Universal Messaging Developer Guide

Version 9.7

October 2014

This document applies to Universal Messaging Version 9.7 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: NUM-DG-97-20141015

Table of Contents

Universal Messaging Client Development..... 15

Enterprise APIs..... 17
Overview of the Enterprise Client APIs..... 18
Enterprise Developer's Guide for Java..... 19
General Features..... 19
Create Session..... 19
Events..... 19
Channel Joins..... 20
Event Dictionaries..... 22
Google Protocol Buffers..... 22
Publish / Subscribe Using Channels/Topics..... 26
Creating a Universal Messaging Channel..... 27
Finding a Universal Messaging Channel..... 28
Publishing events to a Channel..... 28
Sending XML Dom Objects..... 30
Asynchronous Subscriber..... 30
Channel Iterator..... 31
Batched Subscribe..... 32
Batched Find..... 33
Durable channel consumers and named objects..... 33
Event Fragmentation on Channels..... 35
The Merge Engine and Event Deltas..... 35
Priority Messaging..... 37
Publish / Subscribe Using DataStreams and DataGroups..... 37
DataStreamListener..... 38
Creating and Deleting DataGroups..... 39
Managing DataGroup Membership..... 39
DataGroup Conflation Attributes..... 41
DataGroups Event Publishing..... 43
DataStream Event Publishing..... 44
Priority Messaging..... 44
Message Queues..... 45
Creating a Queue..... 45
Finding a Queue..... 46
Queue Publish..... 46
Asynchronous Queue Consuming..... 47
Synchronous Queue Consuming..... 48
Asynchronous Transactional Queue Consuming..... 49
Synchronous Transactional Queue Consuming..... 50
Queue Browsing / Peeking..... 51

Request Response....52
Event Fragmentation on Queues....53
Peer to Peer Services....56
Peer to Peer Event-based Clients....57
Peer to Peer Event-based Server Services....58
Peer to Peer Stream-based Clients....59
Peer to Peer Stream-based Server Services....60
Provider for JMS....61
JMSAdmin....61
Client SSL Configuration....63
Application Server Integration (JBoss)....64
JMS Message / Event Mapping....66
Fanout Engine....67
Code Examples....70
Pub/Sub Channels....72
Java Client: Channel Publisher....72
Java Client: Transactional Channel Publisher....73
Java Client: Asynchronous Channel Consumer....73
Java Client: Synchronous Channel Consumer....74
Java Client: Asynchronous Named Channel Consumer....74
Java Client: Synchronous Named Channel Consumer....75
Java Client: XML Channel Publisher....75
Java Client: Asynchronous XML Channel Consumer....75
Java Client: Event Delta Delivery....76
Java Client: Batching Server Calls....76
Java Client: Batching Subscribe Calls....76
Pub/Sub Datagroups....77
Java Client: DataStream Listener....77
Java Client: DataGroup Publishing with Conflation....77
Java Client: DataGroup Manager....77
Java Client: Delete DataGroup....78
Java Client: DataGroup Delta Delivery....78
Message Queues....78
Java Client: Queue Publisher....78
Java Client: Transactional Queue Publisher....79
Java Client: Asynchronous Queue Consumer....79
Java Client: Asynchronous Transactional Queue Consumer....79
Java Client: Synchronous Queue Consumer....80
Java Client: Synchronous Transactional Queue Consumer....80
Java Client: Peek events on a Queue....80
Java Client: Requester - Request/Response....81
Java Client: Responder - Request/Response....81
Peer to Peer....82
Java Client: An Event-based Peer to Peer Client....82
Java Client: An Event-based Peer to Peer Server Service....82

Java Client: A Stream-based Peer to Peer Client..... 82
Java Client: A Stream-based Peer to Peer Service.....82
Administration API..... 83
Java Client: Add a Queue ACL Entry.....83
Java Client: Modify a Channel ACL Entry..... 83
Java Client: Delete a Realm ACL Entry..... 84
Java Client: Add a Schedule to a Universal Messaging Realm.....84
Java Client: Simple authentication server.....84
Java Client: Monitor realms for cluster creation, and cluster events.....85
Java Client: Monitor realms for client connections coming and going.....85
Java Client: Copy a channel and its events..... 85
Java Client: Monitor the remote realm log and audit file..... 85
Java Client: Export a realm to XML.....86
Java Client: Import a realm's configuration information..... 86
Java Client: Console-based Realm Monitor.....86
Java Client: Delete Service ACL..... 87
Java Client: Realm Monitor.....87
Provider for JMS..... 87
Java Client: JMS BytesMessage Publisher..... 87
Java Client: JMS BytesMessage Subscriber..... 88
Java Client: JMS MapMessage Publisher..... 88
Java Client: JMS MapMessage Subscriber..... 88
Java Client: JMS ObjectMessage Publisher.....89
Java Client: JMS ObjectMessage Subscriber.....89
Java Client: JMS StreamMessage Publisher.....89
Java Client: JMS StreamMessage Subscriber.....90
Java Client: JMS BytesMessage Queue Publisher.....90
Java Client: JMS BytesMessage Queue Subscriber..... 90
Java Client: JMS Queue Browser.....91
Channel / Queue / Realm Management.....91
Java Client: Creating a Channel.....91
Java Client: Deleting a Channel..... 92
Java Client: Creating a Queue..... 92
Java Client: Deleting a Queue.....92
Java Client: Create a Channel Join.....93
Java Client: Delete a Channel Join..... 93
Java Client: Purge events from a channel.....93
Java Client: Find the event id of the last event..... 94
Java Client: Add a realm to another realm.....94
Java Client: Multiplex a Session.....94
Enterprise Developer's Guide for C++..... 95
General Features.....95
Creating a Session.....95
Universal Messaging Events.....96
Channel Joins.....96

Universal Messaging Event Dictionaries....97
Google Protocol Buffers....97
Publish / Subscribe using Channel Topics....98
Publish / Subscribe Using Channels/Topics....98
Creating a Universal Messaging Channel....99
Finding a Channel....99
How to publish events to a Universal Messaging Channel....100
Asynchronous Subscriber....101
Channel Iterator....102
Batched Subscribe....103
Batched Find....104
Durable channel consumers and named objects....104
The Merge Engine and Event Deltas....106
Priority Messaging....107
Publish / Subscribe using Datastreams and Datagroups....108
Publish / Subscribe Using DataStreams and DataGroups....108
DataStreamListener....108
Creating and Deleting DataGroups....109
Managing DataGroup Membership....110
DataGroup Conflation Attributes....112
DataGroups Event Publishing....113
DataStream Event Publishing....114
Priority Messaging....114
Message Queues....115
Message Queues....115
Creating a Queue....115
Finding a Queue....116
Queue Publish....116
Asynchronous Queue Consuming....117
Synchronous Queue Consuming....118
Asynchronous Transactional Queue Consuming....119
Synchronous Transactional Queue Consuming....121
Queue Browsing / Peeking....122
Peer to Peer....123
Peer to Peer Services....123
Peer to Peer Event-based Clients....125
Peer to Peer Event-based Server Services....126
Peer to Peer Stream-based Clients....127
Peer to Peer Stream-based Server Services....129
Examples....130
Publish / Subscribe using Channel Topics....130
C++ Client: Channel Publisher....130
C++ Client: Transactional Channel Publisher....130
C++ Client: Asynchronous Channel Consumer....131
C++ Client: Synchronous Channel Consumer....131

C++ Client: Asynchronous Named Channel Consumer....131
C++ Client: Synchronous Named Channel Consumer....132
C++ Client: Event Delta Delivery....132
C++ Client: Batching Server Calls....133
C++ Client: Batching Subscribe Calls....133
Publish / Subscribe using Datastreams and Datagroups....133
C++ Client: DataStream Listener....133
C++ Client: DataGroup Publishing with Conflation....134
C++ Client: DataGroup Manager....134
C++ Client: Delete DataGroup....134
C++ Client: DataGroup Delta Delivery....135
Message Queues....135
C++ Client: Queue Publisher....135
C++ Client: Transactional Queue Publisher....135
C++ Client: Asynchronous Queue Consumer....136
C++ Client: Synchronous Queue Consumer....136
C++ Client: Peek Events on a Queue....136
Peer to Peer....137
C++ Client: An Event-based Peer to Peer Client and Server Service....137
C++ Client: A Stream-based Peer to Peer Client and Server Service....137
Administration API....138
C++ Client: Add a Queue ACL Entry....138
C++ Client: Modify a Channel ACL Entry....138
C++ Client: Delete a Realm ACL Entry....139
C++ Client: Monitor realms for client connections coming and going....139
C++ Client: Console-based Realm Monitor....139
C++ Client: Remove Node ACL....139
C++ Client: Authserver....139
Channel / Queue / Realm Management....140
C++ Client: Creating a Channel....140
C++ Client: Deleting a Channel....140
C++ Client: Creating a Queue....140
C++ Client: Deleting a Queue....141
C++ Client: Create Channel Join....141
C++ Client: Delete a Channel Join....142
C++ Client: Purge Events From a Channel....142
C++ Client: Create Queue Join....142
C++ Client: Delete Queue Join....143
Prerequisites....143
Prerequisites....143
Client SSL Configuration....144
Environment Setup : Windows....145
Environment Setup : Linux....146
Enterprise Developer's Guide for C#....147
Publish / Subscribe using Channel Topics....147

Creating a Channel....147
Finding a Channel....148
How to publish events to a Channel....149
Subscribe Asynchronously to a Channel....150
Synchronous Consumers....151
Batched Subscribe....151
Batched Find....152
Durable Channel Consumers and Named Objects....153
The Merge Engine and Event Deltas....154
Event Fragmentation on Channels....156
Consuming a JMS Map Message....156
Priority Messaging....157
Publish / Subscribe using Datastreams and Datagroups....157
Publish / Subscribe Using DataStreams and DataGroups....157
Enabling DataGroups and Receiving Event Callbacks....158
DataStreamListener....158
Managing Datagroups....159
Creating and Deleting DataGroups....159
Managing DataGroup Membership....160
DataGroup Conflation Attributes....161
Publishing to Datagroups....164
DataGroups Event Publishing....164
DataStream Event Publishing....164
Priority Messaging....164
Message Queues....165
Message Queues....165
Creating a Queue....165
Finding a Queue....166
Publishing events to a Queue....167
Asynchronously Consuming a Queue....168
Synchronously Consuming a Queue....168
Asynchronous Transactional Queue Consumption....169
Synchronous Transactional Queue Consumption....170
Browse (Peek) a Universal Messaging Queue....172
Event Fragmentation on Queues....173
Peer to Peer....175
Peer to Peer Services....175
Peer to Peer Event-based Clients....177
Peer to Peer Event-based Server Services....178
Peer to Peer Stream-based Clients....179
Peer to Peer Stream-based Server Services....180
Google Protocol Buffers....181
Examples....182
Publish / Subscribe using Channel Topics....182
Publish / Subscribe....182

Channel Publisher....182
Transactional Channel Publisher....182
Asynchronous Channel Consumer....183
Synchronous Channel Consumer....183
Asynchronous Named Channel Consumer....184
Synchronous Named Channel Consumer....184
Event Delta Delivery....185
Batching Server Calls....185
Batching Subscribe Calls....185
Publish / Subscribe using Datastreams and Datagroups....186
DataStream Listener....186
DataGroup Publishing with Conflation....186
DataGroup Manager....186
Delete DataGroup....187
DataGroup Delta Delivery....187
Message Queues....187
Queue Publisher....187
Transactional Queue Publisher....188
Asynchronous Queue Consumer....188
Synchronous Queue Consumer....188
Peek Events on a Queue....189
Requester - Request/Response....189
Responder - Request/Response....190
MyChannels.Universal Messaging API....190
MyChannels.Universal Messaging DataGroup Publisher....190
MyChannels.Universal Messaging Queue Publisher....190
MyChannels.Universal Messaging Topic Publisher....190
MyChannels.Universal Messaging DataGroup Listener....191
MyChannels.Universal Messaging Queue Consumer....191
MyChannels.Universal Messaging Topic Subscriber....191
RX Topic Subscriber....191
RX Queue Consumer....191
RX DataGroup Listener....191
Peer to Peer....192
An Event-based Peer to Peer Client and Server Service....192
A Stream-based Peer to Peer Client and Server Service....192
Administration API....192
Add a Queue ACL Entry....192
Modify a Channel ACL Entry....193
Delete a Realm ACL Entry....193
Monitor realms for client connections coming and going....194
Export a realm to XML....194
Import a realm's configuration information....194
Console-based Realm Monitor....194
Remove Service ACL....195

Authserver ... 195
Set Container ACL ... 195
Difference between 2 realms ... 200
Channel / Queue / Realm Management ... 200
Creating a Channel ... 200
Deleting a Channel ... 200
Creating a Queue ... 201
Deleting a Queue ... 201
Create Channel Join ... 201
Delete a Channel Join ... 202
Multiplex a Session ... 202
Purge Events From a Channel ... 202
Create Queue Join ... 203
Delete Queue Join ... 203
Prerequisites ... 203
C# Prerequisites ... 203
C# Client SSL Configuration ... 205
Globally Accessible DLLs ... 206
Messaging API ... 207
MyChannels.Universal Messaging API: Creating and Disposing of a Session ... 207
MyChannels.Universal Messaging API: Producers ... 208
MyChannels.Universal Messaging API: Consumers ... 208
MyChannels.Universal Messaging API: Reactive Extensions ... 209
Enterprise Developer's Guide for VBA ... 210
Publish / Subscribe ... 210
Publish/Subscribe ... 210
Subscribing Tasks ... 210
Subscribing to a Channel ... 210
Publishing Tasks ... 212
Creating a Session ... 212
Finding a Channel ... 212
Universal Messaging Events ... 212
Publishing Events to a Channel ... 213
Learn More ... 213
Event Properties ... 213
How the RTD Server Works ... 214
Setting the RTD Throttle Interval ... 214
Internal Event Processing ... 215
Universal Messaging RTD Server Internal Queues ... 216
OnChange() Event Using RTD ... 216
Prerequisites ... 217
Enterprise Developer's Guide for Python ... 217
Enterprise Client Development ... 218
Environment Configuration ... 218
Creating a Session ... 219

Subscribing to a Channel/Topic or Queue....219
DataStream - Receiving DataGroup Events....220
Publishing Events to a Channel or Queue....221
Writing an Event to a DataGroup....221
Asynchronous Exception Listener....222
Synchronously Requesting Events....223
Sample Applications....223
Publish / Subscribe using Channel Topics....223
Channel Publisher....223
Asynchronous Channel Subscriber....223
Channel Iterator....223
Publish / Subscribe using Datastreams and Datagroups....224
DataGroup Publisher....224
DataStream Listener....224
Message Queues....224
Queue Publisher....224
Asynchronous Queue Consumer....224
Synchronous Queue Reader....224
Python Objects....225
Universal Messaging Events....225
Event Dictionaries....225
API Language Comparisons....226

Mobile Client APIs....229
Client API for iPhone....230
iPhone Developer's Guide....230
Client API for Android....231
Android Developer's Guide....231

Web Client APIs....233
Overview of Web Client APIs....234
Web Developer's Guide for Javascript....235
Overview....235
Server Configuration for JavaScript....235
Server Configuration for HTTP Delivery....235
Server Configuration for HTTPS Delivery....236
Serving From Another Webserver....237
Web Client Development in JavaScript....238
Overview of using Publish/Subscribe....238
Publish/Subscribe Tasks....239
Using a Universal Messaging Channel....239
Subscribing to a Channel....239
Publishing Events to a Channel....240
DataStream - Receiving DataGroup Events....241
Optimizing Throughput....241
The Merge Engine and Event Deltas....241

Overview of using Message Queues....244
Message Queue Tasks....244
Using a Queue....244
Subscribing to a Queue....245
Publishing Events to a Queue....245
Asynchronous Transactional Queue Consuming....246
Web Developer's Guide for Adobe Flex....247
Overview....247
Publish / Subscribe using Channel Topics....247
Publish / Subscribe....247
Publishing Events to a Channel....247
Subscribing to a Channel....249
Durable channel consumers and named objects....250
The Merge Engine and Event Deltas....251
Publish / Subscribe using Datastreams and Datagroups....252
Publish / Subscribe....252
DataGroup Conflation Attributes....253
DataStreamListener....254
Message Queues....255
Message Queues....255
Publishing Events to a Queue....255
Asynchronous Queue Consuming....257
Asynchronous Transactional Queue Consuming....257
Peer to Peer....259
Peer to Peer Services....259
Peer to Peer Event-based Client....260
Flex Socket SSL....261
Flex socket SSL....261
Examples....262
Sample Socket Cross Domain Policy....262
Sample Flash Cross Domain Policy....262
Flex Example : Peer to Peer Echo Application....262
Flex Example : Chat Application....263
Web Developer's Guide for Silverlight....266
Developer's Guide for Silverlight....266
Silverlight Deployment....266
Examples....268
Live Stock Chart....268
Live Stock Indices....270
Simple Chat Room....273
Web Developer's Guide for Java....275
Web Developer's Guide for Java....275
Java Web Start....276

Commonly Used Features....277

Sessions....278
Channel Attributes....278
Channel Publish Keys....281
Queue Attributes....283
Native Communication Protocols....285
Comet Communication Protocols....288
Durable Consumers....290
Google Protocol Buffers....290
Named Objects....291
Event Filtering....291
Advanced Filtering with Selectors....293
Using Shared Memory Protocol....296
Storage Properties....296

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 18
- "Web Client APIs" on page 234
- "Mobile Client APIs" on page 229

1 Enterprise APIs

- Overview of the Enterprise Client APIs 18
- Enterprise Developer's Guide for Java 19
- Enterprise Developer's Guide for C++ 95
- Enterprise Developer's Guide for C# 147
- Enterprise Developer's Guide for VBA 210
- Enterprise Developer's Guide for Python 217
- API Language Comparisons 226

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 fully-featured enterprise-class client API:

  - "Enterprise Developer's Guide for Java" on page 19: developing Java applications/systems that will use Universal Messaging
  - Java Client API : the entire Universal Messaging Java client API

- **C++**

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

  - "Enterprise Developer's Guide for C++" on page 95: developing C++ applications/systems that will use Universal Messaging
  - C++ Client API : the entire Universal Messaging C++ client API

- **C# .NET**

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

  - "Enterprise Developer's Guide for C#" on page 147
  - C# Client API : the entire Universal Messaging C# .NET client API

- **Excel VBA**

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

  - "Enterprise Developer's Guide for VBA" on page 210

- **Python**

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

  - "Enterprise Developer's Guide for Python" on page 217

See Universal Messaging's "Language API Comparison Grid" on page 226 for an overview of basic differences between each API.

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

Create 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

   ```
   Session 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

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 Javacode snippet, we construct our Universal Messaging Event object (`nConsumeEvent`), and, in this example, passa byte array data into the constructor:

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

Channel Joins

Creating Channel Joins

Channel joins can be created using the nmakechanjoin join sample application which is provided in the bin directory of the Universal Messaging installation. For further information on using this example please see the Make Channel Join example page.

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

Channel joins can also be deleted. Please see the Delete Channel Join example for more information.

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, ech 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);
```

Inter-Cluster joins can also be created through the Enterprise Manager.

Interest Propagation

Overview

Universal Messaging offers the ability to forward data received on one independent realm or cluster to many other independent realms or clusters, which may reside in geographically distinct locations. Traditionally this is done using the Join mechanism, which will forward all events from one channel to another.

There is an alternative mechanism, namely Interest Propagation. This mechanism expands upon the functionality provided by joins by providing the ability to forward events only when there are subscribers to a channel of the same name on the remote realm or cluster. Forwarding only events which have an active subscription reduces the number of events and bandwidth used on these links.

Realms and clusters keep track of interest on remote realms that they have a direct connection to. This means that beyond the initial setup of a channel, no further configuration is required.

Managing Remote Interest

Managing interest on a remote realm is done programmatically using the Universal Messaging Administrative API. Each channel present on a realm or cluster can be linked to a pair of attributes canSend and canReceive.

Enabling the canReceive attribute on a channel of a realm or cluster will enable this realm or cluster to receive information on the given channel from other directly connected realms or clusters that have a channel of the same name. The realm or cluster notifies all connected realms when this attribute changes for a given channel.

Data is only forwarded from a realm or cluster to a remote realm or cluster if all of the following conditions are met:

- A channel with the same name exists on the remote realm or cluster, and
- the canReceive flag is enabled for the remote channel, and
- there is an active subscription present on the remote channel

Enabling the canSend attribute on a channel in a realm or cluster will enable this realm or cluster to begin forwarding data to other realms or clusters it is aware of. Data is forwarded to every realm which the source realm is aware of that has a channel with the same name and is able to receive the event (it has the canReceive flag enabled and has an interested subscriber).

Sample Usage

All installations of Universal Messaging come with a sample application called interestmanagerutility. This is an application which takes a series of commands to manage the interest properties for a given set of realms.

Event Dictionaries

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 19) to make use of our Event Dictionary (`nEventProperties`).

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.

Universal Messaging supports server-side filtering of Google Protocol Buffers, and this, coupled with Google Protocol Buffer's space-efficient serialization can be used to reduce the amount of data delivered to a client. If server side filtering is not required, the serialised protocol buffers could be loaded into a normal nConsume Event as the event data.

The structure of the data is defined in a .proto file, messages are constructed from a number of different types of fields and these fields can be required, optional or repeated. Protocol Buffers can also include other Protocol Buffers.

The serialization uses highly efficient encoding to make the serialized data as space efficient as possible, and the custom generated code for each data format allows for rapid serialization and deserialization.

Using Google Protocol Buffers with Universal Messaging

Google supplies libraries for Protocol Buffers in Java, C++ and Python, and third party libraries provide support for many other languages including Flex, .NET, Perl, PHP etc. Universal Messaging's client APIs provide support for the construction of Google Protocol Buffer events through which the serialized messages can be passed.

These nProtobufEvents are integrated seamlessly in Universal Messaging, allowing for server-side filtering of Google Protocol Buffer events, which can be sent on resources just like a normal Universal Messaging Event. The server side filtering of messages is achieved by providing the server with a description of the data structures

(constructed at the .proto compile time, using the standard protobuf compiler and the --descriptor_set_out option). The default location the server looks in for descriptor files is /plugins/ProtobufDescriptors and this can be configured through the Enterprise Manager. The server will monitor this folder for changes, and the frequency of these updates can be configured through the Enterprise Manager. The server can then extract the key value pairs from the binary Protobuf message, and filter message delivery based on user requirements.

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.

