverse Proxy server's timeout value. This setting is in ms, and a good value is usually around 25000.

Ensure that the Long Poll Idle Delay value is set to be lower than any intermediate proxy server's timeout value. Many proxy servers have timeouts as low as 30 seconds (though an administrator may have lowered this, of course). In the Universal Messaging server, this setting is in ms, and a pragmatic value is usually around 25000. You may need to lower it further if external clients behind remote proxy servers fail to maintain sessions when using a longpolling driver.

WebSocket Delivery Mode

An Introduction to WebSockets

WebSocket is an emerging HTML5 protocol which provides full-duplex communication over a TCP socket. The WebSocket API is currently being developed by W3C and the WebSocket Protocol standardized by IETF.

WebSocket is supported by all major browser vendors. It is currently enabled in Google Chrome (6+), Mozilla Firefox (6+), Apple Safari (5+), Microsoft Internet Explorer (HTML5 Labs only) and Microsoft Edge.

WebSockets serve as an effective low-latency alternative to comet based solutions such as "XMLHTTPRequest LongPoll" on page 268 or "Forever IFrame" on page 269. It provides browser-based communication which does not rely on opening multiple HTTP communications, unlike the aforementioned comet based techniques.

WebSockets in Universal Messaging

Universal Messaging supports WebSocket browser-based communication through our JavaScript API. In addition to WebSockets our JavaScript API offers both "LongPolling" on page 268 and "Forever IFrame" on page 269 communication modes. In situations where it is possible to use WebSocket however there are significant performance gains to be had.

In some cases it is desirable to communicate in a secure manner using WebSocket secure (wss) instead of unencrypted WebSocket (ws). To do this using our JavaScript API, the following snippet is used:

```
var mySession = Nirvana.createSession({
realms : [ "https://host.softwareag.com:443" ],
});
```

Sometimes unsecured WebSocket connections cannot be established because of intermediary network infrastructure such as proxies or firewalls. Some anti-virus software may also block or restrict the creation of a WebSocket on some ports. These problems can often be overcome by using WebSocket over an SSL connection.

WebSocket interaction with Proxy Servers

As the adoption of WebSocket becomes more widespread an increasing number of proxy vendors are supporting communication over WebSocket. There still remain cases however where WebSocket connections cannot be established because of a proxy server or firewall which blocks this connection.

The reasons for this are twofold. Some proxy servers have not been updated to handle the protocol, treating requests to establish the connection as normal HTTP traffic. WebSocket requests are also known not to conform with the HTTP 1.0 spec, causing some proxies to reject them.

Please see JavaScript API Documentation for Drivers for more details of WebSocket and other drivers.

WebSocket communication over a Forward Proxy

A Forward Proxy (often just referred to as a proxy) acts as an intermediary between a client and a browser. If the proxy is explicitly configured it is usually possible to establish a WebSocket connection to a server without the need for further configuration.

This generally works because in cases where a browser is configured to use an explicit proxy server it will issue an `HTTP CONNECT` request to the proxy when establishing the WebSocket connection. The Connect method enables the proxy to act as a tunnel between the client and server.

There are two cases where this method may not work. In some situations the proxy may be configured to restrict use of the `HTTP CONNECT` method. In other situations the request may travel through a proxy which is not explicitly configured (a transparent proxy). As the browser has no knowledge of the proxy, it will not issue the `HTTP CONNECT` request.

One approach that will likely grant connectivity in both of these situations would be to use the secure version of the protocol. As the data sent is encrypted when sent the proxy server it is unable to manipulate it in any way. In these situations it will often forward the request from the client to the server intact and a connection can be established.

An example on configuring a forward proxy which works with WebSocket is available in the section "WebSocket over a Forward Proxy" on page 265.

WebSocket communication over a Reverse Proxy

A reverse proxy is a server which appears to clients to be an ordinary server. The client issues requests directly to this proxy. Based on a set of rules the proxy server will then forward this request to one or more origin servers which handle the request. The response is sent back through the proxy and it appears to the client that this response originated directly from the reverse proxy itself.

