Universal Messaging Concepts

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-CO-97-20141015

Table of Contents

Overview

This guide describes the underlying concepts of the Universal Messaging product, with the focus on the following areas of design and functionality:

- Architecture
- Management
- Performance, Scalability and Resilience
- Deployment
- Security
- MQTT (MQ Telemetry Transport)

1 Architecture

Architecture Overview

Universal Messaging is a Message Orientated Middleware product that guarantees message delivery across public, private and wireless infrastructures. Universal Messaging has been built from the ground up to overcome the challenges of delivering data across different networks. It provides its guaranteed messaging functionality without the use of a web server or modifications to firewall policy.

Universal Messaging design supports both *broker-based* and *brokerless* communication, and thus comprises client and server components.

Broker-Based Communication

The standard UM "broker-based" client component can be subdivided into messaging clients, comet clients and management clients. The server component has specific design features to support each of these classifications of client as well as Scheduling and Triggers, Plugins, Federation, Clustering and Low Latency IO.

Server Components

The Universal Messaging realm server is a heavily optimized Java process capable of delivering high throughput of data to large numbers of clients while ensuring latencies are kept to a minimum. In addition to supporting the client types described below the Universal Messaging realm server has a number of built in features to ensure its flexibility and performance remains at the highest levels.

Client Components

Universal Messaging supports 3 client types:

- Messaging clients
- Comet clients
- Management clients

Each client type is been developed using open protocols with specific attention paid to performance and external deployment. Each client type has been specifically designed to transparently pass through firewalls and other security infrastructure while providing its own inherent security features.

Messaging Clients

Universal Messaging messaging clients support synchronous and asynchronous middleware models. Publish Subscribe, Queues and Peer to Peer functionality is all supported and can be used independently or in combination with each other. Universal Messaging messaging clients can be developed in a wide range of languages on a wide range of platforms. Java, C# and C++ over Win32, Solaris and Linux are all supported. Mobile devices and Web technologies such as Silverlight all exist as native messaging clients.

WebSocket, Comet and LongPolling for JavaScript Clients

In addition to our native binary wire protocol Universal Messaging also supports text based delivery for languages that do not support binary data. Used in conjunction with Universal Messaging server plug-in technology, Comet and Long Polling clients use HTTP and persistent connections to deliver asynchronous Publish Subscribe, and Peer to Peer functionality to clients. JavaScript clients can also use WebSocket as a delivery approach although this is not yet sufficiently supported in users' browsers to warrant a reliance on it over Comet/long Polling.

Management Clients

Universal Messaging provides a very extensive and sophisticated management API written in Java. Management clients can construct resources (Channels, ACL's queues etc.) and query management data (throughput, cluster state, numbers of connections etc.) directly from Universal Messaging realm servers.

Brokerless Communication

Universal Messaging offers, in addition to its standard full-featured client-server API, an extremely lightweight client-client communication API known as the Brokerless API (currently available for Java only).

Support for Open Standards

Universal Messaging supports many open standards at different levels from network protocol support through to data definition standards.

At the network level Universal Messaging will run on an TCP/IP enabled network supporting normal TCP/IP based sockets, SSL enabled TCP/IP sockets, HTTP and HTTPS, providing multiple communications protocols (see "Communication Protocols and RNAMEs" on page 22).

Universal Messaging provides support for the JMS standard.

Realms

A Universal Messaging Realm is the name given to a single Universal Messaging server. Universal Messaging realms can support multiple network interfaces, each one supporting different Universal Messaging protocols.

A Universal Messaging Realm can contain many Channels, Message Queues or Peer to Peer services.

Universal Messaging provides the ability to create clusters of realms that share resources (see "Messaging Paradigms supported" on page 12) within the namespace. Cluster objects can be created, deleted and accessed programmatically or through the Universal Messaging Administration Tool.

Objects created within a cluster can be accessed from any of the realms within the cluster and Universal Messaging ensures that the state of each object is maintained by all realms within a cluster. The clustering technology used within Universal Messaging ensures an unsurpassed level of reliability, resilience, scalability.

Realms can also be added to one another within the namespace. This allows the creation of a federated namespace (see "Federation Of Servers" on page 73) where realms may be in different physical location, but accessible through one physical namespace.