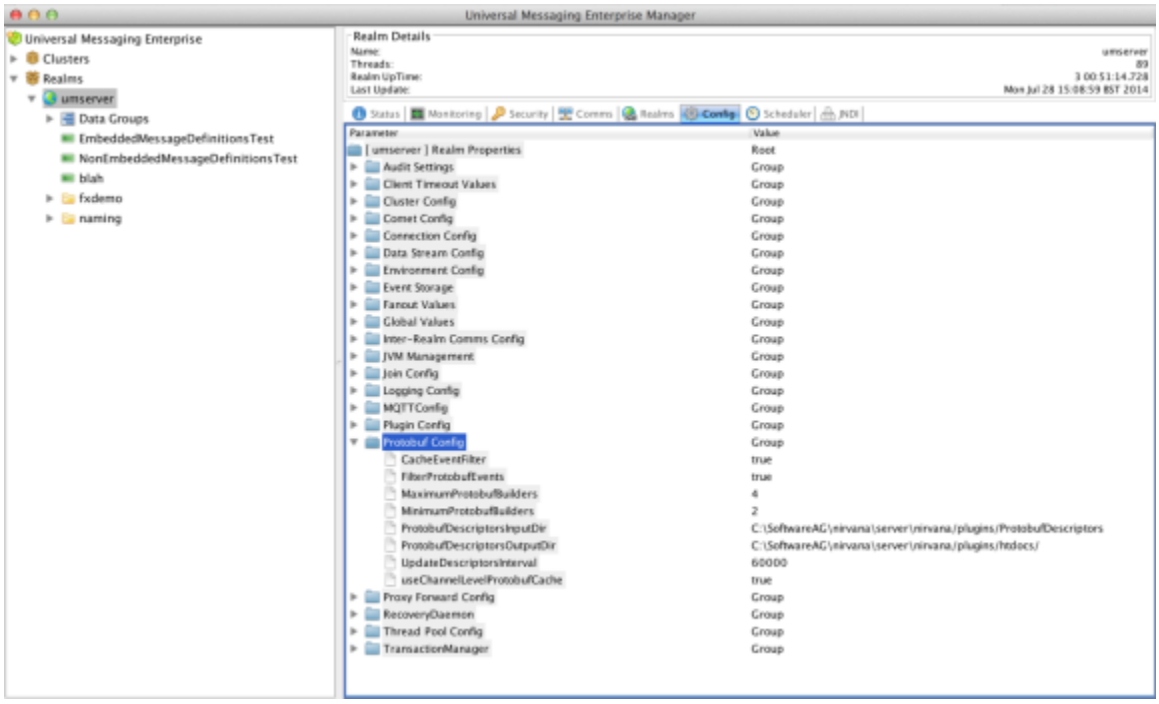
```
public void go(nConsumeEvent evt) {
    if (evt instanceof nProtobufEvent) {
        totalMsgs++;
        // Get the data of the message
        byte[] buffer = evt.getEventData();
        if(((nProtobufEvent) evt).getTypeName().equals("BidOffer")){
            BidOffer bid = null;
            bid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

Configuring Universal Messaging for use with Protocol Buffers

Protocol buffers have their own section in the Enterprise Manager Config panel.



These are explained via their tooltips.

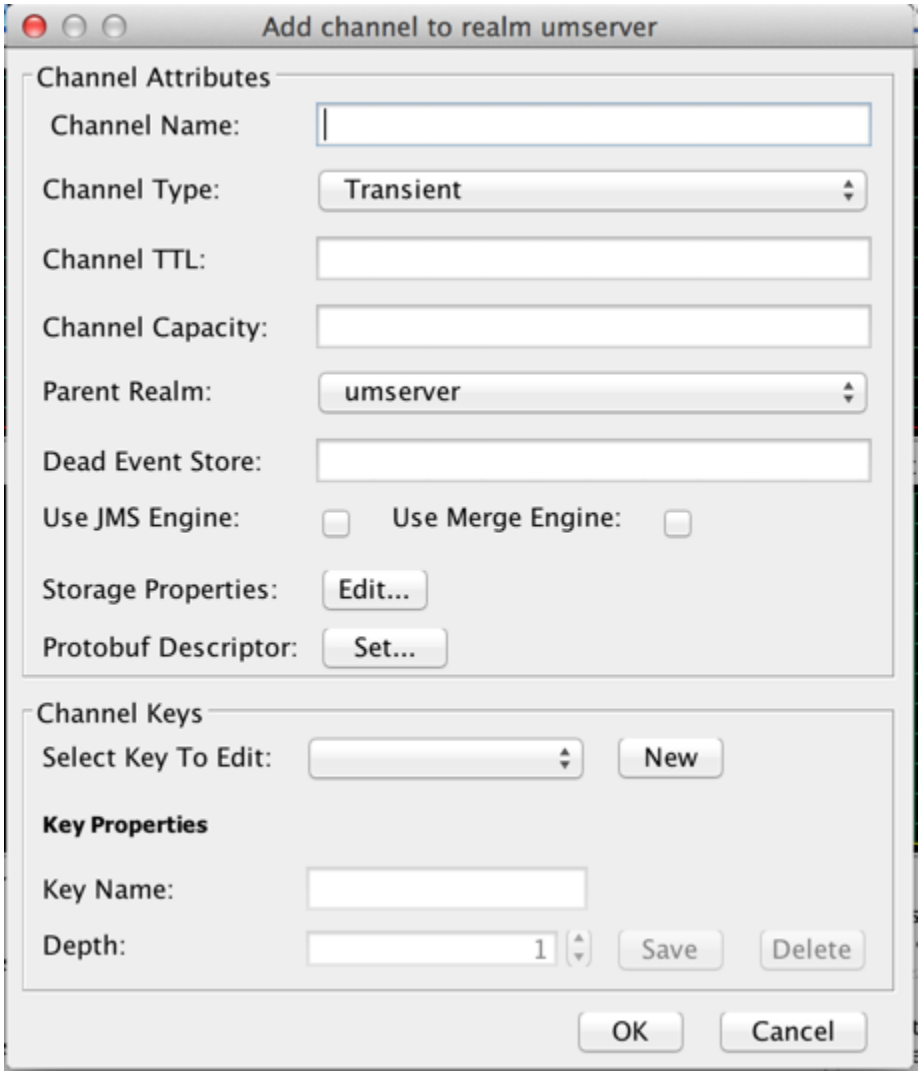
UseChannelLevelProtobufCache indicates whether the descriptors are set against the channel, or put into a folder (see "Legacy Google Protobuf Support" below). UpdateDescriptorInterval, the maximum and minimum builder numbers, and the input and output directories only apply for the legacy option.

FilterProtobufEvents is set to true by default, and must be set to true to enable filtering of protobuf events.

If nested messages need to be filtered on, then GlobalValues -> ExtendedMessageSelectors, must be set to true. Again this is now enabled by default but will not be enabled in installs upgraded from older versions.

Protobuf with the Enterprise Manager

When creating a channel via the Enterprise Manager, there is a protobuf descriptor section on the create dialogue. Clicking "Set..." here brings up a file dialogue where a descriptor file (generated when the protobuf is compiled, as described above) can be selected. Multiple descriptor files can currently only be set programmatically, not via the Enterprise Manager.



Any channel with an associated descriptor can be snooped in the normal way. Enterprise Manager will use the descriptor to deserialise the message, and will show the contents of that message in the event details.

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 serialised 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'".

Legacy protocol Buffer support

These nProtobufEvents are integrated seamlessly in Universal Messaging, allowing for server-side filtering of Google Protocol Buffer events, which can be sent on resources just like normal Universal Messaging events. The server side filtering of messages is achieved by providing the server with a description of the data structures (constructed at the .proto compile time, using the standard protobuf compiler and the --descriptor_set_out option). The default location the server looks in for descriptor files is /plugins/ProtobufDescriptors and this can be configured through the Enterprise Manager. The server will monitor this folder for changes, and the frequency of these updates can be configured through the Enterprise Manager. The server can then extract the key value pairs from the binary Protobuf message and filter message delivery based on user requirements.

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. To enable this, the server outputs a descriptor set to a configurable directory (by default the htdocs directory for the realm) and this can then be made available through a file plugin etc. The directory can be changed through the Enterprise Manager. The Enterprise Manager can then be configured to load this file using -DProtobufDescSetURL and then the contents of the protocol buffers can be parsed.

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 DataStreams and DataGroups provide an alternative style of Publish/Subscribe where user subscriptions can be managed remotely on behalf of clients (see "Publish / Subscribe Using DataStreams and DataGroups" on page 37).

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, and provides example code snippets for all relevant concepts:

- "Creating a Universal Messaging Channel" on page 27
- "Finding a Universal Messaging Channel" on page 28
- "Publishing events to a Channel" on page 28

- "Sending XML Dom Objects" on page 30
- "Asynchronous Subscriber" on page 30
- "Channel Iterator" on page 31
- "Batched Subscribe" on page 32
- "Batched Find" on page 33
- "Durable channel consumers and named objects" on page 33
- "Event Fragmentation on Channels" on page 35
- "The Merge Engine and Event Deltas" on page 35
- "Priority Messaging" on page 37

Example source:

- "Code Examples" on page 70

Creating a Universal Messaging Channel

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

In order to create a channel, first of all you must create an nSession object, which is your effectively 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:

```
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.

Finding a Universal Messaging 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 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);
```

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

Publishing events to a Channel

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

- "Reliable Publish" on page 28
- "Transactional Publish" on page 29

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 );
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 nTransasctionAttributes(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 nTransasctionAttributes(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:

```
nEventProperteis 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.

Durable channel consumers and named objects

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

Universal Messaging supports durable consumers through use of Universal Messaging named objects as shown by the following example code.

Names objects can also be managed via the enterprise manager.

Asynchronous Durable Consumer

An example of how to create a named object that begins from event id 0, 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 named object and begin consuming events from the channel at event id 0
    // i.e. the beginning of the channel
    nNamedObject nobj = myChannel.createNamedObject("unique1", 0, true);
    myChannel.addSubscriber(this , nobj);
  }
  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 named object that begins from event id 0, 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
    // start the iterator at the beginning of the channel (event id 0)
    nNamedObject nobj = myChannel.createNamedObject("unique2", 0, true);
    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();
  }
}
```

Both synchronous and asynchronous channel consumers allow message selectors to be used in conjunction with named objects. 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();
}
```

Priority

Two subscribers can hold a subscription to the same named object. One is given priority and will process events during normal operation. If, however, the subscriber with priority is disconnected for whatever reason, and is unable to process events, the second subscriber to that named object will take over and continue to process events as they come in. This allows failover, with backup subscribers handling events if the subscriber with priority goes down.

To do this, we simply create the subscriber with a boolean specifying if this subscriber priority. Only one subscriber is allowed priority at any given time. An example of a named object specifying priority is shown below:

```
nNamedObject named = myChannel.createNamedObject(subname, startEid,
                     persistent, cluster, priority);
```

Event Fragmentation on Channels

By default, Universal Messaging will only allow events to be published if the size of the event is less than 1Mb. Although this limit can be changed in the Enterprise Manager (see **Config** tab, **FanoutValues/MaxBufferSize**), this is not generally recommended; it is usually far more efficient to fragment large events into smaller chunks for publishing.

Universal Messaging can transparently fragment and reconstruct events. Thus, a developer need only invoke one method call to fragment and publish an event. In the same way, the resulting event will be transparently reconstructed when received by the consumer. Under the hood, however, Universal Messaging will publish several smaller messages representing the large event.

A summary of the code needed to publish and consume fragmented events is provided below.

Publishing

The code to publish a large event using fragmentation is as follows:

```
// The chunk_size is the max size (bytes) for each event.  Multiple events will
// be published of size chunk_size until the entire event has been sent.
int chunk_size = 50000;
fw = new nConsumeEventFragmentWriter(myChannel,chunk_size);
// Rather than myChannel.publish(evt), we let the fragment writer handle the publish
fw.publish(evt)
```

Subscribing

The code to consume a large event using fragmentation is as follows:

```
// In this example the enclosing class implements nEventListener
fr = new nConsumeEventFragmentReader(this);
// Rather than directly add 'this' as the nEventListener, add the new fragment reader
myChannel.addSubscriber(fr);
```

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();
```

Priority Messaging

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.

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.

It is possible to specify multiple levels of priority for events on the same channel. This behaviour will cause the events to be delivered highest priority first. When doing this it is important to realise that events on a channel will no longer be delivered on a first in first out basis.

Publish / Subscribe Using DataStreams and DataGroups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging DataGroups 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.

There are two resources that are used when interacting with DataGroups: DataStreams and *DataGroups*.

DataStreams

A Data Stream is a destination for published events. Publishers with appropriate permissions can write events directly to Data Streams. A Universal Messaging client session can optionally have a Data Stream, and receive events through it.

A Data Stream can be a member of one or more Data Groups.

DataGroups

Any event written to a Data Group will be propagated to all Data Streams that are members of that Data Group.

Data Groups may also contain other Data Groups. Any event written to an upper level Data Group will be written to all contained Data Groups, and thus to all contained Data Streams.

Note that all Data Streams are automatically added to the realm server's Default Data Group. Writing an event to the Default Data Group, therefore, will ensure it is delivered to any client with a session configured to use a Data Stream.

This section demonstrates Universal Messaging pub / sub using DataGroups 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.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

```
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.sMergeEvent, 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)
```

Priority Messaging

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 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.

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 Administration Tool, 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 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 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.

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 nTransasctionAttributes(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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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.

Event Fragmentation on Queues

By default, Universal Messaging will only allow events to be published if the size of the event is less than 1Mb. Although this limit can be changed in the Enterprise Manager (see **Config** tab, **FanoutValues/MaxBufferSize**), this is not generally recommended; it is usually far more efficient to fragment large events into smaller chunks for publishing.

Universal Messaging can transparently fragment and reconstruct events. Thus, a developer need only invoke one method call to fragment and publish an event. In the same way, the resulting event will be transparently reconstructed when received by the consumer. Under the hood, however, Universal Messaging will publish several smaller messages representing the large event.

A summary of the code needed to publish and consume fragmented events is provided below.

Publishing

The code to publish a large event using fragmentation is as follows:

```
    // The chunk_size is the max size (bytes) for each event.  Multiple events will
```

```
    // be published of size chunk_size until the entire event has been sent.
    int chunk_size = 50000;
    fw = new nConsumeEventFragmentWriter(myQueue,chunk_size);
    // Rather than myQueue.publish(evt), we let the fragment writer handle the publish
    fw.publish(evt)
```

Subscribing

There are various approaches to consuming fragmented events from queues:

Asynchronous Queue Consumer

```
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
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 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 Transactional Queue Consumer

```
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
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 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();
                }
        }
}
```

Synchronous Queue Consumer

```
public class mySyncQueueReader {
        nQueueSyncReader reader = null;
```

```
        nQueue myQueue = null;
        public mySyncQueueReader() throws Exception {
                // construct your session and queue objects  here
                // construct the queue reader
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 10);
                reader = myQueue.createFragmentReader(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 Transactional Queue Consumer

```
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
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 10);
                reader = myQueue.createTransactionalFragmentReader(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();
                }
        }
 }
```

Peer to Peer Services

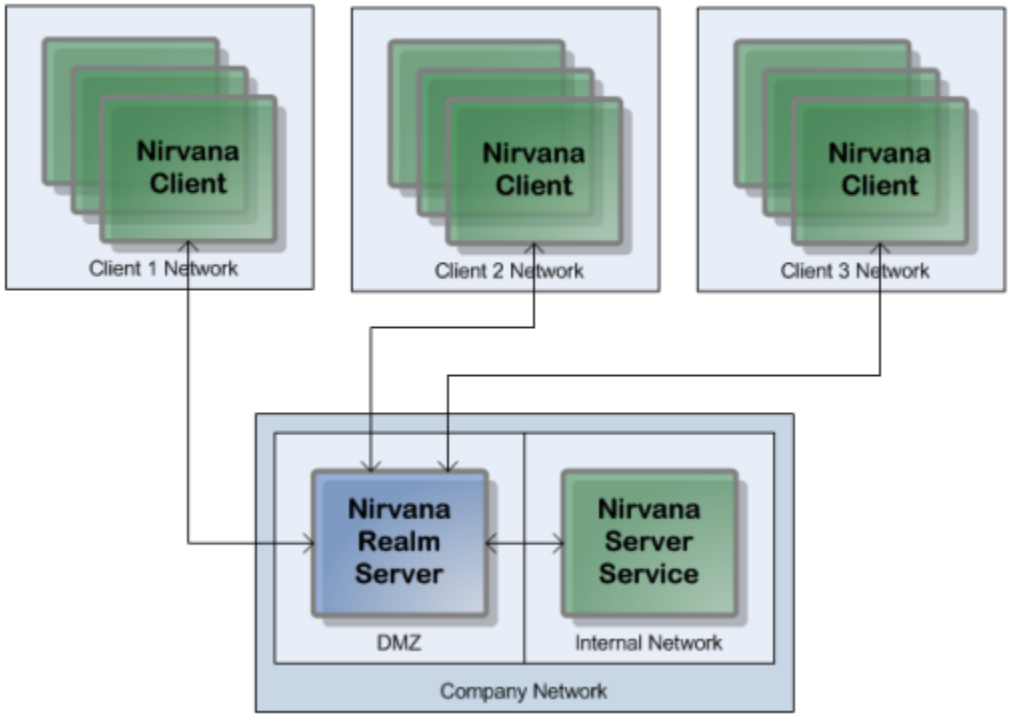
Peer to peer is one of several messaging paradigms supported by Universal Messaging.

Universal Messaging provides a rich set of APIs that provide developers with the ability to create Peer to Peer (P2P) applications. We call these Peer to Peer applications *Services*. This guide will demonstrate how Universal Messaging Peer to Peer Services work, and provides examples code snippets for all relevant concepts.

P2P Service Components

There are two parts to a Peer to Peer Service in Universal Messaging: a *Server Service* and a *Client*.

When a Server Service is running, it is visible within the Universal Messaging Namespace and is available to any Client wishing to connect. The Universal Messaging Realm Server acts as the bridge that connects Clients to Server Services. Each Server Service can support multiple Clients.



Universal Messaging Peer to Peer Client and Server Services

The *Server Service* is a process that registers itself with a Universal Messaging Realm so it is visible to Clients wishing to connect.

A Universal Messaging Peer to Peer *Service Client* is a process that connects to a Universal Messaging Realm, obtains a reference to a Server Service and begins communicating with it.

When a Client connects to the Server Service, all communication between the Client and server service takes place through the Universal Messaging Realm, using Universal Messaging's standard communication protocols.

P2P Service Types

There are two types of Universal Messaging Peer to Peer Services:

- *Event-based Services*

  Universal Messaging Peer to Peer Event-based Services communicate via events which are published by the Event-based Client, and received and responded to by the Event-based Server Service.

- *Stream-based Services*

  Universal Messaging Peer to Peer Stream-based Services communicate via input and output streams on both the Client and Server Service. Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Peer to Peer Event-based Clients

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by a Client, and received and responded to by an Event-based Server Service.

The Universal Messaging P2P API is simple to use. There are only a very small number of objects and calls that need to be made in order for you to construct a P2P Service Client, connect to a Realm, and find or list available Services.

Creating an Event-based Service Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
String[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nServiceFactory factory = new nServiceFactory( nsa );
nServiceInfo info = factory.findService("example");
nEventService serv = (nEventService)factory.connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Sending Events to Server Services

Once you have connected to the Service, and you have an instance of the Service, you can then begin publishing your Universal Messaging events to the Service, by using the following command:

```
serv.write(new nConsumeEvent("TAG", message.getBytes()));
```

The Client Service can receive events from the Server Service either synchronously, or asynchronously via a callback interface.

Synchronously Receiving Events from the Server Service

Clients can synchronously read incoming events. The following code will return an event once one is received from the Server Service:

```
nConsumeEvent event = serv.read();
```

Asynchronously Receiving Events from the Server Service

A Client may alternatively asynchronously receive events from the Event-based Server Service by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

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

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Peer to Peer Event-based Server Services

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by an Event-based Client, and received and responded to by an *Event-based Server Service*.

Creating an Event-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

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

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Event-based Server Service:

```
nServerService Server = factory.createEventService( "example",
        "Example Event-based Service" );
while ( true ) {
        nEventService serv = (nEventService) server.accept();
        // your logic goes here....
        // e.g. query a database, make a connection, send an email, etc.
        System.out.println("Got connection " + serv.getServiceInfo().getName());
}
```

The code snippet above shows how to create an Event-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When connections are made to the Event-based Server Service, the Service can receive events from Clients either synchronously or asynchronously via a callback interface.

Synchronously Receiving Events from the Client

The Server Service can synchronously read incoming events. The following code will return an event once one is received from the Client:

```
nConsumeEvent event = serv.read();
```

Asynchronously Receiving Events from the Client

The Server Service may alternatively asynchronously receive events by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

```
public void receivedEvent(nConsumeEvent evt) {
        Console.WriteLine("Consumed event " + event.getEventID());
}
```

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Sending Events to Clients

You can send events back to the Client as follows:

```
serv.write(new nConsumeEvent("TAG", message.getBytes()));
```

Peer to Peer Stream-based Clients

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the *Stream-based Client* and the Stream-based Server Service.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating a Stream-based Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
String[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nServiceFactory factory = new nServiceFactory( nsa );
nServiceInfo info = factory.findService("example");
nEventService serv = (nEventService)factory.connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Writing Client Data to a Stream-based Server Service

Once a client has connected to a Service, the client can write data to the Service. The client can obtain a reference to the Service's Output Stream object and then write to it as follows:

```
OutputStream oStream = serv.getOutputStream();
oStream.write((new UTF8Encoding()).GetBytes("Hello World"));
oStream.flush();
```

Receiving Responses from a Stream-based Server Service

To receive responses from the Service, the client must first obtain a reference to the Service's Input Stream object, and then read from it as follows:

```
InputStream iStream = serv.getInputStream();
byte[] buff = new byte[ 100 ];
try {
        InputStream is = serv.getInputStream();
        while ( true ) {
                is.read( buff );
                System.out.println("Read "+new String(buff));
        }
} catch ( Exception ex ) {
}
```

Peer to Peer Stream-based Server Services

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the Stream-based Client and the *Stream-based Server Service*.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating an Stream-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

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

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Stream-based Server Service:

```
nServerService Server = factory.createStreamService( "example",
        "Example Stream-based Service" );
while ( true ) {
        nStreamService serv = (nStreamService) server.accept();
        InputStream Stream inputstream = serv.getInputStream();
        OutputStream Stream outputstream = serv.getOutputStream();
        // your logic goes here....
        // e.g. query a database, make a connection, send an email, etc.
        System.out.println("Got connection " + serv.getServiceInfo().getName());
}
```

The code snippet above shows how to create an Stream-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When a connection is made to the Stream-based Server Service, the Service has an Input Stream (which can be read from), and an Output Stream (which can be written to).

Receiving Data from a Stream-based Client

The Server Service's Input Stream represents data coming from the client. The following code snippet shows how to obtain this Input Stream:

```
InputStream iStream = serv.getInputStream();
```

Sending Data to a Stream-based Client

The Server Service's Output Stream represents data going to the client. The following code snippet shows how to obtain this Output Stream:

```
OutputStream oStream = serv.getOutputStream();
```

Provider for JMS

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

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, network address translation devices and it does not require either any additional web server to perform.

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.

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

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 Administration Tool section.

JMSAdmin

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 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. Example usage is as follows:

```
Java -DRNAME [-DPRINCIPAL] [-DPASSWORD] -DCONTEXT_FACTORY
  -DPROVIDER_URL JMSAdmin bind | unbind | list | queueFactory |
  topicFactory |connectionFactory | queue | topic name / alias
```

where:

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.

PRINCIPAL is the subject (if any) you 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 providers context factory implementation. The default CONTEXT FACTORY is com.pcbsys.nirvana.nSpace.Universal MessagingContextFactory 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, TopicConnectionFactory, QueueConnectionFactory, rates and movie. The Universal MessagingContext used with your JMS application will now be able to lookup these objects and use them within your application.

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 JMS application passes the required System properties used by your jsse enabled JVM. The Universal Messaging download contains some example Java key store 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 CA keystore called nirvanacacerts.jks, which is again located in the /server/Universal Messaging/bin directory

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.

Application Server Integration (JBoss)

JMS provides extensions that allows JMS providers to be integrated into Application Servers. This section describes the steps involved in integrating Universal Messaging Provider for JMS with jboss. All references to jboss assume jboss version 3.2.x or 4.0.x are being used.

This guide will provide the following information:

- "Message Queue Configuration" on page 64
- "Server Session Pool Configuration" on page 65
- "Jboss Configuration & Service Deployment" on page 65
- "Universal Messaging Server Configuration" on page 65
- "Running Message Driven Beans" on page 66

Configuration Terms

Firstly, for the following sections, we will be referencing certain directories for the install. These are described below:

- <*jboss_home*> - the jboss installation directory
- <*jboss_bin*> - the jboss bin directory located under <jboss_home>/bin
- <*jboss_default*> - default server, under <jboss_home>/server/default
- <*jboss_default_lib*> - default server lib directory, under <jboss_default>/lib
- <*jboss_default_deploy*> - default server deploy directory, under <jboss_default>/deploy
- <*jboss_default_conf*> - default server configuration directory, usually <*jboss_default*>/conf

Message Queue Configuration

Jboss provides its own JMS Message Queue service that we need to replace with Universal Messaging's own message queue service. This section will describe the steps needed to integrate Universal Messaging's Message service into Jboss.

To do this we need to change the references in the jboss xml configuration files so that the Universal Messaging Message Queue service is used:

In the <jboss_default_conf>/standardjboss.xml file and replace the tags that say DefaultJMSProvider with Universal MessagingJMSProvider.

Server Session Pool Configuration

Jboss provides its own server session pool objects that allow multiple JMS sessions to be pooled within the Message Queue Service. Universal Messaging also provides its own session pool objects. This section describes the steps necessary to integrate Universal Messaging's Server Session Pool into jboss.

To do this we need to change the references in the jboss xml configuration files so that Universal Messaging's Server Session Pools are used by the Message Queue Service:

In the <jboss_default_conf>/standardjboss.xml file and replace the tags that say StdJMSPool with Universal MessagingJMSPool.

Jboss Configuration

This section describes the steps necessary to ensure the jboss server is ready to begin using Universal Messaging as the Message Queue Service provider. Please complete the following steps:

1. Remove the <jboss_default_deploy>/JMS directory completely
2. Put the Universal Messaging-service.xml file into the <jboss_default_deploy> directory (found in the src/xml/jboss directory of your install)
3. Put the Universal Messaging nJMS.jar, nClient.jar and nJ2EE.jar files into the <jboss_default_lib> directory from your /lib directory of the install
4. Modify the run script for jboss to include the following -D parameter when the jboss server is started:

   -Dnirvana.provider.url=<your.Universal Messaging.rname (e.g.) nsp://localhost:9000 which is the default RNAME

Universal Messaging Realm Server Configuration

In order to configure the Universal Messaging Realm Server, please ensure you have either read the Universal Messaging Enterprise Manager JNDI integration section or are familiar with the jmsadmin sample programs. These tools enable you to create the Universal Messaging JNDI objects necessary for the jboss server to successfully use Universal Messaging as the JMS message queue provider. In this example, we will use the jmsadmin example program, however should you choose to, you can also use the Universal Messaging Enterprise Manager by following the steps found in the guide.

Please follow the steps below:

1. Start the Universal Messaging 3.0 server
2. Open a Universal Messaging Client environment prompt
3. Type : jmsadmin bind topicFactory TopicConnectionFactory (followed by return)
4. Type : jmsadmin bind queueFactory QueueConnectionFactory (followed by return)
5. Type : jmsadmin bind queue queue/DLQ (followed by return)

This will set up the queue and topic factories used by the Universal Messaging Provider for JMS message service, as well as setting up the jboss DLQ used for internal message processing.

Once these steps have been completed, you can then start the jboss server which will now be using Universal Messaging Provider for JMS as the message queue provider

Running Message Driven Beans

Message driven beans can be deployed within application servers to provide a run-frame for JMS services. Once you have created your message driven beans and they are deployed into the jboss server, you must ensure that all topics and queues used by the MDBs have been created using the jmsadmin tool, so that they can be referenced within the Universal Messaging JNDI context used by the Universal Messaging messaging service.

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.

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 has five different channel types each of which have different requirements when storing data:

- The Persistent channel always writes data to disk regardless of the number of subscribers.
- The Simple and Reliable channels both store all events in memory. The difference being that Reliable channels increment and store the Event ID on disk rather than in memory so that in the event of failure, event IDs are not reset to 0.
- The Mixed channel stores the Event ID in the same way as the Reliable channel but the storage of events is specified in the nConsumeEvent by calling setPersistent(). (in JMS the persistence is set by the DeliveryMode).
- With a Transient channel all events are fanned out to subscribers and then dropped so no events are stored in memory or on disk.

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 68.

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.

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 table 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 and Simple channels will be stored in memory. Transient channels do not store events prior to delivery.

JMS Engine

Channel Type	Mixed	Persistent	Reliable	Simple	Transient
Active Durable Subscribers					
One or more Synchronous Consumers or Inactive					

JMS Engine

Channel Type	Mixed	Persistent	Reliable	Simple	Transient
Durable Subscribers					
No Durable Subscribers					
No Subscribers					

Universal Messaging Channel Engine

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

Universal Messaging Queue Engine

Channel Type	Mixed	Persistent	Reliable	Simple	Transient
Active Consumers					
No Subscribers					

Code Examples

Pub / Sub - Channels

The following are self-contained pub / sub examples which include full application source code:

- "Java Client: Channel Publisher" on page 72
- "Java Client: Transactional Channel Publisher" on page 73
- "Java Client: Asynchronous Channel Consumer" on page 73
- "Java Client: Synchronous Channel Consumer" on page 74
- "Java Client: Asynchronous Named Channel Consumer" on page 74
- "Java Client: Synchronous Named Channel Consumer" on page 75
- "Java Client: XML Channel Publisher" on page 75
- "Java Client: Asynchronous XML Channel Consumer" on page 75
- "Java Client: Event Delta Delivery" on page 76
- "Java Client: Batching Server Calls" on page 76
- "Java Client: Batching Subscribe Calls" on page 76

Pub / Sub - DataGroups

The following are self-contained nirvana DataGroup examples which include full application source code:

- "Java Client: DataStream Listener" on page 77
- "Java Client: DataGroup Publishing with Conflation" on page 77
- "Java Client: DataGroup Manager" on page 77
- "Java Client: Delete DataGroup" on page 78
- "Java Client: DataGroup Delta Delivery" on page 78

Message Queues

The following are self-contained message queue examples which include full application source code:

- "Java Client: Queue Publisher" on page 78
- "Java Client: Transactional Queue Publisher" on page 79
- "Java Client: Asynchronous Queue Consumer" on page 79
- "Java Client: Asynchronous Transactional Queue Consumer" on page 79

- "Java Client: Synchronous Queue Consumer" on page 80
- "Java Client: Synchronous Transactional Queue Consumer" on page 80
- "Java Client: Peek events on a Queue" on page 80
- "Java Client: Requester - Request/Response" on page 81
- "Java Client: Responder - Request/Response" on page 81

Peer to Peer (P2P)

The following are self-contained P2P examples which include full application source code:

- "Java Client: An Event-based Peer to Peer Client" on page 82
- "Java Client: An Event-based Peer to Peer Server Service" on page 82
- "Java Client: A Stream-based Peer to Peer Client" on page 82
- "Java Client: A Stream-based Peer to Peer Service" on page 82

Universal Messaging Administration API

The following are self-contained examples of Universal Messaging's Administration API which include full application source code:

- "Java Client: Add a Queue ACL Entry" on page 83
- "Java Client: Modify a Channel ACL Entry" on page 83
- "Java Client: Delete a Realm ACL Entry" on page 84
- "Java Client: Add a Schedule to a Universal Messaging Realm" on page 84
- "Java Client: Simple authentication server" on page 84
- "Java Client: Monitor realms for cluster creation, and cluster events" on page 85
- "Java Client: Monitor realms for client connections coming and going" on page 85
- "Java Client: Copy a channel and its events" on page 85
- "Java Client: Monitor the remote realm log and audit file" on page 85
- "Java Client: Export a realm to XML" on page 86
- "Java Client: Import a realm's configuration information" on page 86
- "Java Client: Console-based Realm Monitor" on page 86
- "Java Client: Delete Service ACL" on page 87
- "Java Client: Realm Monitor" on page 87

Universal Messaging Provider for JMS

The following are self-contained JMS examples which include full application source code:

- "Java Client: JMS BytesMessage Publisher" on page 87
- "Java Client: JMS BytesMessage Subscriber" on page 88
- "Java Client: JMS MapMessage Publisher" on page 88
- "Java Client: JMS MapMessage Subscriber" on page 88
- "Java Client: JMS ObjectMessage Publisher" on page 89
- "Java Client: JMS ObjectMessage Subscriber" on page 89
- "Java Client: JMS StreamMessage Publisher" on page 89
- "Java Client: JMS StreamMessage Subscriber" on page 90
- "Java Client: JMS BytesMessage Queue Publisher" on page 90
- "Java Client: JMS BytesMessage Queue Subscriber" on page 90
- "Java Client: JMS Queue Browser" on page 91

Universal Messaging Channel / Queue / Realm Management

The following are self-contained nirvana client examples which include full application source code:

- "Java Client: Creating a Channel" on page 91
- "Java Client: Deleting a Channel" on page 92
- "Java Client: Creating a Queue" on page 92
- "Java Client: Deleting a Queue" on page 92
- "Java Client: Create a Channel Join" on page 93
- "Java Client: Delete a Channel Join" on page 93
- "Java Client: Purge events from a channel" on page 93
- "Java Client: Find the event id of the last event" on page 94
- "Java Client: Add a realm to another realm" on page 94
- "Java Client: Multiplex a Session" on page 94

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 74

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 73.

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 Named Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel using a named object.

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 named subscription
                 (default: OS username)
[start eid]    - The Event ID to start subscribing from if the named subscriber 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 named object 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 Named Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel using a named object 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 named 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 named object 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)
[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 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.

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 Datagroups

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 80.

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 80

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 79.

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 79.

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 79.

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.

Peer to Peer

Java Client: An Event-based Peer to Peer Client

This example shows how to build a simple Event-based P2P Client.

The example consists of a server and a client; the server will echo anything typed by the client.

Usage

```
np2pecho
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: An Event-based Peer to Peer Server Service

This example shows how to build a simple Event-based P2P Server Service.

The example consists of a server and a client; the server will echo anything typed by the client.

Usage

```
np2pechoserver
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: A Stream-based Peer to Peer Client

This example shows how to build a simple Stream-based P2P service.

The example consists of a server and a client; the server essentially exposes a shell to the client.

Usage

```
np2pshell
Note: -? provides help on environment variables
```

Application Source Code

See the online documentation for a code example.

Java Client: A Stream-based Peer to Peer Service

This is a class that implements the Server Service component of the ShellServer example application.

Usage

```
np2pshellserver <shell>
<Required Arguments>
<shell> - The type of shell yoou want to offer.
          For example cmd for win32 or bash for unix
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
[peak] - 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 events

This example demonstrates how to copy a channel and its events from one location to another.

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 RNAME of a 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.out
<-a auditfile> - A file name to store the audit messages to (without this it
                 will go to system.out
<-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 and configuration information to an XML file so that it can be imported into any other realm.

Usage

```
nexportrealmxml [export_file_location]
<Optional Arguments> -all -realms -cluster -realmacl -realmcfg -channels
                     -channeacls -joins -queues -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 and configuration information from an XML file.

Usage

```
nimportrealmxml file_name
<Optional Arguments> -all -realmacl -realmcfg -channels -channeacls -queues
                     -queueacls -interfaces
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: Delete Service ACL

This shows how the ACL for a P2P service can be removed.

Usage

```
ndelp2pacl <service name> <user> <host> [-r]
<Required Arguments>
<service name> - Service name parameter to delete the service ACL entry from
<user> - User name parameter to delete the service ACL entry from
<host> - Host name parameter to delete the service 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: 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.

Provider for JMS

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)
```

```
<topicName> - JMS Topic to publish on
<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)
<destinationName> - JMS Destination to subscribe to
<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)
<topicName> - JMS Topic to publish on
<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)
```

```
<destinationName> - JMS Destination to subscribe to
<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)
<topicName> - JMS Topic to publish on
<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)
<destinationName> - JMS Destination to subscribe to
<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)
<topicName> - JMS Topic to publish on
```

```
<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)
<destinationName> - JMS Destination to subscribe to
<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)
<queueName> - JMS Queue to publish on
<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)
<destinationName> - JMS Destination to subscribe to
<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]
<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
S - For a simple (stored in memory) channel with non-persistent eids
T - For a transient (no server based storage)
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)
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
S - For a simple (stored in memory) queue with non-persistent eids
T - For a transient (no server based storage)
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.

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

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);
```

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.

Channels can be joined using the Universal Messaging Enterprise Manager GUI or programmatically.

In joining two Universal Messaging channels there is one compulsory option and two optional ones. The compulsory option is the destination channel. The optional parameters are the maximum join hops and a message selector to be applied to the join.

Multiple Path Delivery

Universal Messaging users can define multiple paths over different network protocols between the same places in Universal Messaging. Universal Messaging guarantees that the data always gets delivered once and once only.

Channel joins can be created using the Make Channel Join sample application which is provided in the bin directory of the Universal Messaging installation. For further information on using this example please see the make channel join example page.

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 );
```

Channel joins can also be deleted. Please see the delete channel join example for more information.

Universal Messaging Event Dictionaries

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 96) to make use of our Event Dictionary (`nEventProperties`).

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.

Universal Messaging supports server-side filtering of Google Protocol Buffers, and this, coupled with Google Protocol Buffer's space-efficient serialization can be used to reduce the amount of data delivered to a client. If server side filtering is not required, the serialised protocol buffers could be loaded into a normal nConsume Event as the event data.

The structure of the data is defined in a .proto file, messages are constructed from a number of different types of fields and these fields can be required, optional or repeated. Protocol Buffers can also include other Protocol Buffers.

The serialization uses highly efficient encoding to make the serialized data as space efficient as possible, and the custom generated code for each data format allows for rapid serialization and deserialization.

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 Flex, .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.

These nProtobufEvents are integrated seamlessly in nirvana, allowing for server-side filtering of Google Protocol Buffer events, which can be sent on resources just like a

normal nirvana Events. The server side filtering of messages is achieved by providing the server with a description of the data structures(constructed at the .proto compile time, using the standard protobuf compiler and the --descriptor_set_out option). The default location the sever looks in for descriptor files is /plugins/ProtobufDescriptors and this can be configured through the enterprise manager. The server will monitor this folder for changes, and the frequency of these updates can be configured through the enterprise manager. The server can then use to extract the key value pairs from the binary Protobuf message and filter message delivery based on user requirements.

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 EM is no running on the same machine as the server. To enable this, the server outputs a descriptor set to a configurable directory (by default the htdocs directory for the realm) and this can then be made available through a file plugin etc. The directory can be changed through the enterprise manager. The enterprise manager can then be configured to load this file using -DProtobufDescSetURL and then the contents of the protocol buffers can be parsed.

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 DataStreams and DataGroups 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, and provides example code snippets for all relevant concepts.

- "Creating a Universal Messaging Channel" on page 99
- "Finding a Channel" on page 99
- "How to publish events to a Universal Messaging Channel" on page 100
- "Asynchronous Subscriber" on page 101
- "Channel Iterator" on page 102
- "Batched Subscribe" on page 103

- "Batched Find" on page 104
- "Durable channel consumers and named objects" on page 104
- "The Merge Engine and Event Deltas" on page 106
- "Priority Messaging" on page 107

Creating a Universal Messaging Channel

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

In order to create a channel, first of all you must create an nSession object, which is your effectively 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 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.

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 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);
```

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

How to publish events to a Universal Messaging 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 nTransasctionAttributes(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.

Durable channel consumers and named objects

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

Universal Messaging supports durable consumers through use of Universal Messaging named objects as shown by the following example code.

Names objects can also be managed via the enterprise manager.

Asynchronous Durable Consumer

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

```
class mySubscriber : public nEventListener {
  public:
  mySubscriber(){
    // construct your session
```

```
    // and channel objects here
    // create the named object and begin consuming events from the channel at event id 0
    // i.e. the beginning of the channel
    nNamedObject *nobj = myChannel->createNamedObject("unique1", 0, true);
    myChannel->addSubscriber(this , nobj);
  }
  void go(nConsumeEvent *event) {
    printf("Consumed event %d",event->getEventID());
  }
  int main(int argc, char** argv) {
    new mySubscriber();
    return 0;
  }
}
```

Synchronous Durable Consumer

An example of how to create a named object that begins from event id 0, persistent and is used in conjunction with a synchronous event consumer:

```
class myIterator {
  private:
   nChannelIterator *iterator = null;
  public:
   myIterator(){
     // construct your session
     // and channel objects here
     // start the iterator at the beginning of the channel (event id 0)
     nNamedObject *nobj = myChannel->createNamedObject("unique2", 0, true);
     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;
   }
}
```

Both synchronous and asynchronous channel consumers allow message selectors to be used in conjunction with named objects. 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:

```
void go(nConsumeEvent *event) {
  printf("Consumed event %d",event->getEventID());
  event->ack();
}
```

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.

Priority Messaging

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.

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 channel. This behaviour will cause the events to be delivered highest priority first. When doing this it is important to realise that events on a channel will no longer be delivered on a first in first out basis.

Publish / Subscribe using Datastreams and Datagroups

Publish / Subscribe Using DataStreams and DataGroups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging DataGroups 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.

There are two resources that are used when interacting with DataGroups: *DataStreams* and *DataGroups*.

DataStreams

A Data Stream is a destination for published events. Publishers with appropriate permissions can write events directly to Data Streams. A Universal Messaging client session can optionally have a Data Stream, and receive events through it.

A Data Stream can be a member of one or more Data Groups.

DataGroups

Any event written to a Data Group will be propagated to all Data Streams that are members of that Data Group.

Data Groups may also contain other Data Groups. Any event written to an upper level Data Group will be written to all contained Data Groups, and thus to all contained Data Streams.

Note that all Data Streams are automatically added to the realm server's Default Data Group. Writing an event to the Default Data Group, therefore, will ensure it is delivered to any client with a session configured to use a Data Stream.

This section demonstrates Universal Messaging pub / sub using DataGroups 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

```
public class datagroupListener implements nDataGroupListener {
    nSession* mySession;
    nDataGroup* myDataGroup;
    public datagroupListener(nSession* session, nConflationAttributes* confattribs,
           std::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::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 110) 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)
```

Priority Messaging

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.

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 Administration Tool, or programmatically (see "Creating a Queue" on page 115). 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 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 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.

Queue Publish

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

- "Reliable Publish" on page 116
- "Transactional Publish" on page 117

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 96) 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 95) and our queue (see "Finding a Queue" on page 116), we then need to construct our events (see "Universal Messaging Events" on page 96) 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 nTransasctionAttributes(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 "Universal Messaging Event Dictionaries" on page 97) 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 "Universal Messaging Event Dictionaries" on page 97) 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 "Universal Messaging Event Dictionaries" on page 97) 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 "Universal Messaging Event Dictionaries" on page 97) 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 "Universal Messaging Event Dictionaries" on page 97) 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".

Peer to Peer

Peer to Peer Services

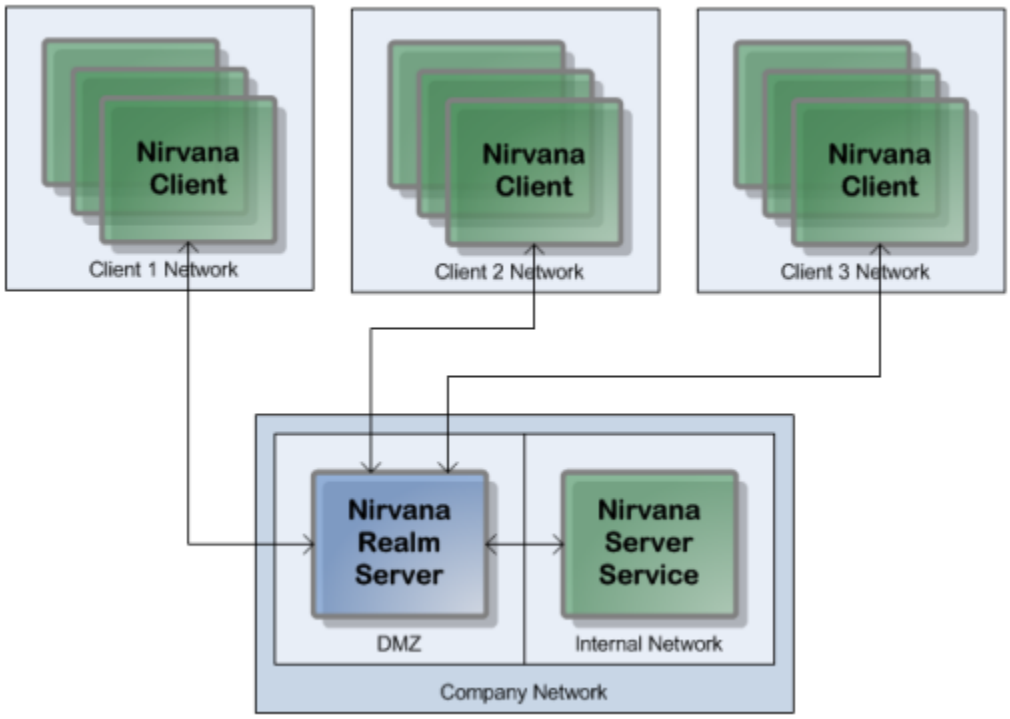
Universal Messaging provides a rich set of APIs that provide developers with the ability to create Peer to Peer (P2P) applications. We call these Peer to Peer applications *Services*.

This guide will demonstrate how Universal Messaging Peer to Peer Services work, and provides examples code snippets for all relevant concepts.

P2P Service Components

There are two parts to a Peer to Peer Service in Universal Messaging: a *Server Service* and a Client.

When a Server Service is running, it is visible within the Universal Messaging Namespace and is available to any Client wishing to connect. The Universal Messaging Realm Server acts as the bridge that connects Clients to Server Services. Each Server Service can support multiple Clients.



Universal Messaging Peer to Peer Client and Server Services

The *Server Service* is a process that registers itself with a Universal Messaging Realm so it is visible to Clients wishing to connect.

A Universal Messaging Peer to Peer *Service Client* is a process that connects to a Universal Messaging Realm, obtains a reference to a Server Service and begins communicating with it.

When a Client connects to the Server Service, all communication between the Client and server service takes place through the Universal Messaging Realm, using Universal Messaging's standard communication protocols.

P2P Service Types

There are two types of Universal Messaging Peer to Peer Services:

- **Event-based Services**

Universal Messaging Peer to Peer Event-based Services communicate via events which are published by the Event-based Client (see "Peer to Peer Event-based Clients" on page 125), and received and responded to by the Event-based Server Service.

- **Stream-based Services**

  Universal Messaging Peer to Peer Stream-based Services communicate via input and output streams on both the Client and Server Service. Anything written to the output stream of the Stream-based Service Client (see "Peer to Peer Stream-based Clients" on page 127) is received via the input stream of the Stream-based Server Service (see "Peer to Peer Stream-based Server Services" on page 129) and vice versa.

Examples

The code examples "P2P Echo" and "P2P Shell" give full application source code to implement Server Services and Clients:

Peer to Peer Event-based Clients

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by a Client, and received and responded to by an Event-based Server Service.

The Universal Messaging P2P API is simple to use. There are only a very small number of objects and calls that need to be made in order for you to construct a P2P Service Client, connect to a Realm, and find or list available Services.

Creating an Event-based Service Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
std::string[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nServiceFactory *factory = new nServiceFactory( nsa );
nServiceInfo *info = factory->findService("example");
nEventService *serv = (nEventService)factory->connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Sending Events to Server Services

Once you have connected to the Service, and you have an instance of the Service, you can then begin publishing your Universal Messaging events to the Service, by using the following command:

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

To receive responses from the Server Service, the client Service can receive events either synchronously or asynchronously via a callback interface.

Synchronously Receiving Events from the Server Service

Clients can synchronously read incoming events. The following code will return an event once one is received from the Server Service:

```
nConsumeEvent *event = serv->read();
```

Asynchronously Receiving Events from the Server Service

A Client may alternatively asynchronously receive events from the Event-based Server Service by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

```
void receivedEvent(nConsumeEvent *evt) {
    printf("Consumed event %d",event->getEventID());
}
```

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Peer to Peer Event-based Server Services

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by an Event-based Client (see "Peer to Peer Event-based Clients" on page 125), and received and responded to by an *Event-based Server Service*.

Creating an Event-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

```
std::string[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nServiceFactory *factory = new nServiceFactory( nsa );
```

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Event-based Server Service:

```
nServerService *Server = factory->createEventService( "example",
    "Example Event-based Service" );
while ( true ) {
    nEventService *serv = (nEventService) server->accept();
    Stream *inputstream = serv->getInputStream();
    Stream *outputstream = serv->getOutputStream();
    // your logic goes here....
    // e.g. query a database, make a connection, send an email, etc.
    printf("Got connection %d",(serv->getServiceInfo())->getName());
}
```

The code snippet above shows how to create an Event-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When connections are made to the Event-based Server Service, the Service can receive events from Clients either synchronously or asynchronously via a callback interface.

Synchronously Receiving Events from the Client

The Server Service can synchronously read incoming events. The following code will return an event once one is received from the Client:

```
nConsumeEvent *event = serv->read();
```

Asynchronously Receiving Events from the Client

The Server Service may alternatively asynchronously receive events by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

```
void receivedEvent(nConsumeEvent *evt) {
    printf("Consumed event %d",event->getEventID());
}
```

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Sending Events to Clients

You can send events back to the Client as follows:

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

Examples

The code example "P2P Echo" source code shows how to implement an Event-based Server Service and Client.