WebSocket communication through reverse proxy servers using the application layer (HTTP in this case) may work in some but not all scenarios. The proxy server must provide support for WebSockets (HAProxy for example, does). Furthermore the client wishing to communicate using WebSockets must use a version greater than 8 of the protocol.

WebSocket communication through a reverse proxy using clients which use a previous version of the protocol is generally not possible. This is because of a recognised flaw corrected after version 8 of the protocol. This flaw manifests itself because older versions of the protocol included 8 bytes of key data after the header during a connection upgrade request. This data was not advertised in the Content-Length header. Because of this most application layer proxies will discard this data.

Reverse proxies which support forwarding at the Transport layer (layer 4 of the OSI model) can still be used with older versions of the WebSocket protocol. As layer 4 proxies do not inspect the contents of a HTTP requests header (whereas application layer proxies do) the key data is not removed when traversing over them.

When traversing reverse proxies which forward at the transport layer both secure and non-secure communication modes are likely to work. When specifying a WebSocket port to connect to when using a reverse proxy, the client must use the port which the reverse proxy is listening to connections on. It should not attempt to connect to the port which the Universal Messaging realm interface is listening on.

Examples of configured reverse proxies which work with WebSocket are available in the section "WebSocket over a Reverse Proxy" on page 266.

WebSocket over a Forward Proxy

WebSocket communication can take successfully take place in the presence of forward proxies, providing the client and proxy server have been configured properly to deal with it. This page explains how to configure a Universal Messaging JavaScript client and Apache serving as a forward proxy to permit WebSocket use.

Configuration for an Explicit Forward Proxy

An explicit forward proxy is a forward proxy which the client is configured to use. The client is aware of the presence of this proxy. In these situations it is easier for the client to establish a WebSocket connection with the server for reasons outlined in the section "WebSocket Delivery Mode" on page 262.

Warning:
Before detailing how to configure Apache as a forward proxy we warn you that this can be a dangerous thing to do. Before enabling Apache to act as a forward proxy you must secure your server correctly. Failing to do so will provide malicious entities with an open proxy server which are dangerous to both your own network and the rest of the internet.

To proxy requests from your server enable the `ProxyRequests` directive, located in `mod_proxy`. An example configuration file configured as a forward proxy would be:

```
# Example Apache forward proxy configuration
...
Listen 80
# Ensure Proxy Module is Loaded
LoadModule proxy_module path/to/mod_proxy
# Turn on forward proxying
# DO NOT DO THIS UNLESS YOUR FORWARD PROXY IS CORRECTLY SECURED
ProxyRequests On
ProxyVia On
AllowCONNECT 9000 # Allow HTTP CONNECT on the nirvana realm port
<Proxy *>
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1 # Restrict to localhost only
</Proxy>
...
```

A JavaScript client may then connect using this forward proxy by initialising a session using the following options:

```
NirvanaSession.start({
    ...
    webSocket : true,
    webSocketPort : 9000 // Port of the Universal Messaging interface
});
```

If the proxy is configured to restrict use of the `HTTP CONNECT` method then the steps above may fail even if the proxy is explicitly declared to the client browser. In these cases it is possible to take the same steps as detailed below for transparent proxy servers to establish a WebSocket connection.

Configuration for a Transparent Forward Proxy

A transparent forward proxy is an invisible proxy which sits between the client and server. In these cases as the client browser does not know about the presence of this proxy it will not send a `HTTP CONNECT` request to the proxy when establishing a WebSocket connection. Establishing the connection will likely fail using the configuration above.

Communicating with the server using SSL will alleviate this problem. Transparent proxies will usually by default forward SSL traffic. This will allow us to establish a WebSocket connection.

To do this we must configure apache to allow the `HTTP CONNECT` header to be sent to the secure nirvana interface. In this case the example is exactly the same as above, except with the line `AllowCONNECT 9443` where 9443 is the port of the secure realm interface.