Messaging Paradigms supported

Universal Messaging supports three broad messaging paradigms - Publish/Subscribe, Message Queues and Peer to Peer. Universal Messaging clients can use a mixture of these paradigms from a single session. In addition to this it is possible for clients to further control the conversation that takes place with the server by choosing particular styles of interaction. These styles are available to both readers and writers of messages and include asynchronous, synchronous, transactional and non-transactional.

Publish/Subscribe (using Channels/Topics)

Publish/Subscribe is an asynchronous messaging model where the sender (publisher) of a message and the consumer (subscriber) of a message are decoupled. When using the Channels/Topics, readers and writers of events are both connected to a common topic or channel. The publisher publishes data to the channel. The channel exists with the Universal Messaging realm server. As messages arrive on a channel, the server automatically sends them on to all consumers subscribed to the channel. Universal Messaging supports multiple publishers and subscribers on a single channel.

Publish/Subscribe (using DataStreams and DataGroups)

Universal Messaging DataGroups provide an alternative to Channels/Topics for Publish/Subscribe. Using DataGroups it is possible for remote processes to manage subscriptions on behalf of subscribers. The process of managing subscribers can be carried out by publishers themselves or by some other process. DataGroups are lightweight in nature and are designed to support large numbers of users whose subscriptions are fluid in nature. The addition / removal of users from DataGroups can be entirely transparent from the user perspective.

Message Queues

Like pub/sub, message queues decouple the publisher or sender of data from the consumer of data. The Universal Messaging realm server manages the fan out of messages to consumers. Unlike pub/sub with channels, however, only one consumer can read a message from a queue. If more than one consumer is subscribed to a queue then the messages are distributed in a round-robin fashion.

Peer to Peer

Peer to Peer provides a direct communications path between an instance of a service and the client requiring access to the service. The Universal Messaging realm server brokers the relationship between the service and the client and in doing so becomes transparent as messages pass through it.

Details of Usage for Software Developers

Details of how to develop applications in various languages for the paradigms described above, including code examples, are provided in the appropriate language-specific guides.

Messaging Paradigms Comparison

	DataGroups	Channels	Queues	Peer to Peer
Subscriptions Managed Remotely	✓			
Nestable	✓			
Clusterable	✓	✓	✓	
Persistence (data)		✓	✓	
Persistence (state)	✓	✓	✓	
Message Replay	✓	✓	✓	✓
Synchronous Consumer		✓	✓	
Asynchronous Consumer	✓	✓	✓	✓
Synchronous Producer	✓	✓	✓	

	DataGroups	Channels	Queues	Peer to Peer
Asynchronous Producer	✓	✓	✓	✓
Transactional Consumer		✓	✓	
Non-Transactional Consumer	✓	✓	✓	✓
Transactional Producer		✓	✓	
Non-Transactional Producer	✓	✓	✓	✓
Destructive Read			✓	✓
Delta Delivery	✓	✓		
Conflation (Event Merge)	✓	✓		
Conflation (Event Overwrite)	✓	✓		
Conflation (Throttled Delivery)	✓			
User Specified Filters		✓	✓	
Addressable Messages		✓	✓	✓

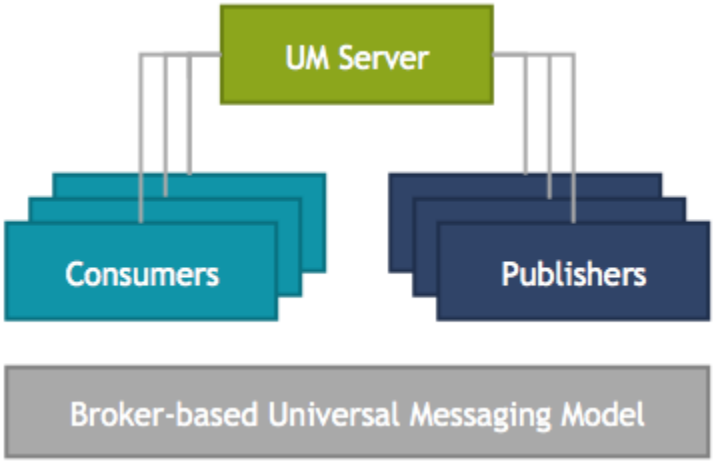
	DataGroups	Channels	Queues	Peer to Peer
User Access Controlled via ACLs	○	✓	✓	✓
Microsoft Reactive Extensions	✓	✓	✓	○
Accessible via JMS	○	✓	✓	○

umTransport API

Universal Messaging offers, in addition to its standard full-featured client-server API, an extremely lightweight client-client communication API known as the umTransport API.