Peer to Peer Stream-based Clients

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the *Stream-based Client* and the Stream-based Server Service.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating a Stream-based Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
std::string[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nServiceFactory *factory = new nServiceFactory( nsa );
nServiceInfo *info = factory->findService("example");
nStreamService *serv = (nStreamService *)factory->connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Writing Client Data to a Stream-based Server Service

Once a client has connected to a Service, the client can write data to the Service. The client can obtain a reference to the Service's Output Stream object and then write to it as follows:

```
Stream *os = serv->getOutputStream();
//Read a character from the standard input until a \n is reached which indicates
// the end of a command.
int pos = 0;
char command[256];
try
{
    bool run = true;
    while (run)
    {
        command[pos] = getchar();
        if (command[pos] == '\n')
        {
            if (pos > 0)
            {
                pos++;
                unsigned char *temp = new unsigned char[pos];
                memcpy (temp, command, pos);
                os->write(temp, 0, pos);
                os->flush();
                pos = 0;
            }
            else
            {
                run = false;
            }
        }
        else
        {
            pos++;
        }
    }
}
```

Receiving Responses from a Stream-based Server Service

To receive responses from the Service, the client must first obtain a reference to the Service's Input Stream object, and then read from it as follows:

```
Stream *is = serv->getInputStream();
unsigned char *pBuf = new unsigned char[bufferSize];
try
{
    int size = 0;
    while (true)
```

```
    {
        is->wait();
        is->lock();
        is->setPosition (0);
        int i = 0;
        while ((i = is->readByte()) != -1)
        {
            putchar((char)i);
            size++;
        }
        is->unlock();
    }
}
```

Peer to Peer Stream-based Server Services

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the *Stream-based Client* and the *Stream-based Server Service*.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating an Stream-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

```
std::string[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes *nsa = new nSessionAttributes(RNAME);
nServiceFactory *factory = new nServiceFactory( nsa );
```

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Stream-based Server Service:

```
nServerService *Server = factory->createStreamService( "example",
    "Example Stream-based Service" );
while ( true ) {
        nEventService *serv = (nEventService) server->accept();
        Stream *inputstream = serv->getInputStream();
        Stream *outputstream = serv->getOutputStream();
        // your logic goes here....
        // e.g. query a database, make a connection, send an email, etc.
        printf("Got connection %s",(serv->getServiceInfo())->getName());
}
```

The code snippet above shows how to create an Stream-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When a connection is made to the Stream-based Server Service, the Service has an Input Stream (which can be read from), and an Output Stream (which can be written to).

Receiving Data from a Stream-based Client

The Server Service's Input Stream represents data coming from the client. The following code snippet shows how to obtain this Input Stream:

```
Stream *iStream = serv->getInputStream();
```

Sending Data to a Stream-based Client

The Server Service's Output Stream represents data going to the client. The following code snippet shows how to obtain this Output Stream:

```
Stream *oStream = serv->getOutputStream();
```

Examples

The code example "P2P Shell" shows how to implement a Stream-based Server Service and Client:

For more information on Universal Messaging Peer to Peer Services please see the API documentation.

Examples

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 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)
[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 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)
[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 131

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 131

Usage

```
channeliterator <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the rname of the server 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: Asynchronous Named Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel using a named object.

Usage

```
namedsubscriber <rname> <channel name> <named object> [persist] [auto] [start eid]
                                                      [debug] [count] [selector]
<Required Arguments>
<rname> - the rname of server to connect to
<channel name> - Channel name parameter for the channel to subscribe to
<named object> - Unique id of the named object
[Optional Arguments]
[persist] - If the named object will be stored persistently
[auto] - If messages are acknowledged auomatically or manually
[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 Named Channel Consumer

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

Usage

```
namedchanneliterator <rname> <channel name> [name] [start eid] [debug] [count]
                                            [clusterwide] [persistent] [selector]
<Required Arguments>
<rname> - the rname of the server to connect to
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name] - specifies the unique name to be used for a named subscription
[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
[clusterwide] - specifies whether the named object is to be used across a cluster
                (default : false)
[persistent] - specifies whether the named object state is to be stored on 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.

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 Datastreams and Datagroups

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 rname of the server to connect to
[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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
[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 rname of the server to connect to
<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 rname of the server to connect to
<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 136

Usage

```
qsubscriber <rname> <queue name> [debug] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the rname of the server to connect to
<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 136

Usage

```
qreader <rname> <queue name> [debug] [timeout] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the rname of the server to connect to
<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 rname of the server to connect to
<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.

Peer to Peer

C++ Client: An Event-based Peer to Peer Client and Server Service

This example shows how to build a simple Event-based P2P Service.

The example consists of a server and a client; the server will echo anything typed by the client.

Usage

```
nP2PEcho <rname> [server]
<Required Arguments>
<rname> - the rname of the server to connect to
[Optional Arguments]
[server] - write 'server' to run echo server, or leave out to run echo client
```

Application Source Code

See the online documentation for a code example.

C++ Client: A Stream-based Peer to Peer Client and Server Service

This example shows how to build a simple Stream-based P2P service.

The example consists of a server and a client; the server essentially exposes a shell to the client.

Usage

```
nP2PShell <rname> [shell]
<Required Arguments>
<rname> - the rname of the server to connect to
[Optional Arguments]
[shell] - The type of shell you want to offer. For example cmd for win32 or
          bash for unix, leave out for client
```

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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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
S - For a simple (stored in memory) channel with non-persistent eids
T - For a transient (no server based storage)
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 rname of the server to connect to
<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 rname of the server to connect to
<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
S - For a simple (stored in memory) queue with non-persistent eids
T - For a transient (no server based storage)
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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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

Prerequisites

Universal Messaging C++ makes use of certain non-standard C++ libraries. Before using Universal Messaging C++, these libraries must be installed and the environment setup accordingly.

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

OpenSSL is also required on the system running Universal Messaging C++. OpenSSL is installed by default on most Unix based operating systems, however if you require OpenSSL please refer to the OpenSSL website at http://www.openSSL.org.

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 144.

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. For example, to compile the applications the compiler needs to know where the POCO libraries and headers are.

Environment setup is different for different operating systems:

- "Environment Setup : Windows" on page 145
- "Environment Setup : Linux" on page 146

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

Once Universal Messaging has been installed, the sample applications can be run from the "C++ Examples Command Prompt". The guide below explains how the Universal Messaging C++ environment can be set up for compiling and running the applications on a Windows 32-bit operating system.

The Universal Messaging C++ and POCO libraries can be found in the cplus\windows\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:

```
C:\> set PATH=C:\Universal Messaging 5.0.xxxx\cplus\windows\lib;%PATH%
```

   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. The default location of the libraries is:

     C:\Universal Messaging 5.0.xxxx\cplus\windows\lib;

     where xxxx is the build number and 5.0 is the version number.

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, Universal Messaging.lib and certain C++ header files. The libs are located in cplus\windows\lib 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

Once the environment has been set up as described above, the sample applications can be built by either opening the application's project file in Microsoft Visual Studio or by running vcbuild. In order to use vcbuild, either run the Microsoft Visual Studio command prompt and ensure that the Universal Messaging environment is set up, or run vsvars32.bat in a command prompt:

```
C:\> "C:\Program Files\Microsoft Visual Studio 9.0\Common7\Tools\vsvars32.bat"
```

The environment will now be set up for running vcbuild. In order to compile an application, navigate to the application's directory and run:

```
C:\Universal Messaging 5.0.xxxx\cplus\examples\channeliterator> vcbuild
```

This will compile the application in a new folder called Release. To clean this directory so that the application can be recompiled, run:

```
C:\Universal Messaging 5.0.xxxx\cplus\examples\channeliterator> vcbuild /clean
```

After compilation the executable (.exe) will be present in a folder called Release found in the same directory as the source code for the application.

Environment Setup : Linux

The guide below explains how the Universal Messaging C++ environment can be set up for compiling and running the applications on Linux 64-bit operating system.

Running a Universal Messaging C++ application 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 Universal Messaging.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=
      /home/username/Universal Messaging_5.0.xxxx/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 /home/username/Universal Messaging_5.0.xxxx/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 Universal Messaging C++ application, 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/example directory contains sample applications written using the Universal Messaging C++ API 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

Once the environment has been set up as described above, the sample applications can be compiled by navigating to the application's directory (cplus/examples/applicationName) and running:

```
[user@host channeliterator]$ make
```

To clean this directory so that the application can be recompiled, run:

```
[user@host channeliterator]$ make clean
```

The executable (no file extension) will now be present in the same directory as the source code after compilation.

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.

Publish / Subscribe using Channel Topics

Creating a Channel

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

In order to create a channel, first of all you must create an nSession object, which is your effectively 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:

```
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.

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);
```

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", (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 nTransasctionAttributes(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 nTransasctionAttributes(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:

```
nEventProperteis 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:

```
nEventProperteis 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.

Durable Channel Consumers and Named Objects

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

Universal Messaging supports durable consumers through use of Universal Messaging named objects as shown by the following example code.

Named objects can also be managed via the enterprise manager.

Asynchronous

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

```
public class mySubscriber : nEventListener {
        public mySubscriber() {
                // construct your session and channel objects here
                // create the named object and begin consuming events
                // from the beginning of the channel (event id 0)
                nNamedObject nobj = myChannel.createNamedObject("unique1", 0, true);
                myChannel.addSubscriber(this , nobj);
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                new mySubscriber();
        }
}
```

Synchronous

An example of how to create a named object that begins from event id 0, persistent and is used in conjunction with a synchronous event consumer:

```
public class myIterator {
        nChannelIterator iterator = null;
        public myIterator() {
                // construct your session and channel objects here
                // start the iterator
                nNamedObject nobj = myChannel.createNamedObject("unique2", 0, true);
                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();
        }
}
```

Both synchronous and asynchronous channel consumers allow message selectors to be used in conjunction with named objects. 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) {
        Console.WriteLine("Consumed event " + event.getEventID());
        event.ack();
}
```

Priority

Two subscribers can hold a subscription to the same named object. One is given priority and will process events during normal operation. If, however, the subscriber with priority is disconnected for whatever reason, and is unable to process events, the second subscriber to that named object will take over and continue to process events as they come in. This allows failover, with backup subscribers handling events if the subscriber with priority goes down.

To do this, we simply create the subscriber with a boolean specifying if this subscriber priority. Only one subscriber is allowed priority at any given time. An example of a named object specifying priority is shown below:

```
nNamedObject named = myChannel.createNamedObject(
                     subname, startEid, persistent, cluster, priority);
```

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.

Event Fragmentation on Channels

By default, Universal Messaging will only allow events to be published if the size of the event is less than 1Mb. Although this limit can be changed in the Enterprise Manager (see *Config* tab, *FanoutValues/MaxBufferSize*), this is not generally recommended; it is usually far more efficient to fragment large events into smaller chunks for publishing.

Universal Messaging can transparently fragment and reconstruct events. Thus, a developer need only invoke one method call to fragment and publish an event. In the same way, the resulting event will be transparently reconstructed when received by the consumer. Under the hood, however, Universal Messaging will publish several smaller messages representing the large event.

A summary of the code needed to publish and consume fragmented events is provided below.

Publishing

The code to publish a large event using fragmentation is as follows:

```
// The chunk_size is the max size (bytes) for each event.  Multiple events will
// be published of size chunk_size until the entire event has been sent.
int chunk_size = 50000;
fw = new nConsumeEventFragmentWriter(myChannel,chunk_size);
// Rather than myChannel.publish(evt), we let the fragment writer handle the publish
fw.publish(evt)
```

Subscribing

The code to consume a large event using fragmentation is as follows:

```
// In this example the enclosing class implements nEventListener
fr = new nConsumeEventFragmentReader(this);
// Rather than directly add 'this' as the nEventListener, add the new fragment reader
myChannel.addSubscriber(fr);
```

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();
  }
}
```

Priority Messaging

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.

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 channel. This behaviour will cause the events to be delivered highest priority first. When doing this it is important to realise that events on a channel will no longer be delivered on a first in first out basis.

Publish / Subscribe using Datastreams and Datagroups

Publish / Subscribe Using DataStreams and DataGroups

Publish / Subscribe is one of several messaging paradigms supported by Universal Messaging. Universal Messaging DataGroups 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.

There are two resources that are used when interacting with DataGroups: *DataStreams* and *DataGroups*.

DataStreams

A Data Stream is a destination for published events. Publishers with appropriate permissions can write events directly to Data Streams. A Universal Messaging client session can optionally have a Data Stream, and receive events through it.

A Data Stream can be a member of one or more Data Groups.

DataGroups

Any event written to a Data Group will be propagated to all Data Streams that are members of that Data Group.

Data Groups may also contain other Data Groups. Any event written to an upper level Data Group will be written to all contained Data Groups, and thus to all contained Data Streams.

Note that all Data Streams are automatically added to the realm server's Default Data Group. Writing an event to the Default Data Group, therefore, will ensure it is delivered to any client with a session configured to use a Data Stream.

This section demonstrates Universal Messaging pub / sub using DataGroups in C#, and provides example code snippets for all relevant concepts.

Enabling DataGroups 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 Datagroups

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.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

```
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.sMergeEvent, 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 Datagroups

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)
```

Priority Messaging

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 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.

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 Administration Tool, 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 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 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.

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 nTransasctionAttributes(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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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:

```
nEventProperteis 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.

Event Fragmentation on Queues

By default, Universal Messaging will only allow events to be published if the size of the event is less than 1Mb. Although this limit can be changed in the Enterprise Manager (see **Config** tab, **FanoutValues/MaxBufferSize**), this is not generally recommended; it is usually far more efficient to fragment large events into smaller chunks for publishing.

Universal Messaging can transparently fragment and reconstruct events. Thus, a developer need only invoke one method call to fragment and publish an event. In the same way, the resulting event will be transparently reconstructed when received by the consumer. Under the hood, however, Universal Messaging will publish several smaller messages representing the large event.

A summary of the code needed to publish and consume fragmented events is provided below.

Publishing

The code to publish a large event using fragmentation is as follows:

```
// The chunk_size is the max size (bytes) for each event.  Multiple events will
// be published of size chunk_size until the entire event has been sent.
int chunk_size = 50000;
fw = new nConsumeEventFragmentWriter(myQueue,chunk_size);
// Rather than myQueue.publish(evt), we let the fragment writer handle the publish
fw.publish(evt)
```

Subscribing

There are various approaches to consuming fragmented events from queues:

Asynchronous Queue Consumer

```
public class myAsyncQueueReader : nEventListener {
        nQueue myQueue = null;
        public myAsyncQueueReader() {
                // construct your session and queue objects here
                // begin consuming events from the queue
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 10);
        }
        public void go(nConsumeEvent event) {
                Console.WriteLine("Consumed event "+event.getEventID());
        }
        public static void Main(String[] args) {
                new myAsyncQueueReader();
        }
}
```

Asynchronous Transactional Queue Consumer

```
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
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 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();
        }
}
```

Synchronous Queue Consumer

```
public class mySyncQueueReader {
        nQueueSyncReader reader = null;
        nQueue myQueue = null;
        public mySyncQueueReader() {
                // construct your session and queue objects here
                // construct the queue reader
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 10);
                reader = myQueue.createFragmentReader(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();
        }
}
```

Synchronous Transactional Queue Consumer

```
public class mySyncTxQueueReader {
        nQueueSyncTransactionReader reader = null;
        nQueue myQueue = null;
        public mySyncTxQueueReader() {
                // construct your session and queue objects here
                // construct the transactional queue reader
                nConsumeEventFragmentReader cefr = new nConsumeEventFragmentReader(this);
                nQueueReaderContext ctx = new nQueueReaderContext(cefr, 10);
                reader = myQueue.createTransactionalFragmentReader(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();
        }
}
```

Peer to Peer

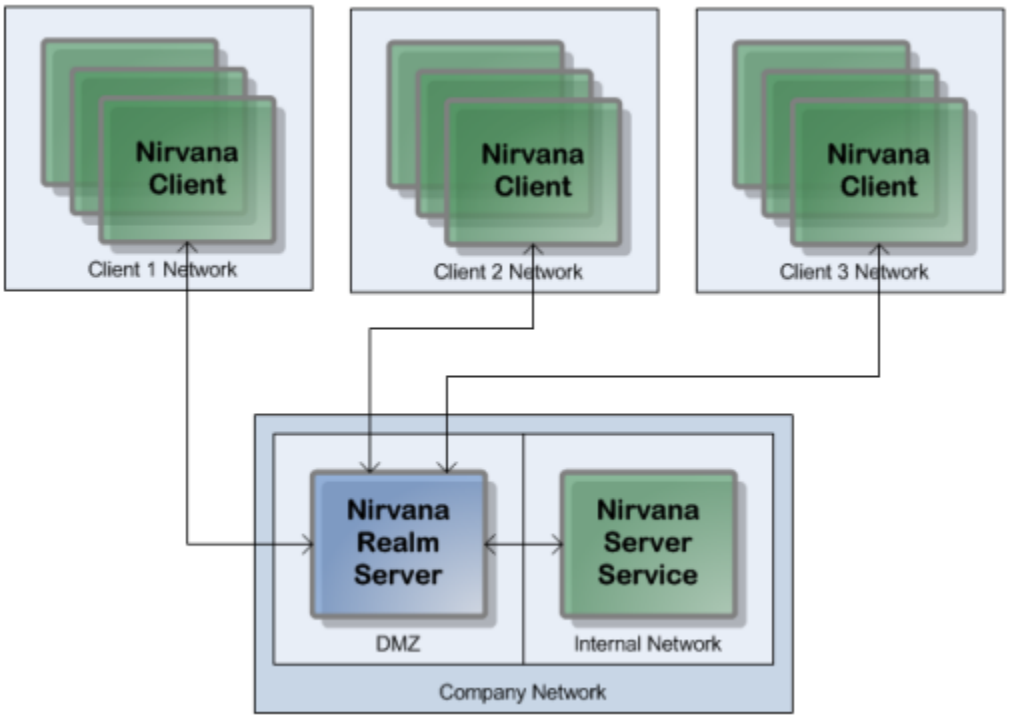
Peer to Peer Services

Universal Messaging provides a rich set of APIs that provide developers with the ability to create Peer to Peer (P2P) applications. We call these Peer to Peer applications *Services*. This guide will demonstrate how Universal Messaging Peer to Peer Services work, and provides examples code snippets for all relevant concepts.

P2P Service Components

There are two parts to a Peer to Peer Service in Universal Messaging: a *Server Service* and a *Client*.

When a Server Service is running, it is visible within the Universal Messaging Namespace and is available to any Client wishing to connect. The Universal Messaging Realm Server acts as the bridge that connects Clients to Server Services. Each Server Service can support multiple Clients.



Universal Messaging Peer to Peer Client and Server Services

The *Server Service* is a process that registers itself with a Universal Messaging Realm so it is visible to Clients wishing to connect.

A Universal Messaging Peer to Peer *Service Client* is a process that connects to a Universal Messaging Realm, obtains a reference to a Server Service and begins communicating with it.

When a Client connects to the Server Service, all communication between the Client and server service takes place through the Universal Messaging Realm, using Universal Messaging's standard communication protocols.

P2P Service Types

There are two types of Universal Messaging Peer to Peer Services:

- *Event-based Services*

  Universal Messaging Peer to Peer Event-based Services communicate via events which are published by the Event-based Client, and received and responded to by the Event-based Server Service.

- *Stream-based Services*

  Universal Messaging Peer to Peer Stream-based Services communicate via input and output streams on both the Client and Server Service. Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Peer to Peer Event-based Clients

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by a Client, and received and responded to by an Event-based Server Service.

The Universal Messaging P2P API is simple to use. There are only a very small number of objects and calls that need to be made in order for you to construct a P2P Service Client, connect to a Realm, and find or list available Services.

Creating an Event-based Service Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
String[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nServiceFactory factory = new nServiceFactory( nsa );
nServiceInfo info = factory.findService("example");
nEventService serv = (nEventService)factory.connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Sending Events to Server Services

Once you have connected to the Service, and you have an instance of the Service, you can then begin publishing your Universal Messaging events to the Service, by using the following command:

```
serv.write(new nConsumeEvent("TAG", (new UTF8Encoding()).GetBytes("Hello World")));
```

To receive responses from the Server Service, the client Service can receive events either synchronously or asynchronously via a callback interface.

Synchronously Receiving Events from the Server Service

Clients can synchronously read incoming events. The following code will return an event once one is received from the Server Service:

```
nConsumeEvent event = serv.read();
```

Asynchronously Receiving Events from the Server Service

A Client may alternatively asynchronously receive events from the Event-based Server Service by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

```
public void receivedEvent(nConsumeEvent evt) {
        Console.WriteLine("Consumed event " + event.getEventID());
}
```

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Peer to Peer Event-based Server Services

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by an Event-based Client, and received and responded to by an *Event-based Server Service*.

Creating an Event-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

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

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Event-based Server Service:

```
nServerService Server = factory.createEventService( "example",
                        "Example Event-based Service" );
while ( true ) {
        nEventService serv = (nEventService) server.accept();
        Stream inputstream = serv.getInputStream();
        Stream outputstream = serv.getOutputStream();
        // your logic goes here....
        // e.g. query a database, make a connection, send an email, etc.
        Console.WriteLine("Got connection " + serv.getServiceInfo().getName());
}
```

The code snippet above shows how to create an Event-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When connections are made to the Event-based Server Service, the Service can receive events from Clients either synchronously or asynchronously via a callback interface.

Synchronously Receiving Events from the Client

The Server Service can synchronously read incoming events. The following code will return an event once one is received from the Client:

```
nConsumeEvent event = serv.read();
```

Asynchronously Receiving Events from the Client

The Server Service may alternatively asynchronously receive events by implementing the `nEventServiceListener` interface and its `receivedEvent` method:

```
public void receivedEvent(nConsumeEvent evt) {
        Console.WriteLine("Consumed event " + event.getEventID());
}
```

You will also need to call `registerListener(your_listener_class)` on the `nEventService` object.

Sending Events to Clients

You can send events back to the Client as follows:

```
serv.write(new nConsumeEvent("TAG", (new UTF8Encoding()).GetBytes("Hello World")));
```

Peer to Peer Stream-based Clients

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the *Stream-based Client* and the Stream-based Server Service.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating a Stream-based Client

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can find our Service, or obtain a list of available Services:

```
String[] RNAME=({"nsp://127.0.0.1:9000"});
nSessionAttributes nsa = new nSessionAttributes(RNAME);
nServiceFactory factory = new nServiceFactory( nsa );
nServiceInfo info = factory.findService("example");
nEventService serv = (nEventService)factory.connectToService( info );
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Writing Client Data to a Stream-based Server Service

Once a client has connected to a Service, the client can write data to the Service. The client can obtain a reference to the Service's Output Stream object and then write to it as follows:

```
Stream oStream = serv.getOutputStream();
oStream.write((new UTF8Encoding()).GetBytes("Hello World"));
oStream.flush();
```

Receiving Responses from a Stream-based Server Service

To receive responses from the Service, the client must first obtain a reference to the Service's Input Stream object, and then read from it as follows:

```
Stream iStream = serv.getInputStream();
byte[] buff = new byte[ 100 ];
try {
  int i = 0;
  while ((i = iStream.ReadByte()) != -1)
  {
      Console.Write((char)i);
  }
} catch ( Exception ex ) {
}
```

Peer to Peer Stream-based Server Services

Universal Messaging Peer to Peer *Stream-based Services* communicate via input and output streams on both the Stream-based Client and the *Stream-based Server Service*.

Anything written to the output stream of the Stream-based Service Client is received via the input stream of the Stream-based Server Service and vice versa.

Creating an Stream-based Server Service

Firstly, in the same way that Publish/Subscribe and Message Queues use an RNAME, the P2P API also requires one to connect to the Realm. The code snippet below shows how this is achieved:

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

The `nServiceFactory` object establishes a connection with the Universal Messaging Realm, and is the factory object from which we can construct our Stream-based Server Service:

```
nServerService Server = factory.createStreamService( "example",
                        "Example Stream-based Service" );
while ( true ) {
        nEventService serv = (nEventService) server.accept();
        Stream inputstream = serv.getInputStream();
        Stream outputstream = serv.getOutputStream();
        // your logic goes here....
        // e.g. query a database, make a connection, send an email, etc.
        Console.WriteLine("Got connection " + serv.getServiceInfo().getName());
}
```

The code snippet above shows how to create an Stream-based Server Service and wait for Client connections. Developers are free to decide how the Server Service should respond once a Client connects to the Server Service.

When a connection is made to the Stream-based Server Service, the Service has an Input Stream (which can be read from), and an Output Stream (which can be written to).

Receiving Data from a Stream-based Client

The Server Service's Input Stream represents data coming from the client. The following code snippet shows how to obtain this Input Stream:

```
Stream iStream = serv.getInputStream();
```

Sending Data to a Stream-based Client

The Server Service's Output Stream represents data going to the client. The following code snippet shows how to obtain this Output Stream:

```
Stream oStream = serv.getOutputStream();
```

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.

Universal Messaging supports server-side filtering of Google Protocol Buffers, and this, coupled with Google Protocol Buffer's space-efficient serialization can be used to reduce the amount of data delivered to a client. If server side filtering is not required, the serialised protocol buffers could be loaded into a normal nConsume Event as the event data.

The structure of the data is defined in a .proto file, messages are constructed from a number of different types of fields and these fields can be required, optional or repeated. Protocol Buffers can also include other Protocol Buffers.

The serialization uses highly efficient encoding to make the serialized data as space efficient as possible, and the custom generated code for each data format allows for rapid serialization and deserialization.

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 Flex, .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.

These nProtobufEvents are integrated seamlessly in nirvana, allowing for server-side filtering of Google Protocol Buffer events, which can be sent on resources just like a normal nirvana Events. The server side filtering of messages is achieved by providing the server with a description of the data structures(constructed at the .proto compile time, using the standard protobuf compiler and the --descriptor_set_out option). The default location the sever looks in for descriptor files is /plugins/ProtobufDescriptors and this can be configured through the enterprise manager. The server will monitor this folder for changes, and the frequency of these updates can be configured through the enterprise manager. The server can then use to extract the key value pairs from the binary Protobuf message and filter message delivery based on user requirements.

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 EM is no running on the same machine as the server. To enable this, the server outputs a descriptor set to a configurable directory (by default the htdocs

directory for the realm) and this can then be made available through a file plugin etc. The directory can be changed through the enterprise manager. The enterprise manager can then be configured to load this file using -DProtobufDescSetURL and then the contents of the protocol buffers can be parsed.

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 DataStreams and DataGroups 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 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)
[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 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)
[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 rname of the server 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.

Synchronous Channel Consumer

This example shows how to synchronously consume events from a Universal Messaging Channel. See also: " Asynchronous Subscription" on page 183

Usage

```
channeliterator <rname> <channel name> [start eid] [debug] [count] [selector]
<Required Arguments>
<rname> - the rname of the server 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.