The client can then be configured to start a WebSocket session as follows:

```
NirvanaSession.start({
    ...
    webSocket : true,
    webSocketPort : 9443, // Port of the Universal Messaging secure interface
    secure : true
});
```

To maximise the chance of establishing a successful WebSocket connection we recommend using this secure method of transport over the alternative.

WebSocket over a Reverse Proxy

WebSocket communication can take place over any reverse proxy which is configured to perform forwarding at the transport layer. Some proxies are able to handle WebSocket communication from certain clients at the application layer. This page details example configurations for the open source proxy and load balancing software HAProxy.

Application Layer (HTTP) Proxy

Clients communicating using versions of the WebSocket protocol later than version 8 are able to negotiate some reverse proxies which use application layer forwarding. HAProxy is one such vendor able to handle WebSocket communication in this manner.

An example configuration file for HAProxy is as follows:

```
# Example HAProxy Configuration file
# Here we forward all requests on port 443 to our nirvana server
# listening on port 9443
# backend defines the nirvana server to forward to. We declare
# two backends, one for # serving the http page and another for
# websocket communication. Note that these could be two different
# ports (80 for serving the web content and 9443 for the secure
```

```
# nirvana websocket connection)
backend nirvana_www
    balance roundrobin
    option forwardfor
    timeout connect 10s
    timeout server 30s
    server nirvana1 nirvanahost:9443 weight 1 maxconn 1024 check
backend nirvana_socket
    balance roundrobin
    option forwardfor
    timeout connect 10s
    timeout server 30s
    server nirvana1 nirvanahost:9443 weight 1 maxconn 1024 check
frontend https_proxy
    bind *:443
    timeout client 30s
    default_backend nirvana_www
    acl is_websocket hdr(Upgrade) -i WebSocket
    acl is_websocket hdr_beg(Host) -i ws
    use_backend nirvana_socket if is_websocket
```

Transport Layer (TCP) Proxy

As forwarding occurs at the transport layer it can only be performed based on the port of the received packet. To perform forwarding based on URL we would need access to the HTTP object at the application layer.

The configuration file for HAProxy is as follows:

```
# Example HAProxy Configuration file
# Here we forward all incoming requests on port 443 to our nirvana
# server which has an nhps interface listening on port 9443
# backend defines the nirvana server to forward to
backend nirvana
    mode tcp
    timeout connect 10s
    timeout server 30s
    balance roundrobin
    server nirvana1 nirvanahost:9443 weight 1 maxconn 1024
# nirvanahost translates to an ip address
# frontend defines the interfaces for the reverse proxy to listen on
frontend https_proxy
    bind *:443
    mode tcp
    timeout client 30s
    default_backend nirvana
```

Configuring the Client

The client JavaScript session should be configured as follows:

```
NirvanaSession.start({
    ...
    webSocket : true,
    webSocketPort : 443,
    secure : true
```

The client can then connect by visiting the page `https://proxyhost:443`

It is possible to communicate using WebSockets over a reverse proxy without using a secure connection. This can be achieved similar to the example above except modifying the port (to use a non-secure port) and changing the session options to this port and setting the `secure` flag to `false`. It is however recommended that to maximise the success of establishing a WebSocket connection a secure communication method is chosen.

Comet Streaming Delivery Mode

Comet Streaming drivers in JavaScript have been implemented in several ways, including XHR with CORS, XDR, and "Forever IFrames".

Comet Forever IFrames in Universal Messaging

An IFrame is an inline frame within a web page which contains an inline document. The term 'Forever IFrame' is used to denote an inline frame which is implicitly declared as infinitely long. This property of such frames allow the server to continually populate it with data. As browsers render and execute scripts on a page incrementally, the pushed data can be manipulated by the client as it passes into the frame.

The Forever IFrame delivery mode is supported by Universal Messaging's JavaScript API and can be used in all browsers.