Broker-based Model

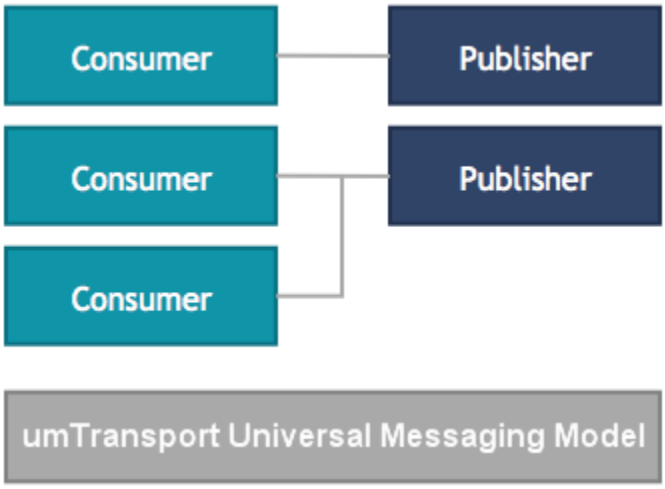
Historically, messaging architecture has predominantly been based on a 'broker in the middle' approach. This is often referred to as 'hub and spoke'. The broker acts as the communications hub, routing messages between logically decoupled peers:



The pub-sub model is a common paradigm for broker based architecture, where one or more publishers send messages to the broker, which then distributes the messages to interested consumers.

umTransport Model

The umTransport model is a peer to peer model that allows peers to be aware of how to communicate directly with one another rather than through a broker. In effect, each publisher peer acts like a server, and each consumer can communicate directly with the publishers:



While this model bypasses broker messaging functionality such as persistence or transactional semantics, it results in a considerably lower latency delivery of information from a publisher to a consumer. By halving of the number of "hops" between client and publisher, latency too is effectively halved. This is especially useful when ultra-low latency message delivery is paramount (in, for example, the links between pricing, quant and risk engines in FX trading platforms).

The umTransport API is currently available for Java and C++. For Java, it is located in the com.softwareag.um.io package. For C++, it is located in com::softwareag::umtransport.

The Java API

The Java API is very simple, allowing each client to accept connections from other clients, and to receive arbitrary binary data from these clients synchronously or asynchronously. In many ways the API is similar to a standard TCP socket API, but offers the additional benefit of being able to use not just TCP sockets as a communication transport, but any of the following Universal Messaging communication technologies:

- TCP Sockets: data is transmitted directly over TCP Sockets
- SSL: data is SSL encrypted then transmitted over TCP Sockets
- SHM: data is transmitted via Shared Memory (for near-instant access by processes on the same machine)
- RDMA: data is transmitted via Remote Direct Memory Access (for access by processes on a remote machine; requires network adapters that support RDMA)

The C++ API

The C++ API provides a subset of the functionality available in the Java API, with the following restrictions:

- The C++ API does not support asynchronous communication between clients.
- The C++ API supports only TCP sockets as a communication transport.

Using the Java API

Let's take a quick look at how to use this API. Here is an example "echo" Java client and server; the EchoClient will write a string to the EchoServer; the EchoServer will respond to the EchoClient.