Asynchronous Named Channel Consumer

This example shows how to asynchronously subscribe to events on a Universal Messaging Channel using a named object.

Usage

```
namedsubscriber <rname> <channel name> [name] [start eid] [debug] [count] [auto ack]
                                    [cluster wide] [persistent] [selector] [priority]
<Required Arguments>
<rname> - the rname of the server to connect to
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name]         - Specifies the unique name to be used for a named subscription
                 (default: OS username)
[start eid]    - The Event ID to start subscribing from if the named subscriber
                 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 named object 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)
[selector]     - The event filter string to use
[priority]     - Whether priority is enabled for this named subscriber
                (default: false)
```

Application Source Code

See the online documentation for a code example.

Synchronous Named Channel Consumer

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

Usage

```
namedchanneliterator <rname> <channel name> [name] [start eid] [debug] [count]
                                          [cluster wide] [persistent] [selector]
<Required Arguments>
<rname> - the rname of the server to connect to
<channel name> - Channel name parameter for the channel to subscribe to
[Optional Arguments]
[name]         - Specifies the unique name to be used for a named 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 named object 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)
```

```
[selector]     - The event filter string to use
```

Application Source Code

See the online documentation for a code example.

Event Delta Delivery

This example shows how to deliver only changed keys within events, as opposed to entire events.

Usage

```
RegisteredEvent <rname> <channel name> [count] [size]
<Required Arguments>
<rname> - the 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)
[size] - The size (bytes) of the event to publish (default: 100)
```

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 rname of the server to connect to
<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 Datastreams and Datagroups

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 rname of the server to connect to
[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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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 188

Usage

```
qsubscriber <rname> <queue name> [debug] [transactional] [selecter] [count]
<Required Arguments>
<rname> - the rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
<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.

Requester - Request/Response

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

Usage

```
request <channel name>
<Required Arguments>
<request queue> - Queue onto which request are published
<response queue> - Queue onto which responses are published
<channel name> - Channel name parameter for the channel to subscribe to
<tag> - the tag to identify this requester by.
[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.

Responder - Request/Response

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

Usage

```
response <channel name>
<Required Arguments>
<request queue> - Queue onto which request are published
<response queue> - Queue onto which responses are published
<channel name> - Channel name parameter for the channel to subscribe to
[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.

MyChannels.Universal Messaging API

MyChannels.Universal Messaging DataGroup Publisher

This example shows how to create a DataGroup Publisher using the MyChannels.Universal Messaging API.

Application Source Code

See the online documentation for a code example.

MyChannels.Universal Messaging Queue Publisher

This example shows how to create a Queue Publisher using the MyChannels.Universal Messaging API.

Application Source Code

See the online documentation for a code example.

MyChannels.Universal Messaging Topic Publisher

This example shows how to create a Topic Subscriber using the MyChannels.Universal Messaging API.

Application Source Code

See the online documentation for a code example.

MyChannels.Universal Messaging DataGroup Listener

This example shows how to create a DataGroup Listener using the MyChannels.Universal Messaging API.

Application Source Code

See the online documentation for a code example.

MyChannels.Universal Messaging Queue Consumer

This example shows how to create a Queue Consumer using the MyChannels.Universal Messaging API.

Application Source Code

See the online documentation for a code example.

MyChannels.Universal Messaging Topic Subscriber

This example shows how to create a Topic Subscriber using the MyChannels.Universal Messaging 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.

Peer to Peer

An Event-based Peer to Peer Client and Server Service

This example shows how to build a simple Event-based P2P Service.

The example consists of a server and a client; the server will echo anything typed by the client.

Usage

```
nP2PEcho <rname> [server]
<Required Arguments>
<rname> - the rname of the server to connect to
[Optional Arguments]
[server] - write 'server' to run echo server, or leave out to run echo client
```

Application Source Code

See the online documentation for a code example.

A Stream-based Peer to Peer Client and Server Service

This example shows how to build a simple Stream-based P2P service.

The example consists of a server and a client; the server essentially exposes a shell to the client.

Usage

```
nP2PShell <rname> [shell]
<Required Arguments>
<rname> - the rname of the server to connect to
<Optional Arguments>
[shell] - The type of shell you want to offer. For example cmd for win32
          or bash for unix, leave out for client
```

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 rname of the server to connect 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
<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
[peak] - 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 rname of the server to connect to
<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 rname of the server to connect to
<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 rname of the server to connect to
```

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 rname of the server to connect to
[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.

Remove Service ACL

This shows how the ACL for a P2P service can be removed.

Usage

```
nremoveserviceacl <rname> <service name>
<Required Arguments>
<rname> - the rname of the server to connect to
<service name> - Service name parameter for the service to remove the ACL from
[Optional Arguments]
```

Application Source Code

See the online documentation for a code example.

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.

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);
   }
 }
}
```

Difference between 2 realms

Output all the differences between two realms.

Usage

```
nDiff <realm1> <realm2>
[Required Arguments]
<realm1> - the RNAME of a the first realm to compare
<realm2> - the RNAME of a the second realm to compare
```

Application Source Code

See the online documentation for a code example.

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 rname of the server to connect to
<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
S - For a simple (stored in memory) channel with non-persistent eids
T - For a transient (no server based storage)
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 rname of the server to connect to
<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 rname of the server to connect to
<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
S - For a simple (stored in memory) queue with non-persistent eids
T - For a transient (no server based storage)
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 rname of the server to connect to
<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 rname of the server to connect to
```

```
<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 rname of the server to connect to
<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.

Multiplex a Session

Multiplex two Universal Messaging sessions over one channel.

Usage

```
multiplex <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.

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 rname of the server to connect to
<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 rname of the server to connect to
<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 Universal Messaging Enterprise C# API. The Universal Messaging C# API is available in 2 different DLL distributions. The first is for developing Native Windows Applications (Universal Messaging DotNet.dll), and the second is compatible with Microsoft Silverlight applications (Universal Messaging Silverlight.dll). In both cases, the Client API is exactly the same.

Universal Messaging .NET

Universal Messaging .Net requires .Net version 3.5 or above. You can download .NET from the Microsoft Download website. 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.

Universal Messaging Silverlight

For a client to run a Universal Messaging Silverlight application, Microsoft Silverlight version 2.0 or above must be installed.

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 dotnet\bin directory. For native Windows applications the "Universal Messaging DotNet.dll" library is required and for Silverlight applications the "Universal Messaging Silverlight.dll" 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 dotnet\bin directory of the Universal Messaging download 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 dotnet\examples along with a batch file which can be used to compile the application:

```
> cd C:\Universal Messaging 5.0.xxxx\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

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 "Universal Messaging 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 extention is not required.

Messaging API

MyChannels.Universal Messaging API: Creating and Disposing of a Session

Creating a session is extremely simple with the C# .NET MyChannels.Universal Messaging 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.Universal Messaging 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.Universal Messaging API: Consumers

Consumers are the main means of consuming messages when using the MyChannels.Universal Messaging 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.Universal Messaging 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.

Universal Messaging.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.Universal Messaging 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.Universal Messaging 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 210
- "Prerequisites" on page 217

Publish / Subscribe

Publish/Subscribe

The Universal Messaging VBA API allows you to publish and subscribe to Universal Messaging channels using Microsoft Office products 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 RTD server, the server will be available for use in any 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("Universal MessagingRTD",,RNAME,Channel,Property,Key,Value,Key2,Value2 ...)
```

The parameters are explained below:

"Universal MessagingRTD"

This is the CLSID which has been registered for the Universal Messaging RTD server. By specifying this ID, Excel will lookup 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 216.
- `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, queue or P2P service. 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* wil 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 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

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 214, if 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 210. 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)

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 214) 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 .Net version 3.5 or above. You can download .NET 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 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 Universal MessagingExcel.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.

Enterprise Client Development

Environment Configuration

The Universal Messaging Python API uses a C++ wrapper library to expose functionality from Universal Messaging C++ 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 Universal Messaging C++ Client uses OpenSSL for secure connections. This product does not ship with Universal Messaging because some of the encryption used is restricted in certain countries.

OpenSSL comes pre-installed on most unix based systems. On Windows you can either download and build the source from www.openssl.org. Or you can download pre-compiled binaries. The required binaries for Windows are "Win32 OpenSSL v0.9.8r".

Microsoft Visual C++ 2008 Runtime Libraries

These libraries are requires to run any C++ application. Because the Universal Messaging Python Client uses Universal Messaging C++, these libraries are required. They are available to download from the Microsoft website.

Running the Sample Applications

Once you have installed Universal Messaging, the sample applications can be found in [Universal Messaging Install]/python/examples. To run the applications you first need to setup the required paths by running the Python Examples Command Prompt.

On Windows this can be found in: Start -> All Programs -> Universal Messaging_6.0.XXXXX -> Client -> [RealmName] -> Python Examples Command Prompt.

On Linux this can be found in: [Universal Messaging Install] / links / Client / [RealmName] / Python Examples Command Prompt.

Running this script will set up the environment and change to the directory containing the python samples so now to run the DataStreamListener sample you can simply enter:

```
c:\Python26\python.exe DataStreamListener.py
```

Running with a Different Python Version

The C++ wrapper which the Python API uses has to be compiled against a specific python version. In the Universal Messaging installer we currently release the wrapper compiled against Python 2.6 and 2.7. By default the sample applications will reference the library built against 2.6. To change this you can alter the file named Universal MessagingModule.py which is found in the same directory as the sample applications.

ImportError: DLL load failed

If any libraries cannot be found then you will get an error like this. Please ensure you have installed OpenSSL, Visual C++ 2008 runtime libraries and have run the Python Examples Command Prompt.

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 Universal MessagingSession) is contained within the `Universal MessagingPython` library so you must first include the library and then initiate a new Universal MessagingSession.

```
from Universal MessagingPython import *
Universal MessagingModule = Universal MessagingSession()
```

If the Universal MessagingPython 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 Universal MessagingPython 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 218)

Connecting to a Universal Messaging Realm

Once the Universal MessagingSession object has been initialised, you can connect to a Universal Messaging Realm as follows:

```
rname="nsp://localhost:9000"
Universal MessagingModule.connect(rname)
```

For information of the parameters you can pass to the connect() function e.g a user name, you can enter:

```
help(Universal MessagingModule.connect)
```

Subscribing to a Channel/Topic or Queue

In the Universal MessagingPython 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 Universal MessagingSession.subscribe method along with the Universal MessagingCallback object which will receive the asynchronous events.

Creating a Universal MessagingCallback Object

Asynchronously receiving events requires an object which implements the Universal MessagingPython.Universal MessagingCallback interface. The interface has one method, onMessage which is passed a nConsumeEvent object (see "Universal Messaging Events" on page 225).

```
class Universal MessagingCallback(Universal MessagingPython.Universal MessagingCallback):
    def onMessage(self,message):
        print "received an event"
listener = Universal MessagingCallback()
```

Registering the Universal MessagingCallback Object to Receive Events

Once the Universal MessagingCallback 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 Universal MessagingSession (see "Creating a Session" on page 219). Then you can call the Universal MessagingSession.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 = Universal MessagingSession()
mySession.connect("nsp://localhost:9000")
chanName="demochannel"
mySession.subscribe(chanName,listener)
```

Once the subscription has been registered, the onMessage method of the Universal MessagingCallback 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 Universal MessagingSession can be initialised to receive DataGroup events by passing a Universal MessagingCallback object into the connect method.

Creating a Universal MessagingCallback Object

Asynchronously receiving events requires an object which implements the Universal MessagingPython.Universal MessagingCallback interface. The interface has one method, onMessage which is passed a nConsumeEvent object (see "Universal Messaging Events" on page 225).

```
class Universal MessagingCallback(Universal MessagingPython.Universal MessagingCallback):
    def onMessage(self,message):
        print "received an event"
listener = Universal MessagingCallback()
```

Registering as a DataStream

In order to register the Universal MessagingSession as a DataStream, you simply need to pass the Universal MessagingCallback object into the connect method of Universal MessagingSession along with the RNAME (see "Creating a Session" on page 219).

```
mySession = Universal MessagingSession()
mySession.connect("nsp://localhost:9000",listener)
```

Publishing Events to a Channel or Queue

Once the Universal MessagingSession 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 Universal MessagingSession 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 Universal MessagingSession.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 Universal MessagingSession.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 Universal MessagingPython.AsyncExceptionListener interface. The interface has one method, onException which is passed a string describing the exception.

```
class AsyncExceptionListener(Universal MessagingPython.AsyncExceptionListener):
    def onException(self,message):
        print "Received an exception -> "+message
exceptionListener = AsyncExceptionListener()
```

Registering the listener for events

In order to register the Universal MessagingPython.AsyncExceptionListener to receive notification of exceptions, you can call the addAsyncExceptionListener method of Universal MessagingSession (see "Creating a Session" on page 219).

```
mySession = Universal MessagingSession()
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 = Universal MessagingModule.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 Datastreams and Datagroups

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, DataGroup or P2P service. 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(Universal MessagingPython.Universal MessagingCallback):
    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")
Universal MessagingModule.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).

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	Peer to Peer	Admin	JMS
Java	✓	✓		Native	✓	✓	✓	✓	✓
C# .NET	✓	✓		Native	✓	✓	✓	✓	
C++	✓			Native	✓	✓	✓	✓	
Python	✓			Native	✓	✓			
Excel VBA	✓			Native	✓				
JavaScript		✓		Native (via WebSocket) or Comet	✓	✓			
Adobe Flex		✓		Native	✓	✓	✓		

	Target Environments			Communicatior Protocols	Messaging Paradigms			Extended APIs	
	Enter-prise	Web	Mobile	Native or Comet	Pub/ Sub	Msg. Queue:	Peer to Peer	Admin	JMS
Microsoft Silverlight	○	✓	○	Native	✓	✓	✓	○	○
iPhone	○	○	✓	Native	✓	✓	✓	○	○
Android	○	○	✓	Native	✓	✓	✓	○	○

2 Mobile Client APIs

- Client API for iPhone 230
- Client API for Android 231

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 Android:

- *Apple iPhone*

  Our *Universal Messaging iPhone API* 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:

  - "Universal Messaging iPhone Developer's Guide" on page 230

  Please contact us for a live demonstration, or for access to our Universal Messaging for Apple iPhone API.

- Android

  Android devices are able to make use of our *Universal Messaging Enterprise API for Java* to subscribe to Universal Messaging channels, utilize message queues, and communicate with peer to peer services.

  Our "Android Developer's Guide" on page 231 provides further information, online demonstrations and sample source code.

See Universal Messaging's Language API Comparison Grid for an overview of basic differences between each API.

Client API for iPhone

iPhone Developer's Guide

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. The Universal Messaging iPhone download available above includes all the required libraries, header files and full source for our TradeSpace implementation on the iPhone.

Client API for Android

Android Developer's Guide

Universal Messaging for 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.

We have provided a sample Android application, TradeSpace, along with full source code to get you started writing your own Android applications with Universal Messaging.

Documentation

The Universal Messaging Enterprise Developer's Guide for Java provides full information on how to use *pub/sub*, *message queues* and *peer to peer services* in your Android application.

3 Web Client APIs

- Overview of Web Client APIs 234
- Web Developer's Guide for Javascript 235
- Web Developer's Guide for Adobe Flex 247
- Web Developer's Guide for Silverlight 266
- Web Developer's Guide for Java 275

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*, *Java* or *Adobe Flex*:

- **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 235

  Our JavaScript API is popular because it works without plugins or infrastructure workarounds, using only the browser's built-in JavaScript engine.

- **Adobe Flex**

  The *Universal Messaging Adobe Flex API* is an Adobe ActionScript API allowing the rapid development of publish/subscribe RIA clients. These clients can be run within a browser, or as standalone Adobe AIR applications:

  - "Flex Developer's Guide" on page 247

- **Microsoft Silverlight**

  The *Universal Messaging Silverlight API* is an C# .NET API allowing the rapid development of publish/subscribe RIA clients. These clients can be run within a browser:

  - "Silverlight Developer's Guide" on page 266

- **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 275

  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: Note that since version 7, 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 `/lib/javascript` in the nirvana base installation directory. 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: Note that since version 7, 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: Note that since version 7, 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

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.

Publish/Subscribe Tasks

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 239 and "Publishing Events to a Channel" on page 240.

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);
```

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.

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.

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.

Message Queue Tasks

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 245 and "Publishing Events to a Queue" on page 245.

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.

Web Developer's Guide for Adobe Flex

Overview

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

Publish / Subscribe using Channel Topics

Publish / Subscribe

The Universal Messaging Flex 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.

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, 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 (nEventProperties) for our Universal Messaging Event before publishing it:

```
var event : nConsumeEvent = new nConsumeEvent();
var dictionary : nEventProperties = new nEventProperties();
dictionary.put("exampleKey", "Hello World");
event.properties=dictionary;
channel.publish(evt);
```

The final item to note is that the publish call is asynchronous; it returns immediately, allowing single-threaded Flex clients to continue processing.

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:

```
var event : nConsumeEvent = new nConsumeEvent();
var dictionary : nEventProperties = new nEventProperties();
dictionary.put("exampleKey", "Hello World");
event.properties=dictionary;
var tattrib:nTransactionAttributes = new nTransasctionAttributes(myChannel);
var myTransaction:nTransaction =
   nTransactionFactory.create(tattrib, transactionCallBack);
myTransaction.publish(event);
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:

```
var committed:Boolean = myTransaction.isCommitted(isCommittedCallBack);
```

Subscribing to a Channel

Once a Universal Messaging channel (nChannel) has been found, you can subscribe to the channel, and receive Universal Messaging Events published on the channel.

Simple Subscription

This Flex code snippet demonstrates how to subscribe to a channel:

```
var startEventID : Long = Long.ZERO;
channel.addSubscriber(nEventListener, startEID,
                      subscriptionCompleteCallbackFunc);
```

Note that the addSubscriberFromEID call is asynchronous; it returns immediately, allowing single-threaded Flex clients to continue processing.

To let the developer know when the subscription request has actually completed, the addSubscriber function takes three parameters:

1. nEventListener : the implemented listener that will be called whenever a Universal Messaging Event is received on the channel. The event (nConsumeEvent) object will be passed to this callback function as a parameter.
2. startEID : the ID of the event from which the subscription should start. We use 0 in this example to ensure we receive all available events.
3. postSubCB : the name of a developer-defined Flex function that will be called immediately after the subscription request actually completes.

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 Flex 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 startEventID : int = 0;
var selector : String = "volatility > 0.5"
channel.addSubscriber(nEventListener, startEID,
                      subscriptionCompleteCallbackFunc,selector);
```

Handling Received Events

As discussed above, you must implement a nEventListener to handle any received events. Below is an example of an implemented function from the listener.

```
public function go(event:nConsumeEvent){
```

```
    var dictionary:nEventProperties = event.properties;
    var sender:String = dictionary.get("sender").toString();
    var message:String = dictionary.get("message").toString()
}
```

Durable channel consumers and named objects

Universal Messaging provides the ability for asynchronous consumers to be durable. Durable consumers allow state to be kept at the server with regard to what events have been consumed by a specific consumer of data.

Universal Messaging supports durable consumers through use of Universal Messaging named objects as shown by the following example code.

Names objects can also be managed via the enterprise manager.

Durable Consumer

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

```
public class mySubscriber implements nEventListener {
// construct your session
// and channel objects here
  private var nobj:nNamedObject;
  public function mySubscriber(channelIn:*):void {
    if(channelIn is nChannel){
        myChannel.createNamedObject(namedObjectCreatedCB, "unique1", Long.ZERO, true);
        }
  }
  public function namedObjectCreatedCB(named:*):void {
    if(named is nNamedObject){
          nobj= nNamedObject(named)
          myChannel.addSubscriber(this , subscribedCB, nobj,true);
        }
  }
  public function go(event:*):void {
    if(event is nConsumeEvent){
          //process event
    }
  }
}
```

Channel consumers allow message selectors to be used in conjunction with named objects. 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 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 function go(event:*):void {
    if(event is nConsumeEvent){
          //process event
          nConsumeEvent(event).ack();
    }
  }
```

Priority

Two subscribers can hold a subscription to the same named object. One is given priority and will process events during normal operation. If, however, the subscriber with priority is disconnected for whatever reason, and is unable to process events, the second subscriber to that named object will take over and continue to process events as they come in. This allows failover, with backup subscribers handling events if the subscriber with priority goes down.

To do this, we simply create the subscriber with a boolean specifying if this subscriber priority. Only one subscriber is allowed priority at any given time. An example of a named object specifying priority is shown below:

```
myChannel.createNamedObject(namedObjectCreatedCB, subname, startEid, persistent,
                            cluster, priority);
```

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 over the wire 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:

```
//
//Firstly, create the ChannelAttributes for a simple channel with unlimited capacity
// and no TTL.
//
var cattr:nChannelAttributes = new nChannelAttributes();
cattr.name = "RatesChannel";
cattr.maxEvents=0;
cattr.TTL=Long.ZERO;
cattr.type = 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
//
var keys:Array = new Array();
keys.push(new nChannelPublishKeys("ccy",1));
cattr.publishKeys = keys;
```

```
//
// Now create the channel
//
_session.createStore(cattr,callbackWithChannel, Long.ZERO);
```

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.

```
var audEvent:nRegisteredEvent = (e as nChannel).createRegisteredEvent("AUD");
var props:nEventProperties = audEvent.properties;
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", 0.8999);
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.getEventAttributes("isDelta")
```

For more information on Universal Messaging publish / subscribe, please see the API documentation.

Publish / Subscribe using Datastreams and Datagroups

Publish / Subscribe

The Universal Messaging Flex 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.

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.sDropEvents
nConflationAttributes.sMergeEvents
```

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
var conflattribs : nConflationAttributes =
    new nConflationAttributes(Long.ZERO, nConflationAttributes.sMergeEvent);
//ConflationAttributes specifying merge events and throttled delivery at
// 1 second intervals
var conflattribs : nConflationAttributes =
    new nConflationAttributes(new Long(0,1000), nConflationAttributes.sMergeEvent);
//ConflationAttributes specifying drop events and throttled delivery at
// 1 second intervals
var conflattribs : nConflationAttributes =
    new nConflationAttributes(new Long(0,1000), nConflationAttributes.sDropEvent);
```

Create a Single nDataGroup with Conflation Attributes

```
var conflattribs : nConflationAttributes =
    new nConflationAttributes(Long.ZERO, nConflationAttributes.sMergeEvent);
// Create a Datagroup, passing in a Listener of type nDataGroupListener and
// a nConflationAttributes
mySession.createDataGroup("myGroup", dataGroupCallback, dataGroupListener,
                          conflattribs);
```

Create Multiple nDataGroups with Conflation Attributes

```
var conflattribs : nConflationAttributes =
    new nConflationAttributes(0, nConflationAttributes.sMergeEvent);
var groups : Array = {"myFirstGroup", "mySecondGroup"};
mySession.createDataGroups(groups, dataGroupCallback, conflattribs);
```

Now we will get a call back to dataGroupsCallback and have a reference to the Universal Messaging datagroup(s) within the realm or an error.