In addition to Streaming, "WebSocket" on page 262 and "LongPolling" on page 268 delivery modes are supported.

Please see the JavaScript API Documentation for Drivers for more details.

Comet LongPolling Delivery Mode

An Introduction to Comet LongPolling

LongPolling is a variation of the traditional polling technique. In traditional polling, a client sends requests on a regular basis to the server attempting to pull any new data available on the server. If there are no events on the server an empty response is returned and after a specified delay the client sends a new request.

LongPolling sends requests to the server in much the same way as traditional polling. In a LongPoll implementation however if the server has no data to push it holds the request up until the point where new data is available or the request times out. Once the server sends a LongPoll response the client typically initiates a new request immediately.

As the server usually holds a LongPoll request at all times from the client. It is able to asynchronously push data to the client by providing it with a response.

Comet LongPolling in Universal Messaging

The LongPolling delivery mode is supported by Universal Messaging's JavaScript API and can be used by all browsers. In addition to LongPolling, “WebSocket” on page 262 and “Forever IFrame” on page 269 delivery modes are also offered by the API.

LongPolling is a desirable option when the implementation must support older browser versions, or also when requests must traverse proxy servers which may not be configurable by those developing the application. Other delivery modes often require newer browsers or unobtrusive end-to-end connections.

Longpolling drivers in Universal Messaging JavaScript include XHR with CORS, XHR with postMessage, XDR and JSONP. Please see JavaScript API Documentation for Drivers for more details.

Comet Forever IFrame Delivery Mode

An Introduction to Comet Forever IFrames

An IFrame is an inline frame within a web page which contains an inline document. The term 'Forever IFrame' is used to denote an inline frame which is implicitly declared as infinitely long. This property of such frames allow the server to continually populate it with data. As browsers render and execute scripts on a page incrementally, the pushed data can be manipulated by the client as it passes into the frame.

Comet Forever IFrames in Universal Messaging

The Forever IFrame delivery mode is supported by Universal Messaging's JavaScript API and can be used in all browsers. In addition to Forever IFrame, “WebSocket Delivery Mode” on page 262 and “Comet LongPolling Delivery Mode” on page 268 delivery modes are supported.

A Universal Messaging JavaScript client can communicate with a server using the Forever IFrame delivery mode by specifying the following configuration when starting a session:

```
NirvanaSession.start({
    ...
    protocolSelection : ["streamingcomet"]
});
```

Example: Implementing a Simple Pub/Sub Client

The Universal Messaging JavaScript API makes it easy to implement JavaScript Publish & Subscribe clients. These clients can communicate using Comet techniques (both streaming and long-polling), as well as using Web Sockets when supported by the client browser.

The code shown below is a fully functioning example of such a client, containing JavaScript connection, publishing and subscription logic and an HTML UI.

In some circumstances you may wish to serve your web application from another web server (e.g. Apache). Universal Messaging supports this also but due to security restrictions within browsers it requires that your application is organised differently (see "Serving From Another Webserver" on page 232 for related information).

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <meta http-equiv='X-UA-Compatible' content='IE=edge'>
    <title>Page Title</title>

    <script language="JavaScript" src="lib/nirvana.js"></script>
    <script>

       // this needs to be configured based on the server host you are executing this
 example against
        var hostName = "http://localhost:11000";
        // this needs to be configured based on the channel name
  var channelName = "/showcase/simplechatroom";

  var demoUsername = "anonymous";
         /*********************************************************************
        * As soon as the page loads, we should create our Universal Messaging session.
         * We use the  object Nirvana and define what
         * we want to happen once the session is initialised by implementing
         * a callback function, NirvanaSession.onInit:
         *********************************************************************/
         var session =  Nirvana.createSession({
                realms : [ hostName ],
                // this can be an array of realms
                debugLevel : 4, // 1-9 (1 = noisy, 8 = severe, 9 = default = off)
                sessionTimeoutMs : 10000,
                enableDataStreams : false,
                drivers : [ // an array of transport drivers in preferred order:
                    Nirvana.WEBSOCKET,
                    Nirvana.XHR_STREAMING_CORS,
                    Nirvana.XDR_STREAMING,
                    Nirvana.JSONP_LONGPOLL
                ]
            });
            session.start();
        function isLoaded() {