Here's the Java EchoClient:

```
package com.softwareag.um.io.samples.echo;
import com.softwareag.um.io.ClientContextBuilderFactory;
import com.softwareag.um.io.ClientTransportContext;
import com.softwareag.um.io.SynchronousTransport;
import com.softwareag.um.io.TransportFactory;
import com.softwareag.um.io.samples.SimpleMessage;
import com.softwareag.um.io.samples.SynchronousClient;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
/**
 * This sample app simply writes a string entered into the console to an EchoServer
 * The EchoServer will respond and this response will be output on the console.
 */
public class EchoClient {
  public EchoClient(String url) throws IOException {
    //Use the factory to generate the required builder based on the protocol
    // in the url string
    ClientTransportContext context =
       ClientContextBuilderFactory.getBuilder(url).build();
    //We do not pass any handlers to the connect method because we want a
    // synchronous transport
    SynchronousTransport transport = TransportFactory.connect(context);
    //This is just a basic wrapper for the client transport so it is easier to
    // read/write messages
    SynchronousClient<SimpleMessage> client = new SynchronousClient<>(transport);
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    //Start a new thread to read from the client transport because read is a
    // blocking call
    new ReadThread(client);
    //Now continue to write messages to the EchoServer until the user enter 'quit'
    while(true){
      System.out.println("Enter a message or type 'quit' to exit >");
      String line = br.readLine();
      if(line.equalsIgnoreCase("quit")){
        break;
      }
      else{
        client.write(new SimpleMessage(line));
      }
    }
  }
  private static class ReadThread extends Thread{
    SynchronousClient<SimpleMessage> client;
    public ReadThread(SynchronousClient<SimpleMessage> _client){
      client = _client;
```

```
      start();
    }
    @Override
    public void run(){
      try{
        while(true){
          SimpleMessage mess = client.read(new SimpleMessage());
          System.out.println(mess.toString());
        }
      }
      catch(Exception e){
        e.printStackTrace();
      }
    }
  }
  public static void main(String[] args) throws IOException {
    if(args.length == 0){
      usage();
      System.exit(1);
    }
    new EchoClient(args[0]);
  }
  public static void usage(){
    System.out.println("EchoClient <URL>");
    System.out.println("<Required parameters>");
    System.out.println(
      "\tURL - protocol://host:port for the server to connect to e.g. "
      +TransportFactory.SOCKET+"://localhost:9000");
  }
}
```

And, the EchoServer:

```
package com.softwareag.um.io.samples.echo;
import com.softwareag.um.io.ServerContextBuilderFactory;
import com.softwareag.um.io.ServerTransportContext;
import com.softwareag.um.io.SynchronousServerTransport;
import com.softwareag.um.io.SynchronousTransport;
import com.softwareag.um.io.TransportFactory;
import com.softwareag.um.io.samples.SimpleMessage;
import com.softwareag.um.io.samples.SynchronousClient;
import java.io.IOException;
/**
 * This sample will only handle one client connection at a time. When a client
 * connects, the EchoServer will immediately respond to any messages with exactly
 * the same message.
 */
public class EchoServer implements Runnable{
  private volatile SynchronousClient<SimpleMessage> client;
  private final SynchronousServerTransport transport;
  private volatile boolean stopped = false;
  public EchoServer(String url) throws IOException {
    //The factory will create the correct context based on the protocol in the url
    ServerTransportContext context =
       ServerContextBuilderFactory.getBuilder(url).build();
    //Because we have not passed an AcceptHandler into the bind method, we are returned
    //a SynchronousServerTransport. This means we have to call accept on the transport
    //to accept new client transports.
    transport = TransportFactory.bind(context);
  }
  public static void main(String[] args) throws IOException {
    if(args.length == 0){
      usage();
      System.exit(1);
```

```
    }
    EchoServer echoServer = new EchoServer(args[0]);
    Thread t = new Thread(echoServer);
    t.start();
    System.out.println("Press enter to quit.");
    System.in.read();
    echoServer.close();
  }
  public static void usage(){
    System.out.println("EchoServer <URL>");
    System.out.println("<Required parameters>");
    System.out.println(
       "\tURL - protocol://host:port to bind the server transport to e.g. "
       +TransportFactory.SOCKET+"://localhost:9000");
  }
  protected void close(){
    stopped = true;
    client.close();
    transport.close();
  }
  @Override
  public void run() {
    try{
      while(true){
        System.out.println("Waiting for client");
        //accept() will block until a client makes a connection to our server
        SynchronousTransport clientTransport = transport.accept();
        System.out.println("Client connected.  Echo service started.");
        //The SyncronousClient is simply a wrapper to make reading/writing easier
        client = new SynchronousClient<>(clientTransport);
        try{
          while(!stopped){
            client.write(client.read(new SimpleMessage()));
          }
        }
        catch (IOException e){
          System.out.println("Connection closed");
        }
      }
    }
    catch(IOException e){
      e.printStackTrace();
    }
  }
}
```