Publishing Events to Conflated DataGroups With A Merge Policy

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

```
var audEvent : nRegisteredEvent = dataGroupCallback.createRegisteredEvent();
var props : nEventProperties = audEvent.properties;
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.

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
{
    var mySession:nSession;
    public function DataGroupClient(realmURLs:String, retryAttempts:int)
    {
        var nsa:nSessionAttributes = new nSessionAttributes(realmURLs, retryAttempts);
        mySession = nSessionFactory.create(nsa, username, appName, errorCB);
        mySession.init(sessionInitCallback, null, false, this);
    }
    ////
    // nDataStreamListener Implementation
    ////
    //Callback received when event is available
    public function onMessage(event:nConsumeEvent):void
    {
        //some code to process the message
    }
    public function sessionInitCallback(e:*){
        // do something with the session if needed
    }
    public function errorCB(e:*}{
        //Error handling function
    }
}
```

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) 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.

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

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, a new Universal Messaging Event object (nConsumeEvent) must be constructed prior to use in the publish call being made to the queue.

Note that in this example code, we also create a Universal Messaging Event Dictionary object (nEventProperties) for our Universal Messaging Event before publishing it:

```
var event : nConsumeEvent = new nConsumeEvent();
var dictionary : nEventProperties = new nEventProperties();
dictionary.put("exampleKey", "Hello World");
event.properties=dictionary;
myQueue.push(evt);
```

The final item to note is that the publish call is asynchronous; it returns immediately, allowing single-threaded Flex clients to continue processing.

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 queue 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 queue.

Below is a code snippet for transactional publishing:

```
var event : nConsumeEvent = new nConsumeEvent();
var dictionary : nEventProperties = new nEventProperties();
dictionary.put("exampleKey", "Hello World");
event.properties=dictionary;
var tattrib:nTransactionAttributes = new nTransasctionAttributes(myQueue,0);
var myTransaction:nTransaction = nTransactionFactory.create(tattrib,
       transactionCallBack);
myTransaction.push(event);
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:

```
var committed:Boolean = myTransaction.isCommitted(isCommittedCallBack);
```

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 {
        var myQueue:nQueue = null;
        var reader:nQueueAsynchronousReader = null;
        // construct your session and queue objects here
        public function myAsyncQueueReader(queueIn:*):void{
                if(queueIn is nQueue){
                        myQueue = nQueue(myQueue);
                }else{
                        return;
                }
        // begin consuming events from the queue
                var ctx:nQueueReaderContext = new nQueueReaderContext(this);
                myQueue.createAsynchronousReader(ctx, readerCB);
        }
        public function readerCB(readerIn:*):void {
                if(reader is nQueueAsynchronousReader){
                        reader = readerIn
                }
        }
        public function go(event:*):void {
                if(event is nConsumeEvent){
                        //Process events
                }else{
                        //deal with error
                }
        }
 }
```

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:

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

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

```
    var selector:String = "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 {
        var myQueue:nQueue = null;
        var reader:nQueueAsynchronousTransactionalReader = null;
        // construct your session and queue objects here
        public function myAsyncQueueReader(queueIn:*):void{
                if(queueIn is nQueue){
                        myQueue = nQueue(myQueue);
                }else{
                        return;
                }
                // begin consuming events from the queue
                var ctx:nQueueReaderContext = new nQueueReaderContext(this);
                myQueue.createAsynchronousTransactionalReader(ctx, readerCB);
        }
        public function readerCB(readerIn:*):void {
                if(reader is nQueueAsynchronousTransactionalReader){
                        reader = readerIn
                }
        }
        public function go(event:*):void {
                if(event is nConsumeEvent){
                        //Process events
                        reader.commit(nConsumeEvent(event).eventID)
                }else{
                        //deal with error
                }
        }
 }
```

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.

Asychronous 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:

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

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

```
var selector:String = "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.

Peer to Peer

Peer to Peer Services

Universal Messaging provides a rich set of APIs that provide developers with the ability to create Peer to Peer (P2P) applications. We call these Peer to Peer applications *Services*. This guide will demonstrate how Universal Messaging Peer to Peer Services work, and provides examples code snippets for all relevant concepts.

P2P Service Components

There are two parts to a Peer to Peer Service in Universal Messaging: a *Server Service* and a *Client*, in Flex only the client portion is available. To see more information about the Server service see either the Java , C# or C++.

A Universal Messaging Peer to Peer *Client* is a process that connects to a Universal Messaging Realm, obtains a reference to a Server Service and begins communicating with it.

When a Client connects to the Server Service, all communication between the Client and server service takes place through the Universal Messaging Realm, using Universal Messaging's standard communication protocols.

P2P Service Types

There are two types of Universal Messaging Peer to Peer Services, Flex only supports Event Based Services:

- *Event-based Services*

  Universal Messaging Peer to Peer Event-based Services communicate via events which are published by the Event-based Client, and received and responded to by the Event-based Server Service.

Examples

The following examples give methods needed to implement a client to work with Java, C# or C++ Server Service:

- "An Event-based Peer to Peer Client" on page 262

For more information on Universal Messaging Peer to Peer Services please see the API documentation.

Peer to Peer Event-based Client

Universal Messaging Peer to Peer *Event-based Services* communicate via events which are published by a Client, and received and responded to by an Event-based Server Service.

The Universal Messaging P2P API is simple to use. There are only a very small number of objects and calls that need to be made in order for you to construct a P2P Service Client, connect to a Realm, and find or list available Services.

Creating an Event-based Service Client

The `nServiceFactory` object takes a connected nirvana session and a callback for when the service is connected:

```
private var factory:nServiceFactory
private function createP2P():void{
    factory = new nServiceFactory( mySession, serviceFactoryCB );
}
```

When the nServiceFactory calls the callback then we can use factory to findServices and then connect to a Service. findServices takes a name of a service and connectToService takes the nServiceInfo, a nEventListener and a connected callback. The following snippets shows how this would be done:

```
private function serviceFactoryCB():void{
    var info:nServiceInfo = factory.findService("example");
    factory.connectToService(info,this,connectServiceCB);
}
```

Once the Client has connected to an instance of a Server Service, the developer's custom business logic can then be applied.

Sending Events to Server Services

Once you have connected to the Service, and you have an instance of the Service, you can then begin publishing your Universal Messaging events to the Service, by using the following command:

```
serv.write(new nConsumeEvent("TAG",null, byteArray));
```

The Client Service can receive events from the Server Service asynchronously via a callback interface.

Asynchronously Receiving Events from the Server Service

A Client MUST asynchronously receive events from the Event-based Server Service by implementing the `nEventServiceListener` interface and its `go` method:

```
public void go( event:nConsumeEvent) {
        trace("Event ID: "+event.eventID);
}
```

Examples

The following full example source code shows how to implement an Event-based Client:

- "An Event-based Peer to Peer Client" on page 262

For more information on Universal Messaging Peer to Peer Services please see the API documentation.

Flex Socket SSL

Flex socket SSL

Universal Messaging Flex has a custom SSL implementation for sockets, it does not support self signed certificates and only trusts the certificates included in the Mozilla project Root CA store list.

If you wish for client certificate verification then the Universal Messaging Flex implementation requires the private certificate and the signed key in pem format. The following snippet shows how you can pass a client certificate into the Universal Messaging API

```
var cert:String; //Either read in or embed the certificate string into cert
var key:String; //Either read in or embed the key string into key
var attributes:nSessionAttributes = new nSessionAttributes(completeString, 5);
attributes.sslCertificate = cert;
attributes.sslKey = key;
mySession = nSessionFactory.create(attributes, "subject", appName, errorCB);
```

SSL Realm Configuration

The Universal Messaging Flex API is able to use all 4 of Universal Messaging's protocol's natively. To enable SSL protocols, a number of steps must be followed to ensure the Universal Messaging realm, and your Flex application, are configured correctly.

- An NHP interface on port 80 should be setup, with a file plug-in at root level which points to a directory containing the crossdomain.xml for the server.
- A similar NHPS interface should be set up on port 443, again which a root level file plug-in pointing to a directory containing the crossdomain.xml.
- An nsp interface should be setup on port 843, this should have "Enable Policy File" Enabled through the "Basic" tab in the interface configuration. The policy file(clientaccesspolicy.xml) should be placed in the htdocs directory of the realm. Then add the certificates that you wish to use. These cannot be self signed, but can be from CAcert.
- Client certificate validation can be enforced through the configuration on the interface through which you will be serving your flex application.
- Pass the certificates into flex as shown above, or copy them straight in as strings, a private certificate and a signed key are required.

Examples

Sample Socket Cross Domain Policy

Adobe Flash requires a clientaccesspolicy.xml file to be avaliable on port 843 if it is providing information via a socket stream to a flash application.

Note this is an example cross domain policy only it is not secure, use for testing purposes only.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
        <site-control permitted-cross-domain-policies="all"/>
        <allow-access-from domain="*" to-ports="80,443" />
</cross-domain-policy>
```

Sample Flash Cross Domain Policy

Adobe Flash requires a crossdomaim.xml file to be in the root directory of a web server that is providing information via a http stream to a flash application.

Note this is an example cross domain policy only it is not secure, use for testing purposes only.

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
          SYSTEM "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
        <site-control permitted-cross-domain-policies="all"/>
        <allow-access-from domain="*" to-ports="80,443" />
        <allow-http-request-headers-from domain="*" headers="*" />
</cross-domain-policy>
```

Flex Example : Peer to Peer Echo Application

A Sample Adobe Flex Peer to Peer Echo Client

The code shown below is a exert of the echo client, it is the Universal Messaging portion of a echo application.

```
import com.pcbsys.nirvana.client.*;
import com.pcbsys.nirvana.client.p2p.*;
private var mySession:nSession;
private var factory:nServiceFactory;
private var myNick:String = "";
private var myService:nEventService;
private function startTest():void
{
    var completeString:String = "nhp://127.0.0.1:80";
    var appName:String = "Universal MessagingFlexP2PEchoClient";
    try {
```

```
        var attributes:nSessionAttributes =
            new nSessionAttributes(completeString, 5);
        mySession = nSessionFactory.create(attributes, "subject", appName, errorCB);
        mySession.init(sessionInitCB, this);
    }
    catch(e:SecurityError) {
    }
}
public function errorCB(error:Error, brokenFunction:Function):void
{
    trace(error.message);
}
public function sessionInitCB():void {
    var tmpId:String = mySession.getSessionId().toString(10);
    tmpId = tmpId.substr(tmpId.length - 5);
    myNick = "Flex-Native" + tmpId;
    factory = new nServiceFactory(mySession,serviceFactoryCB);
}
public function serviceFactoryCB():void
{
    var info:nServiceInfo = factory.findService("echo");
    factory.connectToService(info,this,connectServiceCB)
}
public function connectServiceCB(service:nEventService):void
{
    myService= service;
    var props:nEventProperties = new nEventProperties();
    var byteArray:ByteArray = new ByteArray();
    byteArray.writeUTF("This is a test");
    byteArray.position=0;
    myService.write(new nConsumeEvent("",null,byteArray));
}
public function retry(failureCount:int, realmName:String):Boolean {
    return false;
}
public function disconnected():void {
}
public function reconnected():void {
}
public function go(event:nConsumeEvent):void {
     trace("Event ID: "+event.eventID);
```

Flex Example : Chat Application

A Sample Adobe Flex Chat Client

The code shown below is a exert of the chat client, containing Flex connection, publishing and subscription logic.

```
/*
    Copyright 2012 Software AG, Darmstadt, Germany and/or Software AG USA
    Inc., Reston, United States of America, and/or their licensors.
    In the event that you should download or otherwise use this software
    you hereby acknowledge and agree to the terms at
    http://um.terracotta.org/company/terms.html#legalnotices
*/
import com.pcbsys.nirvana.client.*;
import com.pcbsys.foundation.util.Long;
var mySession:nSession;
var chatChannel:nChannel = null;
private var myNick:String = "";
/* ***************************************************************
```

```
 * About this demonstration Universal Messaging Client Application:
 *
 * When this application component is created, Flex will invoke the
 * startDemo() function. startDemo() calls createUniversal MessagingSession() which
 * in turn calls mySession.init(). One of the parameters passed to
 * mySession.init() is the name of the callback function to be invoked
 * after a successful session initialization (in this case, we have
 * named this callback function "sessionInitCB").
 *
 * In turn, sessionInitCB calls mySession.findStore(), again passing
 * the name of a callback function to be invoked after findStore
 * completes (in this case, "chatChannelFoundCB").
 *
 * Similarly, chatChannelFoundCB() calls channel.addSubscriber()
 * passing in the two callback functions - one to be invoked
 * when subscription to the channel is successful ("postPubCB"), and
 * the other to be invoked every time an event is received on the
 * channel ("evtCB"). Note that although the developer is free to name
 * these callback functions, the functions themselves must accept the
 * parameters shown here; for example, the callback function we have
 * named evtCB will always receive an nConsumeEvent object as a
 * parameter.
 * ****************************************************************/
private function startDemo():void {
    showUIMessage("Initializing Session...");
    createUniversal MessagingSession();
}
private function createUniversal MessagingSession():void {
    var RNAME:String = "nhp://localhost:80";
    var appName:String = "Universal MessagingFlexSimpleChatRoomDemo";
    var tmpId:String = mySession.getSessionId().toString(10);
    tmpId = tmpId.substr(tmpId.length - 5);
    myNick = "Flex-Native" + tmpId;
    try {
        // Create a Universal Messaging session attribute to be passed to the nSession
        var attributes:nSessionAttributes = new nSessionAttributes(RNAME, 5);
        // Create a Universal Messaging Session Object (nSession):
        mySession = nSessionFactory.create(attributes, "username", appName, errorCB);
        // Now, start our session. Here we specify the callback for
        // when initiation has completed, we also specify a nConnectionListener
        // in this case it has been implemented in this class thus "this".
        mySession.init(sessionInitCB, this);
    }
    catch(e:SecurityError) {
    }
}
/* ****************************************************************
 * Error callback
 *
 * This function is called if a error is thrown from the server
 *
 * ****************************************************************/
private function errorCB(error:Error, failedFunction:Function):void {
    //trace(error.message);
}
/* ****************************************************************
 * Session Init callback
 *
 * This function is called once the session has initialised, it looks
 * for the channel that the chat is on
 *
 * ****************************************************************/
public function sessionInitCB():void {
```

```
    mySession.findStore("flex", chatChannelFoundCB);
}
/* ****************************************************************
 * Session Callback Functions
 *
 * These are the implementations of nConnectionListener.
 *
 * ****************************************************************/
public function disconnected():void {
    showUIMessage("Disconnected. Reconnecting");
}
public function reconnected():void {
    hideUIMessage();
}
public function retry(failureCount:int, realmName:String):Boolean {
    return false;
}
/* ****************************************************************
 * Chat Channel Callback Functions:
 *
 * These three callback functions are kicked off sequentially, starting
 * as a response to the mySession.findStore call above.
 *
 * chatChannelFoundCB is the function called in response to a findChannel
 *  request will receive a parameters - an nChannel object.
 *
 * postSubCB is the function called in response to a successful subscription
 * to a channel.
 *
 * go the nEventListener implementation, this function is called when an
 * event is received will receive an nConsumeEvent object as its parameter.
 *
 * ****************************************************************/
public function chatChannelFoundCB(channel:nChannel):void {
    chatChannel = channel;
    channel.addSubscriber(this, Long.ZERO, postSubCB);
}
public function postSubCB():void {
    hideUIMessage();
}
public function go(evt:nConsumeEvent):void {
    var dictionary:nEventProperties = evt.properties;
    var sender:String = dictionary.get("sender").toString();
    var message:String = dictionary.get("message").toString();
    var stime:String = evt.attributes.timestamp.toString();
    var num:Number = new Number(stime);
    var dt:Date = new Date();
    dt.setTime(num);
    var when:String = dt.toLocaleTimeString();
    messages.htmlText += "[" + when + "] " + sender + " : " + message + "\n";
    messages.validateNow();
    messages.verticalScrollPosition = messages.maxVerticalScrollPosition;
}
 /* ****************************************************************
  *  This function publishes a message to the channel that we are
  *  connected to.
  *
  *  In this instance it is a chatChannel so the message has been packed
   * accordingly.
  * ****************************************************************/
public function publishMessage():void {
    var dict:nEventProperties = new nEventProperties();
    dict.put("message", messageInput.text);
```

```
    dict.put("sender", myNick);
    var evt:nConsumeEvent = new nConsumeEvent();
    evt.properties = dict;
    chatChannel.publish(evt);
}
/* ****************************************************************
 * UI & Utility Functions:
 * ****************************************************************/
public function showUIMessage(msg:String):void {
    progressbar.x = (this.width / 2) - (progressbar.width / 2);
    progressbar.y = (this.height / 2) - (progressbar.height / 2);
    progressbar.label = msg;
    progressbar.visible = true;
    uioverlay.alpha = .7;
    uioverlay.visible = true;
}
public function hideUIMessage():void {
    progressbar.visible = false;
    uioverlay.alpha = 0;
    uioverlay.visible = false;
}
```

Web Developer's Guide for Silverlight

Developer's Guide for Silverlight

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

Universal Messaging Web Client Development in Microsoft Silverlight

Please refer to the Universal Messaging C# Developer's Guide for more information on developing Silverlight clients.

Silverlight Deployment

We provide a separate 'Tradespace Demos' download that contains a fully working Silverlight application called Tradespace. When you follow the instructions, you are able to load the Silverlight Tradespace demo application from your Universal Messaging realm.

The setup of the demos performs the following actions:

- Adds an http interface on port 8080
- Adds a file plugin to your this interface (this enables the Universal Messaging server to act as a web server and serve content to browsers)
- Creates the required channels in the realm used by the Tradespace demos.

Running through the demos setup is the simplest way to get started with Universal Messaging and deployment of a Silverlight application - in this case the Tradespace

Silverlight demo. You can of course go through the individual steps described above yourself and deploy your own Silverlight application.

Channel ACLs

When creating a Silverlight application, it is worth remembering to correctly set the ACLs for both the realm(s) and channel(s) used, to ensure that the application is able to access the Universal Messaging server. This can easily be performed using the Enterprise Manager.

Silverlight's Client Access Policy File

When deploying a Silverlight application on a different host to the Universal Messaging server, Silverlight requires the use of a client access policy file to validate that the connection is permitted. For example :

- you deploy your Silverlight application from webhost1.yourdomain.com:80
- your Universal Messaging server is located on nirvanahost1.yourdomain.com:80

When Silverlight detects a connect attempt outside of the host from which the application was downloaded, it makes a request for the policy file from the host you are making the connection to. When using an nhp (HTTP) or nhps (HTTPS) RNAME to connect to the Universal Messaging realm, Silverlight makes a GET request for a clientaccesspolicy.xml file from the root of the web server (in this case, the file plugin running on the Universal Messaging server HTTP or HTTPS interface). If no clientaccesspolicy.xml is found, it then makes a second GET for a crossdomain.xml file, again from the root of the file plugin. The crossdomain.xml file is also used by flex clients for the same reason.

An example of a clientaccesspolicy.xml file for Silverlight clients is shown below. This example should not be used for anything other than testing purposes, as it essentially allows open access to and from all domains. For more information on cross domain access with Silverlight, and configuring the clientaccesspolicy.xml file, see the Microsoft MSDN guide.

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
    <cross-domain-access>
            <policy>
                    <allow-from http-request-headers="*">
                            <domain uri="*"/>
                        </allow-from>
                        <grant-to>
                            <resource path="/" include-subpaths="true"/>
                        </grant-to>
                </policy>
        </cross-domain-access>
</access-policy>
```

Examples

Live Stock Chart

This example demonstrates how to subscribe to a Universal Messaging channel and chart prices received in real-time events.

Application Source Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using com.pcbsys.nirvana.client;
using Visifire.Charts;
namespace Silverlight_LiveStockChart
{
    public partial class Page : UserControl,nEventListener,nReconnectHandler
    {
        private bool started;
        public nSession mySession;
        public Thread sessionThread;
        private long myEventCount=0;
        private const string RNAME = "nhps://showcase.um.softwareag.com:443";
        private const string RATES_CHANNEL = "/showcase/stockhistory";
        public Page()
        {
            InitializeComponent();
            CreateChart();
            StartupProgressDialog.IsOpen = true;
            sessionThread = new Thread(new ThreadStart(startSubscribers));
            sessionThread.IsBackground = true;
            sessionThread.Start();
            App.Current.Host.Content.Resized += (s, e) =>
            {
                theBack.Width = App.Current.Host.Content.ActualWidth;
                theBack.Height = App.Current.Host.Content.ActualHeight;
            };
        }
        public void disconnected(nSession anSession)
        {
            StartupProgressDialog.Dispatcher.BeginInvoke(new
              setProgressBarMessage(updateStatusMessage), "Disconnected...");
            StartupProgressDialog.Dispatcher.BeginInvoke(new
              setOverlayPanelVisibleDelegate(setOverlayPanelVisible), true);
            Console.WriteLine("Disconnected");
        }
        public void reconnected(nSession anSession)
        {
            StartupProgressDialog.Dispatcher.BeginInvoke(new
              setOverlayPanelVisibleDelegate(setOverlayPanelVisible), false);
```

```
    Console.WriteLine("Reconnected");
}
public bool tryAgain(nSession anSession)
{
    return true;
}
public void go(nConsumeEvent evt)
{
    if (evt.getChannelName().Equals(RATES_CHANNEL))
    {
        myEventCount++;
        if (myEventCount>=100)
        {
            StartupProgressDialog.Dispatcher.BeginInvoke(new
              setOverlayPanelVisibleDelegate(setOverlayPanelVisible),
              false);
        }
        nEventProperties nep = evt.getProperties();
        nEventAttributes nea = evt.getAttributes();
        long tval = nea.getTimestamp();
        DateTime ttime = ConvertJavaMiliSecondToDateTime(tval);
        myChart.Dispatcher.BeginInvoke(new
           RatesDataDelegate(updateRatesGrid), ttime.ToShortTimeString(),
           nep.get("value").ToString());
        return;
    }
}
public DateTime ConvertJavaMiliSecondToDateTime(long javaMS)
{
    DateTime UTCBaseTime = new DateTime(1970, 1, 1, 0, 0, 0,
         DateTimeKind.Utc);
    DateTime dt = UTCBaseTime.Add(new TimeSpan(javaMS *
    TimeSpan.TicksPerMillisecond)).ToLocalTime();
    return dt;
}
public delegate void RatesDataDelegate(String index, String ival);
private void updateRatesGrid(String time, String ival)
{
    DataPoint dataPoint = new DataPoint();
    // Set YValue for a DataPoint
    dataPoint.YValue = Double.Parse(ival);
    dataPoint.AxisXLabel = time;
    // Add dataPoint to DataPoints collection.
    myChart.Series[0].DataPoints.Add(dataPoint);
}
public delegate void setProgressBarMessage(String message);
public void updateStatusMessage(String message)
{
    myStatusMessage.Text = message;
}
public delegate void setOverlayPanelVisibleDelegate(Boolean flag);
public void setOverlayPanelVisible(Boolean flag)
{
    StartupProgressDialog.IsOpen = flag;
}
public void startSubscribers()
{
    if (!started)
    {
        try
        {
            nSessionAttributes nsa = new nSessionAttributes(RNAME, 5);
            mySession = nSessionFactory.create(nsa, this, "SilverDemoUser");
```