        /*********************************************************************
        * Now that our session has initialised, we can access an automatically
        * instatiated object named Nirvana, which provides access to classes
        * representing events and dictionaries.
        * In this demo, we shall subscribe to a channel and define what we
        * want to happen when certain activities occur (such as subscribing,
        * or receiving events). We do this by implementing callback logic,
        * either with anonymous functions, or if preferred, with named functions:
        *********************************************************************/

        //Obtain channel from the session that was created
        var demoChannel = session.getChannel(channelName);
        demoChannel.subscribe();
        var isSubscribed = demoChannel.isSubscribed();
        if(isSubscribed){
            window.status = "Subscribed to " + demoChannel;
        }
```

```
      demoChannel.on(Nirvana.Observe.DATA, demoEventHandler);
      demoChannel.onPublish = updateUserInputUI;



       onConnect = function() { window.status = "Connected"; }
       onDisconnect =
          function() { window.status = "Disconnected. Reconnecting..."; }

       /*******************************************************************
       * Now that we have defined all that should happen when our session is
       * up and running, let us *start* it.
       ********************************************************************/
      }

      function demoEventHandler(event) {

       /*******************************************************************
       * This method automatically gets invoked every time we receive an
       * event from the demoChannel (since this is the method we specified
       * as the demoChannel.onData event handler. Note that the
       * event object will be passed to this method as a parameter. We can
       * then get the event's data "dictionary", and read the value of any
       * of its keys. In this demo, we use this data to update a textarea.
       ********************************************************************/

       var dictionary = event.getDictionary();
       var newData =
           dictionary.get('publisher') + ": " + dictionary.get('message') + "\n"
       var oldData = document.getElementById("outputTextarea").value;
       document.getElementById("outputTextarea").value = newData + oldData;
      }

       function publishMessage() {

       /*******************************************************************
       * This method is an example of how to publish events to our channel.
       * We first get myChannel from an already created session,
       * then we create an event myEvent - empty Universal Messaging <Event>,
       * we create dictionary variable(myDict), and put the string
       * from the html input. Finally, we call the channel's publish method,
       * It is good practice to wrap code like this in try/catch blocks.
       ********************************************************************/

       if (document.getElementById("demoInput").value == "") return;
       try {

          var myChannel =  session.getChannel(channelName);
          var myEvent = Nirvana.createEvent();
          var myDict = myEvent.getDictionary();
          myDict.putString("message", document.getElementById("demoInput").value);
myDict.putString("publisher", demoUsername);
          myChannel.publish(myEvent);
       } catch (error) {
               alert("Error: " + error.message);
       }
       }


      function updateUserInputUI() {
```

```
        /***************************************************************
        * This method automatically gets invoked after we successfully
        * publish to testChannel (since this is the method we specified
        * as the handler for testChannel.onPublish.
        * A typical implementation of such a function would re-enable UI components
        * that might have been disabled while publishing took place.
        ***************************************************************/

        if(!initialiasedASession)
            alert("We did not get a session to Universal Messaging");
        document.getElementById("demoInput").value = "";
        window.status = "Published";
        }


    </script>
<title>Universal Messaging JavaScript Tutorial Application:
    Simple Chat Room</title>
</head>
<body onload="isLoaded()">
    <h1>Universal Messaging JavaScript Tutorial Application:
        Simple Chat Room</h1>

    <form onsubmit="publishMessage(); return false;">

      <h2>Input</h2>
      <input type="text" id="demoInput"/>
      <input type="submit" value="Publish">

      <h2>Output</h2>
      <textarea id="outputTextarea" rows="10" cols="70"></textarea>