Using the C++ API

Here's the C++ EchoClient:

```
#include "EchoClient.h"
#include "ClientTransportContextFactory.h"
#include "TransportFactory.h"
#include <utility>
#include <iostream>
com::softwareag::umtransport::samples::echo::EchoClient::EchoClient(std::string url)
{
 m_url = url;
}
com::softwareag::umtransport::samples::echo::EchoClient::~EchoClient()
{
}
void com::softwareag::umtransport::samples::echo::EchoClient::run(){
```

```
 try{
  //Use the factory to generate the required builder based on the protocol in
  // the url string
  auto context = ClientTransportContextFactory::build(m_url);
  //We do not pass any handlers to the connect method because we want a
  // synchronous transport
  auto transport = TransportFactory::connect(std::move(context));
  //This is just a basic wrapper for the client transport so it is easier to
  // read/write messages
  SynchronousClient<SimpleMessage> client(std::move(transport));
  //Start a new thread to read from the client transport because read is a
  // blocking call
  ReadThread readThread(client);
  Poco::Thread th;
  th.start(readThread);
  bool canRun = true;
  //Now continue to write messages to the EchoServer until the user enter 'quit'
  while (canRun){
   std::cout << "Enter a message or type 'quit' to exit >" << std::endl;
   std::string input;
   std::getline(std::cin, input);
   if (input == "quit"){
    canRun = false;
   }
   else{
    client.write(SimpleMessage(input));
   }
  }
  readThread.shutdown();
  client.close();
  th.tryJoin(10000);
 }
 catch (Poco::Exception &ex){
  std::cout << ex.displayText();
 }
}
int com::softwareag::umtransport::samples::echo::EchoClient::main(int argc,
  char** argv){
 if (argc < 2){
  std::cout <<
  "EchoClient <URL>\n<Required parameters>\n\tURL -
   protocol://host:port for the server to connect to e.g. tcp://localhost:9000" <<
  std::endl;
 }
 EchoClient client(argv[1]);
 client.run();
 return 0;
}
com::softwareag::umtransport::samples::echo::EchoClient::ReadThread::ReadThread(
  SynchronousClient<SimpleMessage>
      &client) : m_client(client){
}
com::softwareag::umtransport::samples::echo::EchoClient::ReadThread::~ReadThread(){
}
void com::softwareag::umtransport::samples::echo::EchoClient::ReadThread::shutdown(){
 canRun = false;
}
void com::softwareag::umtransport::samples::echo::EchoClient::ReadThread::run(){
 try {
  while (canRun){
   SimpleMessage message;
   m_client.read(message);
   std::cout << "Message Content: " << message << std::endl;
```

```
  }
 }
 catch (Poco::Exception& e) {
  std::cout << "Connection Closed " << e.displayText();
 }
}
```

And here's the EchoServer:

```
#include "EchoServer.h"
#include "ServerTransportContextFactory.h"
#include "TransportFactory.h"
#include "SynchronousClient.h"
#include "SimpleMessage.h"
using namespace com::softwareag::umtransport;
com::softwareag::umtransport::samples::echo::EchoServer::EchoServer(std::string url){
 //The factory will create the correct context based on the protocol in the url
 //Because we have not passed an AcceptHandler into the bind method, we are returned
 //a SynchronousServerTransport. This means we have to call accept on the transport
 //to accept new client transports.
 m_transport = TransportFactory::bind(ServerTransportContextFactory::build(url));
}
com::softwareag::umtransport::samples::echo::EchoServer::~EchoServer(){
}
int com::softwareag::umtransport::samples::echo::EchoServer::main(int argc,
  char** argv){
 if (argc < 2){
  std::cout << "EchoServer <URL>" << std::endl << "EchoServer <URL>" << std::endl <<
    "\tURL - protocol://host:port to bind the server transport to
     e.g. tcp://localhost:9000" <<
    std::endl;
  exit(1);
 }
 EchoServer echoServer(argv[1]);
 Poco::Thread th;
 th.start(echoServer);
 std::cout << "Press any key to finish" << std::endl;
 std::cin.ignore();
 echoServer.close();
 th.tryJoin(10000);
 return 0;
}
void com::softwareag::umtransport::samples::echo::EchoServer::close(){
 stopped = true;
 m_transport->close();
}
void com::softwareag::umtransport::samples::echo::EchoServer::run(){
 try{
  while (!stopped){
   std::cout << "Waiting for a client" << std::endl;
   //accept() will block until a client makes a connection to our server
   SynchronousClient<SimpleMessage> client(m_transport->accept());
   //Client connected echo servcie started
   try{
    while (!stopped){
     SimpleMessage msg;
     client.read(msg);
     client.write(msg);
    }
   }
   catch (Poco::Exception & ex){
    std::cout << "Connection Closed" << std::endl;
   }
  }
```

```
 }
 catch (Poco::Exception &ex){
  std::cout << ex.displayText() << std::endl;
 }
}
```