```
                    mySession.init();
                    StartupProgressDialog.Dispatcher.BeginInvoke(new
                       setProgressBarMessage(updateStatusMessage),
                       "Subscribing to Rates...");
                    nChannelAttributes ncaindices = new nChannelAttributes();
                    ncaindices.setName(RATES_CHANNEL);
                    nChannel myRatesChannel = mySession.findChannel(ncaindices);
                    myRatesChannel.addSubscriber(this, 0);
                    StartupProgressDialog.Dispatcher.BeginInvoke(new
                       setProgressBarMessage(updateStatusMessage),
                       "Waiting for 100 events to create chart...");
                }
                catch (Exception e)
                {
                    Console.WriteLine("Error starting subscribers: " + e.Message);
                    Console.WriteLine(e.StackTrace);
                }
                started = true;
            }
        }
        /// <summary>
        /// Function to create a Visifire Chart
        /// </summary>
        public void CreateChart()
        {
            // Create a new instance of Title
            Title title = new Title();
            // Set title property
            title.Text = "NSL PLC.";
            // Add title to Titles collection
            myChart.Titles.Add(title);
        }
    }
}
```

Live Stock Indices

This example demonstrates how to subscribe to a Universal Messaging channel and render prices received in real-time events.

Application Source Code

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Net;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using com.pcbsys.nirvana.client;
namespace Silverlight_LiveStockIndices
{
    public partial class Page : UserControl,nReconnectHandler,nEventListener
    {
        private bool started;
        public nSession mySession;
```

```
private ObservableCollection<RatesData> myRatesDataListSource =
   new ObservableCollection<RatesData>();
public Thread sessionThread;
private const string RNAME = "nhps://showcase.um.softwareag.com:443";
private const string RATES_CHANNEL = "/showcase/stockindices";
public Page()
{
    InitializeComponent();
    StartupProgressDialog.IsOpen = true;
    myIndexGrid.ItemsSource = this.myRatesDataListSource;
    myIndexGrid.IsReadOnly = true;
    App.Current.Host.Content.Resized += (s, e) =>
    {
        theBack.Width = App.Current.Host.Content.ActualWidth;
        theBack.Height = App.Current.Host.Content.ActualHeight;
    };
    sessionThread = new Thread(new ThreadStart(startSubscribers));
    sessionThread.IsBackground = true;
    sessionThread.Start();
}
public void startSubscribers()
{
    if (!started)
    {
        try
        {
            nSessionAttributes nsa = new nSessionAttributes(RNAME, 5);
            mySession = nSessionFactory.create(nsa, this, "SilverDemoUser");
            mySession.init();
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setProgressBarMessage(updateStatusMessage),
               "Subscribing to Rates...");
            nChannelAttributes ncaindices = new nChannelAttributes();
            ncaindices.setName(RATES_CHANNEL);
            nChannel myRatesChannel = mySession.findChannel(ncaindices);
            myRatesChannel.addSubscriber(this, 0);
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setOverlayPanelVisibleDelegate(setOverlayPanelVisible),
               false);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error starting subscribers: " + e.Message);
            Console.WriteLine(e.StackTrace);
        }
        started = true;
    }
}
public delegate void setProgressBarMessage(String message);
public void updateStatusMessage(String message)
{
    myStatusMessage.Text = message;
}
public delegate void setOverlayPanelVisibleDelegate(Boolean flag);
public void setOverlayPanelVisible(Boolean flag)
{
    StartupProgressDialog.IsOpen = flag;
}
public void disconnected(nSession anSession)
{
    StartupProgressDialog.Dispatcher.BeginInvoke(new
       setProgressBarMessage(updateStatusMessage), "Disconnected...");
    StartupProgressDialog.Dispatcher.BeginInvoke(new
```

```
            setOverlayPanelVisibleDelegate(setOverlayPanelVisible), true);
        Console.WriteLine("Disconnected");
    }
    public void reconnected(nSession anSession)
    {
        StartupProgressDialog.Dispatcher.BeginInvoke(new
            setOverlayPanelVisibleDelegate(setOverlayPanelVisible), false);
        Console.WriteLine("Reconnected");
    }
    public bool tryAgain(nSession anSession)
    {
        return true;
    }
    public void go(nConsumeEvent evt)
    {
        if (evt.getChannelName().Equals(RATES_CHANNEL))
        {
            nEventProperties nep = evt.getProperties();
            myIndexGrid.Dispatcher.BeginInvoke(new
               RatesDataDelegate(updateRatesGrid), nep.getString("name"),
               nep.get("value").ToString());
            return;
        }
    }
    public delegate void RatesDataDelegate(String index, String ival);
    private void updateRatesGrid(String index, String ival)
    {
        try
        {
            Boolean found = false;
            foreach (RatesData item in myRatesDataListSource)
            {
                if (item.Index.Equals(index))
                {
                    if (item.Price != ival)
                    {
                        item.Price = ival;
                        int currentidx = myRatesDataListSource.IndexOf(item);
                        myRatesDataListSource.Remove(item);
                        myRatesDataListSource.Insert(currentidx,
                            new RatesData() {Index = index, Price = ival});
                        myIndexGrid.SelectedIndex = currentidx;
                    }
                    found = true;
                }
            }
            if (!found)
            {
                RatesData newratesd = new RatesData() { Index = index,
                    Price = ival };
                myRatesDataListSource.Insert(0, newratesd);
                myIndexGrid.SelectedIndex = 0;
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error updateing index grid");
            Console.WriteLine(ex.Message);
        }
    }
  }
}
```

Simple Chat Room

This example demonstrates how to subscribe and publish to a Universal Messaging channel.

Application Source Code

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Net;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using com.pcbsys.nirvana.client;
namespace Silverlight_SimpleChatRoom
{
    public partial class Page : UserControl,nEventListener,nReconnectHandler
    {
        private bool started;
        public nSession mySession;
        private ObservableCollection<ChatData> myChatDataListSource =
           new ObservableCollection<ChatData>();
        public Thread sessionThread;
        private nChannel myChatChannel;
        private const string RNAME = "nhps://showcase.um.softwareag.com:443";
        private const string CHAT_CHANNEL = "/showcase/simplechatroom";
        public Page()
        {
            InitializeComponent();
            StartupProgressDialog.IsOpen = true;
            lstChat.ItemsSource = this.myChatDataListSource;
            sessionThread = new Thread(new ThreadStart(startSubscribers));
            sessionThread.IsBackground = true;
            sessionThread.Start();
            App.Current.Host.Content.Resized += (s, e) =>
            {
                theBack.Width = App.Current.Host.Content.ActualWidth;
                theBack.Height = App.Current.Host.Content.ActualHeight;
            };
        }
        public void startSubscribers()
        {
            if (!started)
            {
                try
                {
                    nSessionAttributes nsa = new nSessionAttributes(RNAME, 5);
                    mySession = nSessionFactory.create(nsa, this, "SilverDemoUser");
                    mySession.init();
                    StartupProgressDialog.Dispatcher.BeginInvoke(new
                       setProgressBarMessage(updateStatusMessage),
                       "Subscribing to Chat...");
                    nChannelAttributes ncachat = new nChannelAttributes();
                    ncachat.setName(CHAT_CHANNEL);
```

```
            myChatChannel = mySession.findChannel(ncachat);
            myChatChannel.addSubscriber(this, 0);
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setOverlayPanelVisibleDelegate(setOverlayPanelVisible),
               false);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error starting subscribers: " + e.Message);
            Console.WriteLine(e.StackTrace);
        }
        started = true;
    }
}
public void go(nConsumeEvent evt)
{
    if (evt.getChannelName().Equals(CHAT_CHANNEL))
    {
        nEventProperties nep = evt.getProperties();
        String msg = nep.getString("message");
        String sender = nep.getString("sender");
        nEventAttributes nea = evt.getAttributes();
        long tval = nea.getTimestamp();
        DateTime ttime = ConvertJavaMiliSecondToDateTime(tval);
        lstChat.Dispatcher.BeginInvoke(new
           ChatDataDelegate(updateChatList), sender, msg, ttime.ToString());
        return;
    }
}
private void Send_Button_Click(object sender, RoutedEventArgs e)
{
    if (txtMessage.Text == null) return;
    //to handle enter key pressed in general
    String senderuser = "SilverUser" + ("" +
       mySession.getSessionConnectionId()).Substring(13);
    String message = txtMessage.Text;
    nEventProperties props = new nEventProperties();
    props.put("sender", senderuser);
    props.put("message", message);
    nConsumeEvent evt = new nConsumeEvent(props, "chatmsg");
    myChatChannel.publish(evt);
    txtMessage.Text = "";
}
public DateTime ConvertJavaMiliSecondToDateTime(long javaMS)
{
    DateTime UTCBaseTime = new DateTime(1970, 1, 1, 0, 0, 0,
       DateTimeKind.Utc);
    DateTime dt = UTCBaseTime.Add(new TimeSpan(javaMS *
    TimeSpan.TicksPerMillisecond)).ToLocalTime();
    return dt;
}
public delegate void ChatDataDelegate(String sender, String message,
   String timestamp);
public void updateChatList(String sender, String message, String timestamp)
{
    ChatData somechatmessage = new ChatData() { Message = message,
       Sender = sender, TimeStamp = timestamp };
    myChatDataListSource.Insert(0, somechatmessage);
}
public delegate void setProgressBarMessage(String message);
public void updateStatusMessage(String message)
{
    myStatusMessage.Text = message;
```

```
        }
        public delegate void setOverlayPanelVisibleDelegate(Boolean flag);
        public void setOverlayPanelVisible(Boolean flag)
        {
            StartupProgressDialog.IsOpen = flag;
        }
        public void disconnected(nSession anSession)
        {
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setProgressBarMessage(updateStatusMessage), "Disconnected...");
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setOverlayPanelVisibleDelegate(setOverlayPanelVisible), true);
            Console.WriteLine("Disconnected");
        }
        public void reconnected(nSession anSession)
        {
            StartupProgressDialog.Dispatcher.BeginInvoke(new
               setOverlayPanelVisibleDelegate(setOverlayPanelVisible), false);
            Console.WriteLine("Reconnected");
        }
        public bool tryAgain(nSession anSession)
        {
            return true;
        }
        private void txtMessage_KeyDown(object sender, KeyEventArgs e)
        {
            if (e.Key == Key.Enter && txtMessage.Text != null &&
                        txtMessage.Text.Trim().Length>0)
            {
                //Handle Enter Here.
                e.Handled = true;
                Send_Button_Click(sender, e);
            }
            else
            {
                e.Handled = false;
            }
        }
    }
}
```

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.

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.

Example JNLP (Tradespace)

Our sample Tradespace application is a good example of a Web Start application that uses the Universal Messaging Client API to consume simulated stock index prices, trades as well as news and chat. Below shows the contents of the JNLP file used to launch this application.

4 Commonly Used Features

- Sessions 278
- Channel Attributes 278
- Channel Publish Keys 281
- Queue Attributes 283
- Native Communication Protocols 285
- Comet Communication Protocols 288
- Durable Consumers 290
- Google Protocol Buffers 290
- Named Objects 291
- Event Filtering 291
- Advanced Filtering with Selectors 293
- Using Shared Memory Protocol 296
- Storage Properties 296

This section summarizes commonly used features of Universal Messaging. The features are available using a variety of methods, such as in the Enterprise Manager or in the Server or Client APIs.

Sessions

A session in Universal Messaging represents a logical connection to a Universal Messaging Realm. It consists of a set of session attributes, such as the protocol and authentication mechanism to be used, the host and port the message server is running on and a reconnect handler object.

Most of the session parameters are defined in a string that is called RNAME and resembles a URL. All the sample applications provided use an RNAME Java system property to obtain the necessary session attributes. The following section discusses this in further detail. The RNAME takes the following format.

The RNAME entry can contain an unlimited number of comma-separated values each one representing an interface on a Universal Messaging Realm. Click here for more information on RNAME.

The current version of the Universal Messaging Realm and the Universal Messaging client API supports 4 TCP wire protocols. These are the Universal Messaging Socket Protocol (nsp), the Universal Messaging HTTP Protocol (nhp), the Universal Messaging SSL Protocol (nsps) and the Universal Messaging HTTPS protocol (nhps). These wire protocols are available wherever a connection is required, i.e. client to Realm and Realm to Realm. Click here for more information on communication protocols supported.

Channel Attributes

Universal Messaging channels provide a set of attributes. Depending on the options chosen, these define the behaviour of the events published and stored by the Universal Messaging Realm Server. Each event published onto a channel has a unique id within the channel called Event Id. Using this event id, it is possible for subscribers to re-subscribe to events on a channel from any given point. The availability of the events on a channel is defined by the chosen attributes of the channel upon creation. Channels can be created either using the Universal Messaging Enterprise Manager or programmatically using any of the Universal Messaging Enterprise APIs.

There are a number of important channel attributes which are discussed below.

Channel TTL

The TTL for a channel defines how long (in milliseconds) each event published to the channel will remain available for subscribers to consume. Specifying a TTL of 0 will mean that events will remain on the channel indefinitely. If you specify a TTL of 10000, then after each event has been on the channel for 10000 milliseconds, it will be automatically removed by the server.

Channel Capacity

The capacity of a channel defines the maximum number of events may remain on a channel once published. Specifying a capacity of 0 will mean that there is no limit to the number of events on a channel. If you specify a capacity of 10000, then if there are 10000 events on a channel, and another event is published to the channel, the 1st event will be automatically removed by the server.

Channel Type

Universal Messaging channels can be of the following types:

- persistent
- mixed
- reliable
- simple
- transient
- off-heap
- paged

The difference lies in the type of physical storage used for each type and the performance overhead associated with each type.

Persistent Channels

Persistent channels have their messages stored in the Universal Messaging Realm's persistent channel disk based store. The persistent channel store is a high performance file based store that uses a separate file for each channel on that Realm facilitating migrating whole channels to different Realms. All messages published to a persistent channel will be stored to disk, hence it is guaranteed that they will be kept and delivered to subscribers until it is purged or removed as a result of a TTL or capacity policy. Messages purged from a persistent channel are marked as deleted however the store size will not be reduced until maintenance is performed on the channel using the Universal Messaging Enterprise Manager or an Administration API call. This augments the high performance of the Universal Messaging Realm.

Mixed Channels

Mixed channels allow the users to specify whether the event is stored persistently or in memory as well as the Time To Live (TTL) of the individual event. On construction of a Mixed channel the TTL and Capacity can be set, if the user supplies a TTL for an event this is used instead of the channel TTL.

Reliable Channels

Reliable channels have their messages stored in the Universal Messaging Realm's own memory space. The first fact that is implied is that the maximum number of bytes that all messages across all reliable channels within a Universal Messaging Realm is limited

by the maximum heap size available to the Java Virtual Machine hosting that Realm. The second fact implied is that if the Universal Messaging Realm is restarted for any reason, all messages stored on reliable channels will be removed from the channel as a matter of policy. However, as Universal Messaging guarantees not to ever reuse event ids within a channel, new messages published in those channels will get assigned event ids incremented from the event id of the last message prior to the previous instance stopping.

Simple Channels

Simple channels have their messages stored in the Universal Messaging Realm's own memory space supplying a high-speed channel type. The difference between a Simple and Reliable is the fact that the event ids are reset to 0 in a Simple channel whenever the Universal Messaging Server is restarted.

Transient Channels

A transient channel is like a simple channel in that no event characteristics are stored persistently. In addition to this, data is only ever written to a transient channel when 1 or more consumers are connected to the channel and are able to consume said data. Unlike the simple channel which stores event data in memory transient channels do not store anything, not even in memory. Transient channels can be thought of as a relay between 1 or more publishers and 1 or more subscribers.

Off-heap Channels

Off-heap channels, similar to reliable channels, store the events in memory, but this channel type uses off-heap memory. This allows the normal JVM heap memory to be left free for short lived events, while longer living events can be stored off the JVM heap. This reduces the work the garbage collector needs to do to manage these events since they are out of range of the garbage collector.

Paged Channels

Pages channels allows users access to high speed, off-heap memory mapped files as the basis for the event storage. As for the other channel types, each channel or queue has its own unique file mapping. While similar to the off-heap channel type, where the events are stored in the off heap memory, the paged channel store is also persistent, thereby allowing recovery of events that are written into this store.

Additional Channel Attributes

In addition to the 3 attributes above which define storage behavior for events, there are a number of other important attributes that can be set for a channel.

Dead Event Store

When events are removed automatically, either by the capacity policy of the channel or the age (TTL) policy of the events itself and they have not been consumed, it may be a requirement for those events to be processed separately. If so, channels or queues can be created with a dead event store so any events that are purged automatically from

that have not been consumed will be moved into the dead event store. Dead event stores themselves can be a channel or a queue and can be created with any attributes you wish.

ChannelKeys

Channels can also be created with a set of channel keys which define how channel events can be managed based on the content of the events. For more information, please see the Channel Publish Keys section

Cluster Wide

The cluster wide flag indicates that a channel is created on all cluster realm nodes. For more information on clustering please see our clustering section.

Engine

There are 2 types of optional engine which a channel can use:

- Merge Engine: The Merge Engine is used for Registered Events, and allows delivery of just the portion of an event's data that has changed, rather than of the entire event.
- JMS Engine: The JMS Engine deals with JMS topics within Universal Messaging.

Channel Publish Keys

Channels can be created with a set of Channel Publish Key objects, as well as the default attributes that define behaviour of a channel and the events on a channel.

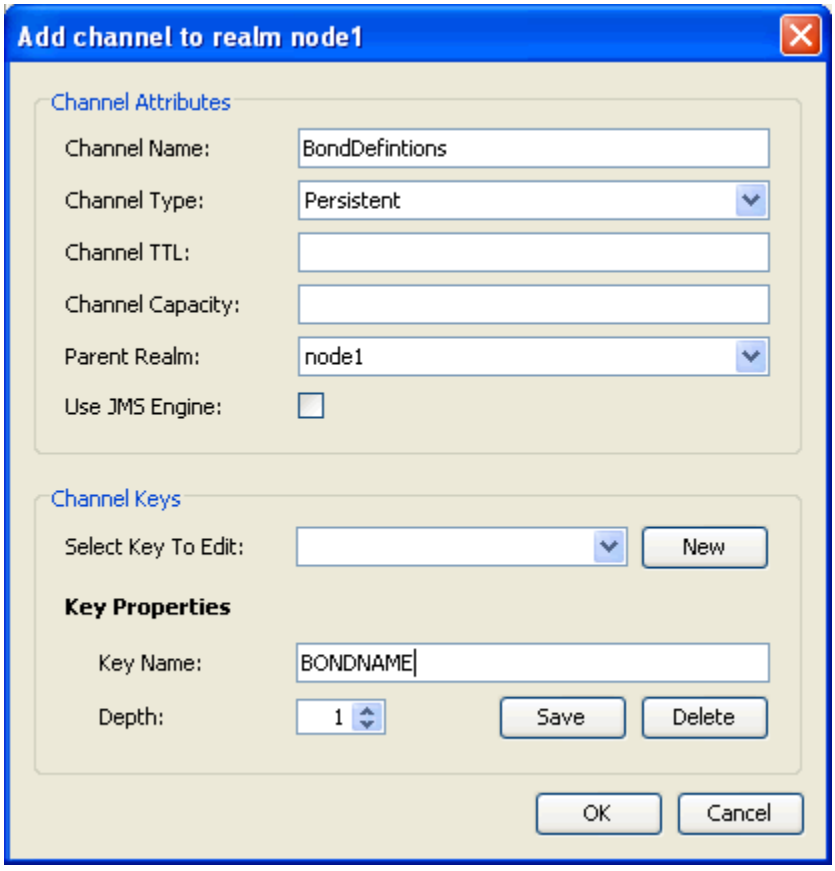
Channel Keys allow a channel or queue to automatically purge old events when new events of the same "type" are received. Two events are of the same "type" if the value in their dictionary (*nEnventAttributes*) for the key defined as the Channel Key are identical. The channel will store the specified number of most recent events whose values match for the Channel Key.

Channel Publish Keys enable the implementation of Last Value Caches. In a last value cache, only the most recent value for a given type of event is kept on the channel. In high-update situations, where only the most recent values are of interest, Channel Publish Keys can greatly improve efficiency in this way. By altering the depth associated with the channel publish key, a recent values cache (where a set number of the most recent events of the same type are stored) can also easily be implemented.

Using Channel Keys to Automatically Purge Redundant Data

For example, if you have a channel called BondDefinitions which should only contain the most recent event published for each Bond, you can enforce this automatically by using a channel key. This functionality vastly simplifies data publication, since the publisher will not have to check the value of data currently on the channel.

In the above example you would create a BondDefinition channel as shown in the following Enterprise Manager screen shot:



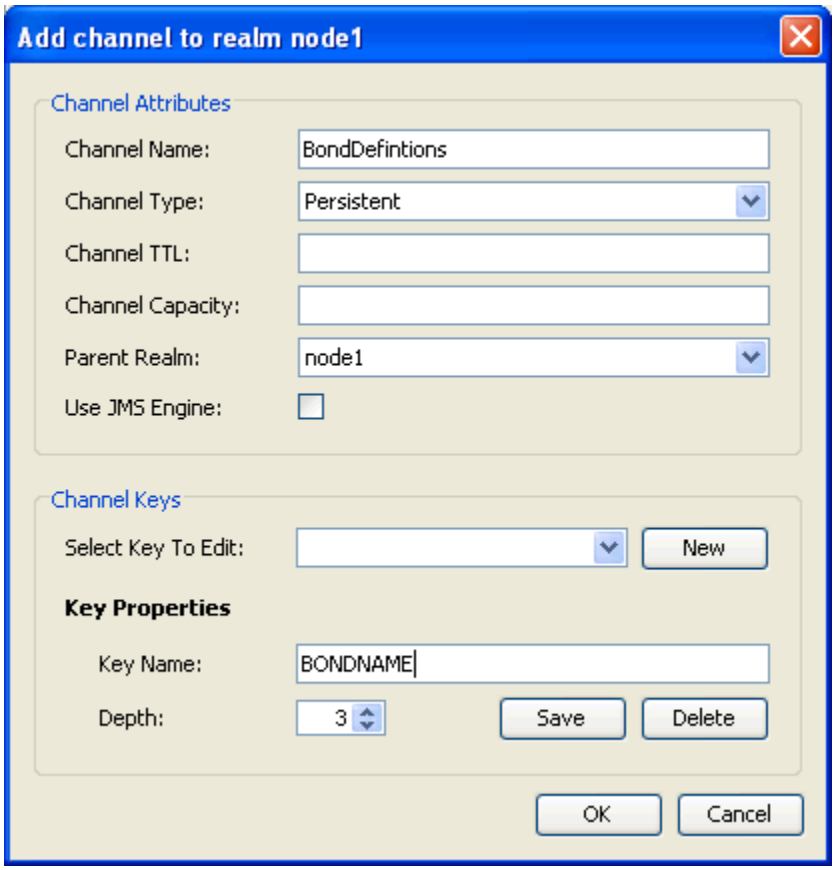
Note that in addition to creating a channel with the Name "BondDefinitions" and a type of "Reliable", the channel also has a Channel Key called BONDNAME with a depth of 1. The channel key defines the key in the nEventProperties which identifies events as being of the same type if their value for this key match. In order to add a ChannelKey, type the name of the key into the Channel Key box on the dialog and click add. If you want the key to have a depth of greater than 1 then click the up arrow adjacent to the Key Depth field or enter the number manually.

If this is configured, as soon as an event is published to the BondDefinitions channel with a Dictionary entry called BONDNAME, the server checks to see if there is another event with the same value for that key. For example, if an event is published with a dictionary containing a key of BONDNAME and value of bondnameA and there is already an event with BONDNAME=bondnameA, then the old event will be removed, and the new one will take its place as the latest definition for bondnameA.

Another example would be if you wanted to keep the latest definition and the 2 before it you would create the channel key with depth 3 as in the following screen shot (implying that maximum 3 events with the same value for key name BONDNAME can exist on the channel).

If you wanted to keep an archive of all bondname values that were published to the channel, you could add a join from the BondDefinitions channel to, for example, a BondDefinitionsArchive channel. On this channel the absence of a Channel Key called

BONDNAME will mean that it will store all events that have been published to the BondDefinitions channel.



Queue Attributes

Universal Messaging channels provide 3 main attributes. Depending on the options chosen, these define the behaviour of the events published and stored by the Universal Messaging Realm Server. The availability of the events on a queue is defined by the chosen attributes of the queue upon creation.

Each of these attributes are described in the following sections.

Queue TTL

The TTL for a queue defines how long (in milliseconds) each event published to the queue will remain available to consumers. Specifying a TTL of 0 will mean that events will remain on the queue indefinitely. If you specify a TTL of 10000, then after each event has been on the queue for 10000 milliseconds, it will be automatically removed by the server.

Queue Capacity

The capacity of a queue defines the maximum number of events may remain on a queue once published. Specifying a capacity of 0 will mean that there is no limit to the number of events on a queue. If you specify a capacity of 10000, then if there are 10000 events on a queue, and another event is published to the queue, the 1st event will be automatically removed by the server.

Queue Type

Universal Messaging queues can be of 4 different types, simple, reliable, persistent and mixed. The difference lies in the type of physical storage used for each type and the performance overhead associated with each type.

Simple Queues

Simple queues have their messages stored in the Universal Messaging Realm's own memory space supplying a high-speed queue type. The difference between a Simple and Reliable is the fact that the event ids are reset to 0 in a Simple queue whenever the Universal Messaging Server is restarted.

Reliable Queues

Reliable queues have their messages stored in the Universal Messaging Realm's own memory space. The first fact that is implied is that the maximum number of bytes that all messages across all reliable queues within a Universal Messaging Realm is limited by the maximum heap size available to the Java Virtual Machine hosting that Realm. The second fact implied is that if the Universal Messaging Realm is restarted for any reason, all messages stored on reliable queues will be removed from the queue as a matter of policy. However, as Universal Messaging guarantees not to ever reuse event ids within a queue, new messages published in those queues will get assigned event ids incremented from the event id of the last message prior to the previous instance stopping.

Persistent Queues

Persistent queues have their messages stored in the Universal Messaging Realm's persistent queue disk based store. The persistent queue store is a high performance file based store that uses a separate file for each queue on that Realm facilitating migrating whole queues to different Realms. All messages published to a persistent queue will be stored to disk, hence it is guaranteed that they will be kept and delivered to subscribers until it is purged or removed as a result of a TTL or capacity policy. Messages purged from a persistent queue are marked as deleted however the store size will not be reduced until maintenance is performed on the queue using the Universal Messaging AdminTool. This augments the high performance of the Universal Messaging Realm.

Mixed Queues

Mixed queues allow the users to specify whether the event is stored persistently or in memory as well as that the Time To Live (TTL) of the individual event. On construction

of a Mixed queue the TTL and Capacity can be set, if the user supplies a TTL for an event this is used instead of the queue

Native Communication Protocols

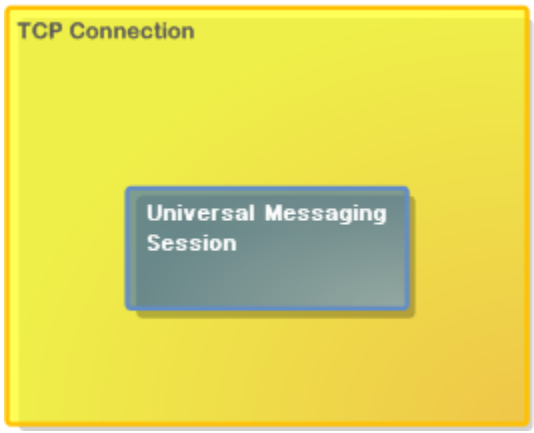
Universal Messaging supports four *Native Communication Protocols*. These TCP protocols are:

- Universal Messaging Socket Protocol (nsp)
- Universal Messaging SSL Protocol (nsps)
- Universal Messaging HTTP Protocol (nhp)
- Universal Messaging HTTPS Protocol (nhps)

These wire protocols are available for client-to-realm and realm-to-realm connections.

Universal Messaging Socket Protocol (nsp)

The Universal Messaging Socket Protocol (NSP) is a plain TCP socket protocol optimized for high throughput, low latency and minimal overhead.



Universal Messaging Socket Protocol (nsp)

Universal Messaging SSL Protocol (nsps)

The Universal Messaging SSL (NSPS) Protocol uses SSL sockets to provide the benefits of the Universal Messaging Socket Protocol combined with encrypted communications and strong authentication.

We strongly recommend use of the NSPS protocol in production environments, where data security is paramount.

Universal Messaging SSL Protocol (nsps)

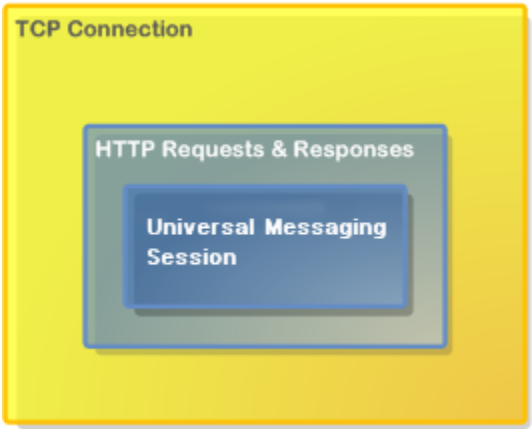
Universal Messaging HTTP Protocol (nhp)

The Universal Messaging HTTP (NHP) Protocol uses a native HTTP stack running over plain TCP sockets, to transparently provide access to Universal Messaging applications running behind single or multiple firewall layers.

This protocol was designed to simplify communication with Realms on private address range (NAT) networks, the Internet, or within another organization's DMZ.

There is no requirement for a web server, proxy, or port redirector on your firewall to take advantage of the flexibility that the Universal Messaging HTTP Protocol offers. The protocol also supports the use of HTTP proxy servers, with or without proxy user authentication.

An nhp interface will also support connections using the nsp protocol. For this reason it is suggested that you use this protocol initially when evaluating Universal Messaging.



Universal Messaging HTTP Protocol (nhp)

Universal Messaging HTTPS Protocol (nhps)

The Universal Messaging HTTPS (NHPS) Protocol offers all the benefits of the Universal Messaging HTTP Protocol described above, combined with SSL-encrypted communications and strong authentication.

We strongly recommend use of the Universal Messaging HTTPS Protocol for production-level applications which communicate over the Internet or mobile networks.

Universal Messaging HTTPS Protocol (nhps)

Recommendation

We generally recommend that you initially use the *Universal Messaging HTTP Protocol (nhp)* for Universal Messaging Native clients, as this is the easiest to use and will support both nhp and nsp connections.

When deploying Internet-applications, we recommend the *Universal Messaging HTTPS Protocol (nhps)* for its firewall traversal and security features.

RNAMEs

The RNAME used by a Native Universal Messaging Client to connect to a Universal Messaging Realm server using a Native Communication Protocol is a non-web-based URL with the following structure:

```
<protocol>://<hostname>:<port>
```

where <protocol> can be one of the 4 available wire protocol identifiers:

- *nsp* (Universal Messaging Socket Protocol),
- *nsps* (Universal Messaging SSL Protocol),
- *nhp* (Universal Messaging HTTP Protocol) or
- *nhps* (Universal Messaging HTTPS Protocol)

An RNAME string consists of a comma-separated list of RNAMEs.

A Universal Messaging realm can have multiple network interfaces, each supporting any combination of Native and Comet communication protocols.

User@Realm Identification

When a Universal Messaging Native Client connects to a Universal Messaging Realm, it supplies the username of the currently-logged-on user on the client host machine.

This username is used in conjunction with the hostname of the realm to create a session credential of the form `user@realm`.

For example if you are logged on to your client machine as user `fred`, and you specify an RNAME string of `nsp://realmserver.mycompany.com:9000`, then your session will be identified as `fred@realmserver.mycompany.com`.

Note, however, that if you were running the client application on the same machine as the Universal Messaging Realm and decided to use the `localhost` interface in your RNAME string, you would be identified as `fred@localhost` - which is a different credential.

The Realm and channel Access Control Lists (ACL) checks will be performed against this credential, so be careful when specifying an RNAME value.

Comet Communication Protocols

Universal Messaging supports Comet and WebSocket over two *Comet Communication Protocols*.

Streaming Comet, Long Polling or WebSocket

The Universal Messaging Comet API supports several both streaming and long polling Comet or WebSocket communications. A developer can select which method to use when starting a session with the JavaScript API.

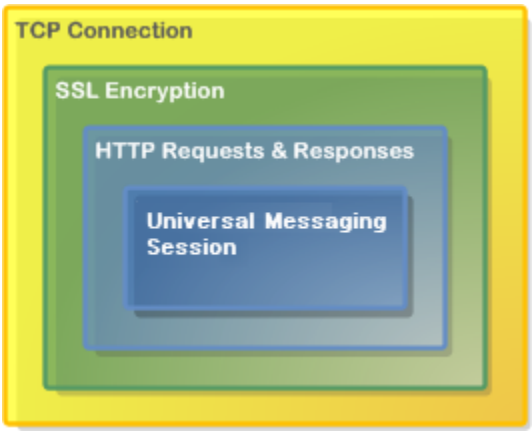
Communication Protocols

HTTPS Protocol (https)

The Universal Messaging Comet HTTPS (SSL-encrypted HTTP) Protocol is a lightweight web-based protocol, optimized for communication over web infrastructure such as client or server-side firewalls and proxy servers.

This protocol simplifies communication between Universal Messaging Clients and Realms running behind single or multiple firewall layers or on private address range (NAT) networks. There is no requirement for an additional web server, proxy, or port redirector on your firewall to take advantage of the flexibility that the Universal Messaging HTTPS Protocol offers.

The protocol is fully SSL-encrypted and also supports the use of HTTP proxy servers, with or without proxy user authentication.

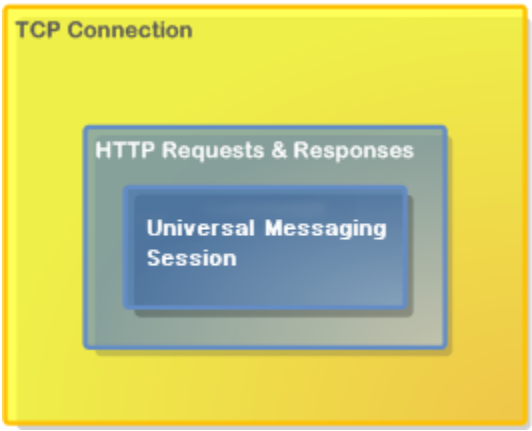


HTTPS Protocol (https)

HTTP Protocol (http)

The Universal Messaging Comet HTTP Protocol is a lightweight web-based protocol, supporting communication through proxies and firewalls at the client or server end of the network.

This protocol provides the same functionality as the Universal Messaging Comet HTTPS protocol, but without SSL encrypted communications.



HTTP Protocol (http)

Recommendation

We generally recommend the *HTTPS Protocol (https)* for Universal Messaging Comet clients, as this is both securely encrypted and easy to use.

RNAMEs

The RNAME used by a Universal Messaging Comet Client to connect to a Universal Messaging Realm server will automatically default to the same protocol/host/port as the web server from which an application is served, unless overridden by the developer when starting a session.

Note that a Universal Messaging realm can have multiple network interfaces, each supporting any combination of Native and Comet communication protocols.

Durable Consumers

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

Universal Messaging supports durable consumers through use of Universal Messaging named objects. When a subscription is created using a named object, the server will ensure that the named object current position is maintained. As named object subscriptions are restarted, say after application restart, the server will begin delivering events from the last event which was successfully acknowledged by the client.

There are 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 on the nConsumeEvent

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.

Universal Messaging supports server-side filtering of Google Protocol Buffers, and this, coupled with Google Protocol Buffer's space-efficient serialization can be used to reduce the amount of data delivered to a client. If server side filtering is not required, the serialised protocol buffers could be loaded into a normal nConsume Event as the event data.

The structure of the data is defined in a .proto file, messages are constructed from a number of different types of fields and these fields can be required, optional or repeated. Protocol Buffers can also include other Protocol Buffers.

The serialization uses highly efficient encoding to make the serialized data as space efficient as possible, and the custom generated code for each data format allows for rapid serialization and deserialization.

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 Flex, .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.

These nProtobufEvents are integrated seamlessly seamlessly in nirvana, allowing for server-side filtering of Google Protocol Buffer events, which can be sent on resources just like a normal nirvana Events. The server side filtering of messages is achieved by providing the server with a description of the data structures(constructed at the .proto compile time, using the standard protobuf compiler and the --descriptor_set_out option). The default location the sever looks in for descriptor files is /plugins/ ProtobufDescriptors and this can be configured through the enterprise manager. The server will monitor this folder for changes, and the frequency of these updates can be configured through the enterprise manager. The server can then use to extract the key value pairs from the binary Protobuf message and filter message delivery based on user requirements.

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.

The Enterprise Manager can be used to view, edit and republish protocol buffer events, even if the EM is no running on the same machine as the server. To enable this, the server outputs a descriptor set to a configurable directory(by default the htdocs directory for the realm) and this can then be made available through a file plugin etc. The directory can be changed through the enterprise manager. The enterprise manager can then be configured to load this file using -DProtobufDescSetURL and then the contents of the protocol buffers can be parsed.

Named Objects

Universal Messaging provides the ability for the server to maintain state for the last event that was consumed by a consumer on a channel. By providing a unique name, you can create a named object on a channel and even when your application is stopped, the next time you start your application, you will only consume available events from the last event id that the server stored as successfully consumed by that named object.

Named objects can be persistent, i.e. the last event id is written to disk, so that if the Universal Messaging Realm Server is restarted, the last event id consumed is retrievable for each named object on a channel.

Event Filtering

Universal Messaging provides a server side filtering engine that allows events to be delivered to the client based on the content of values within the event dictionary.

This section introduces filtering and describes the basic syntax of the filtering engine, and provides examples to assist developers with designing the content of the events within Universal Messaging. The filtering capabilities described in this page are what is defined by the JMS standard.

Universal Messaging filtering can be applied at two levels. The first is between client and server and the second is between server and server.

All Universal Messaging filtering is handled by the Universal Messaging server and therefore significantly reduces client overhead and network bandwidth consumption.

For documentation on filtering functionality which extends beyond that available through the JMS standard please refer to the advanced filtering section (see "Advanced Filtering with Selectors" on page 293).

Basic Filtering

Each Universal Messaging event can contain an event dictionary as well as a byte array of data. Standard filtering, as defined by JMS, allows dictionary entries to be evaluated based on the value of the dictionary keys prior to delivering the data to the consumer.

The basic syntax of the filter strings is defined in the following notation :

```
EXPRESSION
```

where:

```
EXPRESSION ::=
<EXPRESSION> |
<EXPRESSION> <LOGICAL_OPERATOR> <EXPRESSION> |
<ARITHMETIC_EXPRESSION> |
<CONDITIONAL_EXPRESSION>
ARITHMETIC_EXPRESSION ::=
<ARITHMETIC_EXPRESSION> <ARITHMETIC_OPERATOR> <ARITHMETIC_EXPRESSION> |
<ELEMENT> <ARITHMETIC_OPERATOR> <ARITHMETIC_EXPRESSION> |
<ARITHMETIC_EXPRESSION> <ARITHMETIC_OPERATOR> <ELEMENT>
CONDITIONAL_EXPRESSION ::=
<ELEMENT> <COMPARISON_OPERATOR> <ELEMENT> |
<ELEMENT> <LOGICAL_OPERATOR> <COMPARISON_OPERATOR> <ELEMENT>
ELEMENT ::=
<DICTIONARY_KEY> |
<NUMERIC_LITERAL> |
<LOGICAL_LITERAL> |
<STRING_LITERAL> |
<FUNCTION>
LOGICAL_OPERATOR ::= NOT | AND | OR
COMPARISON_OPERATOR ::= <> | > | < | = | LIKE | BETWEEN | IN
ARITHMETIC_OPERATOR ::= + | - | / | *
DICTIONARY_KEY ::= The value of the dictionary entry with the specified key
LOGICAL_LITERAL ::= TRUE | FALSE
STRING_LITERAL ::= <STRING_LITERAL> <SEPARATOR> <STRING_LITERAL> |
                   Any string value, or if using LIKE,
                   use the '_' character to denote wildcard
NUMERIC_LITERAL ::= Any valid numeric value
SEPARATOR ::= ,
FUNCTION ::= <NOW> | <EVENTDATA> | DISTANCE
```

The above notation thus gives rise to the creation of any of the following valid example selector expressions :

```
size BETWEEN 10.0 AND 12.0
country IN ('uk', 'us', 'de', 'fr', 'es' ) AND size BETWEEN 14 AND 16
country LIKE 'u_' OR country LIKE '_e_'
size + 2 = 10 AND country NOT IN ('us', 'de', 'fr', 'es')
size / 2 = 10 OR size * 2 = 20
size - 2 = 8
```

```
 size * 2 = 20
price - discount < 10.0 AND ((discount / price) * price) < 0.4
```

Additional references for event filtering may be found within the JMS message selector section of the JMS standard.

Advanced Filtering with Selectors

Universal Messaging supports *standard selector based filtering* and some advanced filtering concepts which will be described here .

Content Sensitive Filtering

Each Universal Messaging event can contain an event dictionary and a tag, as well as a byte array of data. Standard filtering, as defined by JMS, allows dictionary entries to be evaluated based on the value of the dictionary keys prior to delivering the data to the consumer.

Universal Messaging also supports a more advanced form of filtering based on the content of the event data (byte array) itself as well as time and location sensitive filtering. Universal Messaging also supports filtering based on arrays and dictionaries contained within event dictionaries. There is no limit to the dept of nested properties that can be filtered.

Filtering based on nested arrays and dictionaries

An event dictionary can contain primitive types as well as dictionaries. They can also contain arrays of primitive types and arrays of dictionaries. Universal Messaging supports the ability to filter based on these nested arrays and dictionaries.

if an nEventProperties object contains a key called NAMES which stores a String[] then it is possible to specify a filter that will only deliver events that match based on values within the array.

```
NAMES[] = 'myname'
```

- Returns events where any element in the NAMES array = 'myname'

```
NAMES[1] = 'myname'
```

- Returns events where the second element in the array = 'myname'

Similarly, if the array was an nEventProperties[] it would be possible to filter based on the values within the individual nEventProperties objects contained within the array.

For example if the event's nEventProperties contains a key called CONTACTS which stores an nEventProperties[] then the following selectors will be available.

```
CONTACTS[].name = 'aname'
```

- Return events where the CONTACTS array contains an nEventProperties which contains a key called name with the value 'aname'

```
CONTACTS[1].name = 'aname'
```

- Return events where the second element in the CONTACTS array of nEventProperties contains a key called name with the value 'aname'

```
CONTACTS[].NAMES[] = 'myname'
```

- Return events where the CONTACTS array contains a NAMES arrays with a value 'myname' somewhere in the NAMES array.

EventData Byte[] Filtering

Universal Messaging's filtering syntax supports a keyword called 'EVENTDATA' that corresponds to the actual byte array of data within the Universal Messaging event. There are a number of operations that can be performed on the event data using this keyword.

This enables a reduction in the amount of data you wish to send to clients, since rather than querying pre-determined dictionary values, you can now query the actual data portion of the event itself without having to provide dictionary entries. If you have a message structure and part of this structure includes the length of each value within the structure, then you can refer to each portion of data. Alternatively if you know the location of data within you byte array these can be used for filtering quite easily.

Below is a list of the available operations that can be performed on the EVENTDATA.

```
EVENTDATA.LENGTH()
```

- Returns the length of the byte[] of the data in the event.

```
EVENTDATA.AS-BYTE(offset)
```

- Returns the byte value found within the data at the specified offset.

```
EVENTDATA.AS-SHORT(offset)
```

- Returns a short value found within the data at the specified offset. Length of the data is 2 bytes.

```
EVENTDATA.AS-INT(offset)
```

- Returns a int value found within the data at the specified offset. Length of the data is 4 bytes.

```
EVENTDATA.AS-LONG(offset)
```

- Returns a long value found within the data at the specified offset. Length of the data is 8 bytes.

```
EVENTDATA.AS-FLOAT(offset)
```

- Returns a float value found within the data at the specified offset. Length of the data is 4 bytes.

```
EVENTDATA.AS-DOUBLE(offset)
```

- Returns a double value found within the data at the specified offset. Length of the data is 8 bytes.

```
EVENTDATA.AS-STRING(offset, len)
```

- Returns a String value found within the data at the specified offset for the length specified.

```
EVENTDATA.TAG()
```

- Returns the String TAG of the event if it has one.

For example, we could then provide a filter string in the form of :

```
EVENTDATA.AS-STRING(0, 2) = 'UK'
```

If we knew that at position 0, the first 2 bytes would be a string that represents a value you wish to filter on.

If we had data with 5 string values of varying length, and each length was prepended to each string in 2 bytes, then we could evaluate any portion of the string as follows:

```
EVENTDATA.AS-STRING(0, EVENTDATA.AS-INT(0)) LIKE 'LON'
```

and the second string value after that would be calculated as follows:

```
 EVENTDATA.AS-STRING( EVENTDATA.AS-SHORT(0)+4,
                      EVENTDATA-AS-SHORT(EVENTDATA.AS-SHORT(0)+2) )
```

The offset is calculated based on the length of the first string + the 2 bytes of the first strings size and 2 bytes for the size of the second string (Hence +4). This offset gives you the size of the second string. Then you just need to get size of the second string, which is found by EVENTDATA-AS-SHORT(EVENTDATA.AS-SHORT(0)+2).

This provides a powerful way of embedding functions within functions in order to evaluate the data within an event.

Time Sensitive Filtering

Universal Messaging's filtering syntax also supports a function called 'NOW()' that is evaluated at the server as the current time in milliseconds using the standard Java time epoch. This function enables you to filter events from the server using a time sensitive approach. For example, if your data contained a dictionary key element called 'DATE_SOLD' that contained a millisecond value representing the data when an item was sold, one could provide a filter string on a subscription in the form of:

```
DATA_SOLD < (NOW() - 86400000)
```

Which would deliver events corresponding to items sold in the last 24 hours. This is a powerful addition to the filtering engine within Universal Messaging.

Location Sensitive Filtering

Universal Messaging's filtering engine supports a keyword called DISTANCE. This keyword is used to provide geographically sensitive filtering. This allows the calculation of the distance between two points on the earths surface as defined by the latitude and longitude.

For example, if you were designing a system that tracked the location of a tornado, as the tornado moved position, the latitude and longitude would correspond to the geographical location on the earth's surface. As the position of a tornado changed, an event would be published containing the new latitude and longitude values as keys within the dictionary ('latitude' and 'longitude' respectively). Using this premise, you could provide a filter in the form of:

```
DISTANCE(Lat, Long, Units)
```

where :

- Latitude : the floating point value representing the latitude
- Longitude : the floating point value representing the longitude
- Units : Optional string indicating the return value to be
  - K: Kilometers < Default >
  - M : Miles
  - N : Nautical Miles

For example :

```
DISTANCE ( 51.50, 0.16, M ) < 100
```

Which would deliver events that corresponded to tornadoes that were less than 100 miles away for the latitude and longitude values provided in the filter string.

The DISTANCE keyword provides a valuable and powerful extension to Universal Messaging's filtering capabilities. If you require information that is sensitive to geographical locations

Using Shared Memory Protocol

Universal Messaging supports a special kind of communication protocol called shm *(Shared Memory)*. This communication protocol can only be used for intra host connectivity and uses physical memory to pass data rather than the network stack. Using shared memory as the communication protocol behaves just as any other nirvana communication protocol and therefore can be used within any Universal Messaging client or admin api application.

Once you have configured shared memory on your realm, it is ready to use by any Universal Messaging application you wish to run on the same host. All you need to do is set your RNAME to a the correct shared memory RNAME. For example, if you have configured shared memory to use /tmp, then your RNAME would be:

shm://localhost/tmp

To test this out, you could run any one of the example applications that are provided in the Universal Messaging download, by setting the RNAME from a Universal Messaging Java client examples prompt as described above. For example, a subscriber that subscribes to a channel called /test can be executed as follows:

nsubchan /test 0 1

Storage Properties

This storage properties panel allows for configuration of advanced storage properties, a summary of these properties can be seen below:

- *Auto Maintenance:* Controls whether persistent store is maintained automatically (i.e. events reaching their TTL, or events which have been purged are cleared from the channel storage file.
- *Honour Capacity:* Whether the channel / queue capacity setting will prevent publishing of any more data once full. If true, the client will get an exception on further publishes (a transactional publish will receive an exception on the commit call, a non transactional publish will receive an asynchronous exception through the nAsyncExceptionHandler). If false the oldest event will be purged to make room for the newest.
- *Enable Caching:* Control the caching algorithm within the server, if you set caching to false, all events will be read from the file store. If true then if server has room in memory, they will be stored in memory and reused.
- *Cache on Reload:* When a server restarts it will scan all file based stores and check for corruption. During this test the default behaviour is to disable caching to conserve memory, however, in some instances it would be better if the server had actually cached the events in memory for fast replay.
- *Enable Read Buffering:* Control the read buffering logic for the store on the server.
- *Read Buffer Size:* If ReadBuffering is enabled then this function sets the size in bytes of the buffer to use.
- *Sync Each Write:* Control whether each write to the store will also call sync on the file system to ensure all data is written to the Disk
- *Sync Batch Size:* Control how often in terms of number of events to sync on the file system to ensure all data is written to the Disk
- *Sync Batch Time:* Control how often in terms of time elapsed to sync on the file system to ensure all data is written to the Disk
- *Fanout Archive Target:* Control whether all events fanned out are written to an archive q