    </form>

    </body>
</html>
```

Web Developer's Guide for Java

Web Developer's Guide for Java

This guide describes how to develop and deploy Java Web applications using Universal Messaging, and assumes you already have Universal Messaging installed.

Universal Messaging Web Client Development in Java

Universal Messaging Web Clients have access to the Universal Messaging Enterprise API for Java, which has been streamlined to provide our full messaging capability via a very small client library which is easily deployed as an applet or a Java Web Start application.

Please refer to the Universal Messaging Enterprise Java Development Guide for more information on Java Client Development.

Deploying Java Applications using Java Web Start

This guide describes the basic concepts for deploying feature rich Java applications using Java Web Start.

Java Web Start

Java Web Start enables applications to be deployed quickly and easily launched from a web server. Once launched using Web Start, an application can subsequently be directly launched using a desktop link on the client machine.

Basics

Typically, an application written in Java can be deployed quickly with a few simple steps. Java Web Start applications require all resources to be located within one or more jar files. Once you have packaged up your resources (classes, images etc.) into your jar file(s), you need to create a Java Network Launching Protocol (JNLP) file to be placed onto your web server. This file specifies all the properties required by your application, as well as any Web Start instructions required in order to launch the application.

New versions of your application can be easily deployed to your customer base automatically by updating the resource jar files and deploying them to your web server. Java Web Start applications will automatically check for new versions before launching local cached versions.

Applet Javascript Bridge Example

A Sample JavaScript-Applet Bridged Client

Using the Universal Messaging nApplet, it is easy to bridge communications between a JavaScript front end which delegates all Universal Messaging communication to an Applet.

The code shown below is a fully functioning example of such a client, containing an applet along with JavaScript code which communicates seamlessly with the applet. The applet implements all connection, publishing and subscription logic. All events that are delivered to the applet are called back into JavaScript asynchronously.

```
<?xml version="1.0" encoding="UTF-8"?>
<!doctype html PUBLIC "-// W3C// DTD XHTML 1.0 Transitional// EN"
 "http:// www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http:// www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<script language="JavaScript" src="lib/nirvana.js"></script>
<script>
var appName = "myTestApplication";
var channelName = "/tutorial/testchannel/";
var realm = "nhp://" + location.hostname + ":80";
var sessionName = "myExampleSession";
var username = "TestUser"; // note that in a real app,
 // this should be an authenticated username!
var connected = false;
```

```
var isLoadedCounter = 0;
var mySession = null;
var testChannel = null;
var isTimedOutConnectingCounter = 0;
function isLoaded() {
 /*********************************************************************
 * As soon as the page loads, we should create our Nirvana session.
 * This method is invoked by the <body> tag's "onload" event.
 * We initialise our session by passing in a realm address, a sessionid,
 * a username and a "prefix" for connection listener callback methods.
 * The last parameter is the connection listener "prefix". As we chose
 * to use the string "conHandlerCB" as the prefix, we must now implement
 * four methods to receive asynchronous notifications of our connection
 * status: conHandlerCBgotInitialConnection, conHandlerCBdisconnected,
 * conHandlerCBreconnected and conHandlerCBtryAgain.
 * Note that we could use any string as a prefix; we simply need to
 * name our four implemented methods accordingly.
 *********************************************************************/
 // the nJSCRIPT variable is set by the applet when it initially loads
 if ( typeof nJSCRIPT == "undefined" ) {
 if ( isLoadedCounter > 4) {
 window.status = "unable to initialise nirvana libraries";
 return;
 }
 else{
 setTimeout("isLoaded()", 4000);
 }
 }
 else if ( nJSCRIPT == true ) {
 window.status = "nApplet has been initialised.";
 // session is initialised with a realm address, a sessionid,
 // a username and a connection listener callback stub.
 mySession = new nSessionWithSubjectAndReconnectionHandler(realm,
     appName, username, "conHandlerCB");
 mySession.init();
 connectionTimeoutMonitor();
 }
}
function connectionTimeoutMonitor() {
 /*********************************************************************
 * This method is used to allow the web page to give up on attempting
 * to get an initial connection after a certain number of retries are
 * unsuccessful.
 *********************************************************************/
 isTimedOutConnectingCounter++;
 if ( mySession.isConnected() ) {
 return;
 }
 else if (isTimedOutConnectingCounter < 4) {
 setTimeout("connectionTimeoutMonitor()", 4000);
 }
 else if (isTimedOutConnectingCounter == 4) {
 timedOutConnecting = true;
 window.status = "Timed out connecting to Nirvana. ";
 document.getElementById('nirvana').stop();
 return;
 }
}
function conHandlerCBdisconnected() {
 /*********************************************************************
```