Communication Protocols and RNAMEs

Universal Messaging supports several Native Communication Protocols (see "Native Communication Protocols" on page 155) and Comet Communication Protocols (see "Comet Communication Protocols" on page 158).

The following table shows the Communication Protocols supported by each Universal Messaging Client API:

	Native Communication Protocols					Comet Communication Protocols	
	Socket (nsp)	**SSL (nsps)**	**HTTP (nhp)**	**HTTPS (nhps)**	**Shared Memory (shm)**	**HTTPS (https)**	**HTTP (http)**
Java	✓	✓	✓	✓	✓		
C# .NET	✓	✓	✓	✓			
C++	✓	✓	✓	✓	✓		
Python	✓	✓	✓	✓			
Excel VBA	✓	✓	✓	✓			
JavaScript			✓ via WebSocket	✓ via WebSocket		✓	✓
Adobe Flex	✓	✓	✓	✓			
Microsoft Silverlight	✓		✓	✓			

	Native Communication Protocols					Comet Communication Protocols	
	Socket (nsp)	SSL (nsps)	HTTP (nhp)	HTTPS (nhps)	Shared Memory (shm)	HTTPS (https)	HTTP (http)
iPhone	✓	✓	✓	✓			
Android	✓	✓	✓	✓			

RNAMEs

An RNAME is used by Universal Messaging Clients to specify how a connection should be made to a Universal Messaging Realm Server.

A Native Communication Protocol (see "Native Communication Protocols" on page 155) RNAME string looks like:

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

or

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

where:

- `<protocol>` can be one of the 4 available native communications protocol identifiers nsp (socket), nhp (HTTP), nsps (SSL) and nhps(HTTPS).
- `<hostname>` is the hostname or IP address that the Universal Messaging Realm is running
- `<port>` is the TCP port on that hostname that the Universal Messaging Realm is bound to using the same wire protocol.

The RNAME entry can contain a comma or semicolon separated list of values each one representing the communications protocol, host and port currently running on a Universal Messaging Realm.

Using a comma-separated list indicates in order traversal of RNAME values while using a semicolon-separated list indicates random traversal of RNAME values.

If a list of RNAMEs is used and the Universal Messaging session becomes disconnected and cannot reconnect, the API will cascade through the RNAME list until it manages to reconnect. This functionality is particularly useful within the contexts of both clustering (see "Clusters: An Overview" on page 37) and failover (see "Failover" on page 67).

Native Communications Protocol Client Extensions

In addition to the supported protocols shown above, Universal Messaging clients implemented with APIs that support Native Communication Protocols have a number of extensions available to them:

- *nhpsc*

  This mode of https extracts any configured proxy from within the JVM settings and issues a PROXY CONNECT command via said proxy to establish a connection with the required Universal Messaging realm. The established connection then becomes an SSL encrypted socket connection mode and no longer uses http/ https connections for each server request. If the proxy uses authentication then authentication parameters are also extracted from the JVM settings.

- *nhpm and nhpsm*

  Up to and including Universal Messaging 6.x, nhpm and nhpsm were multiplexed versions of the standard http and https protocols. The key difference is that any sessions established using a multiplexed RNAME only ever establish one connection to the Universal Messaging realm server. This is very useful for circumventing browser connection limits while supporting multiple sessions.

  Note: Please note that since Universal Messaging 7, nhpm and nhpsm protocols are no longer used since multiplexed sessions are supported using any protocol.

Shared Memory (SHM)

Shared Memory (SHM) is our lowest latency communications driver, designed for inter-process communication (IPC). SHM can be used for client and inter-realm (cluster and join) communication. It is operating system independent as it does not depend on any OS specific libraries, making it simple to setup and use.

As the name suggests, shared memory allocates blocks of memory that other processes can access - allowing the same physical computer to make connections without network overhead. This has many advantages, one of which is that when the realm and the data source (publisher) are located on the same physical computer, there is no network latency added between them. This results in less latency for delivery to clients.

Advantages

- Lowest latency
- No network stack involved
- Efficient use of resources with no Network IO required

Disadvantages

- Same physical machine only

- Only currently supported by certain JVMs such as Oracle JDK1.6.0_32, JDK 7 and Azul Zing. On HP-UX systems, shared memory drivers are currently not supported.

2 Management

Administration and Management

In addition to its communications APIs and features Universal Messaging provides a sophisticated collection of management tools and APIs. These tools and APIs are designed exclusively for:

- "Collection of Statistical Data from Universal Messaging" on page 28
- "Monitoring of Events" on page 28
- "Creation of Universal Messaging Resources, ACLs and Clusters" on page 28
- "Management of Configuration Parameters" on page 28
- "Seamless Integration with Third Party Enterprise Systems Management Tools" on page 29

Universal Messaging's management client, the Enterprise Manager is written using the same management APIs thus demonstrating the powerful features of these features.

Statistical Data

Through the use of the Universal Messaging management API clients can access a very detailed range of performance related data. Performance metrics can be gathered at many levels ranging from the realm throughput statistics to individual client connection round trip latency details. An example can be found in the realm log and audit listener.

Management Event Monitoring

Most client and server induced actions in Universal Messaging result in a management event being created. Asynchronous listeners can be created using the management API that enables management clients to capture these events. As an example consider a client connection to a Universal Messaging realm server. This creates a client connection event. A management client at this point might dynamically create channel resources for said client and programmatically set ACLs. An example can be found in the connection watch sample.

Resource Creation

Resources (see "Messaging Paradigms supported" on page 12) can all be created programmatically using the Universal Messaging Administration API. Coupled with statistical data and event monitoring resources can be created on the fly to support users in specific operational configurations. For example, create channel x when user x logs in OR change channel ACL when realm throughput exceeds a specific value. An ACL creation example can be found in the add queue acl sample.

Configuration Management

Every Universal Messaging Server has a number of configurable parameters. In addition specific interfaces supporting specific protocols and plugins can be added to Universal Messaging realms. Universal Messaging's configuration management feature allow

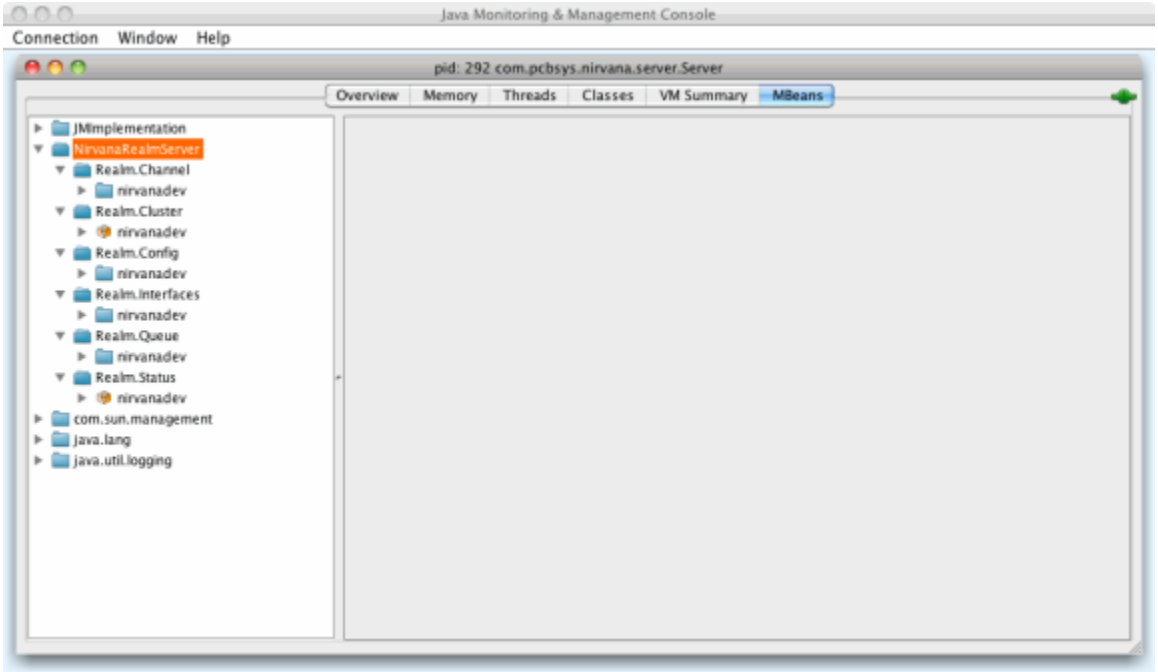
clients to snapshot configurations and generate configuration XML files. New realms can be very quickly configured with the XML files enabling the very fast bootstrapping of new environments. The Enterprise Manager documentation has an XML sample.