```
 * This method automatically gets invoked we get disconnected from our
 * Nirvana session. Note that this is because we specified
 * "conHandlerCB" as the prefix of the implicit "disconnected",
 * "reconnected","initialConnection" & "tryAgain" methods when we
 * created the session (see fourth parameter in the constructor for
 * nSessionWithSubjectAndReconnectionHandler).
 * A typical use for these method would be to re-enable UI components
 * which might have been disabled during the disconnected period.
 * See also the tryAgain method, which allows us to specify whether
 * the application should attempt to reconnect automatically after a
 * disconnect has occurred.
 ********************************************************************/
 connected = false;
 alert("Disconnected");
}
function conHandlerCBreconnected() {
 /*******************************************************************
 * This method automatically gets invoked we get reconnected to our
 * Nirvana session after a disconnect too place. Note that this is
 * because we specified "conHandlerCB" as the prefix of the implicit
 * "disconnected","reconnected","initialConnection" & "tryAgain"
 * methods when we created the session (see fourth parameter in the
 * nSessionWithSubjectAndReconnectionHandler constructor).
 * A typical use for these method would be to re-enable UI components
 * which might have been disabled during the disconnected period.
 * See also the tryAgain method, which allows us to specify whether
 * the application should attempt to reconnect automatically after a
 * disconnect has occured.
 ********************************************************************/
 connected = true;
 alert("Reconnected");
}
function conHandlerCBgotInitialConnection() {
 /*******************************************************************
 * This method automatically gets invoked we get the initial connection.
 * Note that this is because we specified "conHandlerCB" as the prefix
 * of the implicit "disconnected","reconnected","initialConnection" &
 * "tryAgain" methods when we created the session (see fourth parameter
 * in the nSessionWithSubjectAndReconnectionHandler constructor).
 * A typical use for these method would be to wait for confirmation
 * that the session has initialised before continuing with any other
 * processing.
 ********************************************************************/
 connected = true;
 window.status = "Connected to Nirvana.";
 setTimeout("handleNewConnection();", 500);
}
function conHandlerCBtryAgain() {
 /*******************************************************************
 * This method automatically gets invoked after each attempt to
 * reconnect to a disconnected Nirvana session. This allows the
 * developer to control whether or not continued attempts should be
 * made to reconnect. Note that this is because we specified
 * "conHandlerCB" as the prefix of the implicit "disconnected",
 * "reconnected","initialConnection" & "tryAgain" methods when we
 * created the session in our isLoaded() method.
 * This method allows us to specify whether the application should
 * attempt to reconnect automatically after a disconnect has occured.
 ********************************************************************/
 return true;
```

```
}
function handleNewConnection() {
 window.status = "Session Initialised";
 setupTestChannel();
}
function setupTestChannel() {
 /*****************************************************************
 * Here we create an nChannelAttributes object, setting its name to
 * that of the channel we wish to use. We then use our session to
 * a) find the channel, then b) subscribe to the channel.
 *****************************************************************/
 if (connected) {
 var channelAttribs = new nChannelAttributes();
 channelAttribs.setName(channelName);
 testChannel = mySession.findChannel( channelAttribs, "testChannelFoundCB" );
 // make sure we have a usable channel object
 if (testChannel == false) {
 if (mySession.isConnected()) {
 // Channel could not be found. Let us try again.
 // Maybe channel does not exist, or channel ACL is incorrect
 setTimeout("setupTestChannel()", 2000);
 } else {
 // waiting for disconnect
 setTimeout("setupTestChannel()", 4000);
 }
 } else {
 // Add subscriber to the channel object
 var startEID = 0;
 var evtHandler = "myTestChannelEventHandlerCB";
 testChannel.addSubscriberFromEID(evtHandler, startEID)
 }
 } else {
 // if we were disconnected when this method was called, try again
 setTimeout("getServerTime()", 4000);
 }
 window.status = "";
}
function myTestChannelEventHandlerCB(event) {
 /*****************************************************************
 * This method automatically gets invoked every time we receive an
 * event from the testChannel (since this is the method we specified
 * when we subscribed - see testChannelFoundCB method). Note that the
 * event object will be passed to this method as a parameter. We can
 * then get the event data (which is a byte[]), and/or its "dictionary"
 * which contains a set of key-value pairs. In this demo, we use the
 * dictionary keys "publisher" and "message", and update a textarea.
 *****************************************************************/
 var dictionary = event.getDictionary();
 var newData = dictionary.get('publisher') + ": " + dictionary.get('message') + "\n"
 var oldData = document.getElementById("outputTextarea").value;
 document.getElementById("outputTextarea").value = newData + oldData;
}
function publishMessage() {
 /*****************************************************************
 * This method is an example of how to publish events to our channel.
 * We first create an nConsumeEvent, and assign it an nEventProperties
 * object (which represents a data"dictionary" - essentially a hash of
 * key-value pairs). Finally, we publish our event to the channel.
 *****************************************************************/
 if (document.getElementById("demoInput").value == "") return;
```

```
 try {
 var evt = new nConsumeEvent();
 var dictionary = new nEventProperties();
 dictionary.put("publisher", username);
 dictionary.put("message", document.getElementById("demoInput").value);
 evt.setDictionary(dictionary);
 testChannel.publish(evt);
 } catch (error) {
 alert("Error: " + error.message);
 }
}
</script>
<title>Pub/Sub with Nirvana JavaScript</title>
</head>
<body onload="isLoaded()">
<h1>Nirvana : Pub/Sub with Java to JavaScript Bridge</h1>
<applet
 codebase = "/jars/"
 archive = "nClient.jar,nSigned.jar,nApplet.jar"
 code = "com.pcbsys.nirvana.client.jscript.NirvanaAppletThreaded.class"
 id = "nirvana"
 name = "nirvana"
 MAYSCRIPT
 width = "0"
 height = "0"
 hspace = "0"
 vspace = "0"
 align = "bottom"
>
</applet>
<form onsubmit="publishMessage(); return false;">
 <h2>Input</h2>
 <input type="text" id="demoInput"/>
 <input type="submit" value="Publish">
 <h2>Output</h2>
 <textarea id="outputTextarea" rows="10" cols="70"></textarea>
</form>
</body>
</html>
```

5 API Language Comparisons

Universal Messaging APIs for Enterprise, Web and Mobile applications are available in a range of programming languages. The following table provides an overview of each language's support for Universal Messaging features and communication protocols:

	Target Environments			Communication Protocols	Messaging Paradigms		Extended APIs	
	Enter-prise	Web	Mobile	Native or Comet	Pub/Sub	Msg. Queues	Admin	JMS
Java	✓	✓		Native	✓	✓	✓	✓
C# .NET	✓	✓		Native	✓	✓	✓	
C++	✓			Native	✓	✓	✓	
Python	✓			Native	✓	✓		
Excel VBA	✓			Native	✓			
JavaScript		✓		Native (via WebSocket) or Comet	✓	✓		
iPhone			✓	Native	✓	✓		
Android			✓	Native	✓	✓		