3rd Party Integration

While Universal Messaging's Administration API can be using directly to integrate with 3rd party products Universal Messaging servers also support JMX (see "JMX Administration and Management" on page 29) and (security permitting) can be queried by any JMX management tool.

JMX Administration and Management

In addition to Universal Messaging's Administration API, a series of JMX beans enable monitoring and management via any JMX container. The following image illustrates a JConsole view running on Mac OSX:



In order to connect to Universal Messaging over the network using JConsole or another tool that supports JMX, the following JVM System properties need to be added to the nserver.lax file:

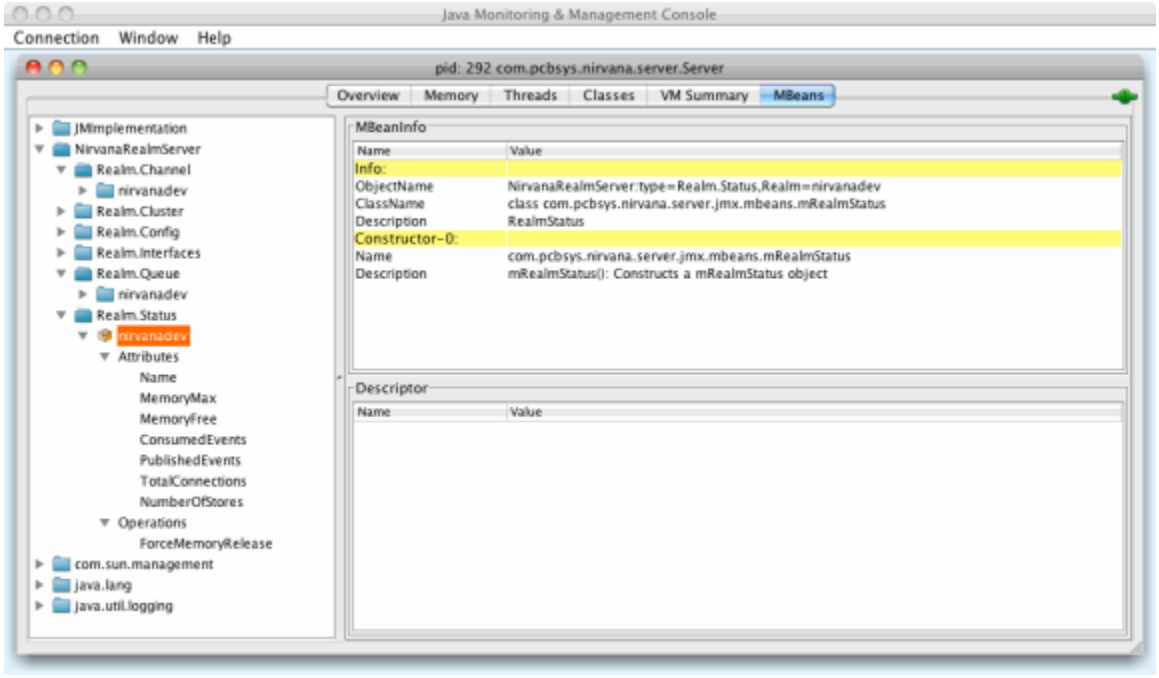
- -Dcom.sun.management.jmxremote
- -Dcom.sun.management.jmxremote.port=9999
- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

If you wish to enable authentication or SSL, please contact consult the JMX documentation or contact our support team

Universal Messaging offers the following JMX beans:

Universal Messaging Realm Status JMX bean

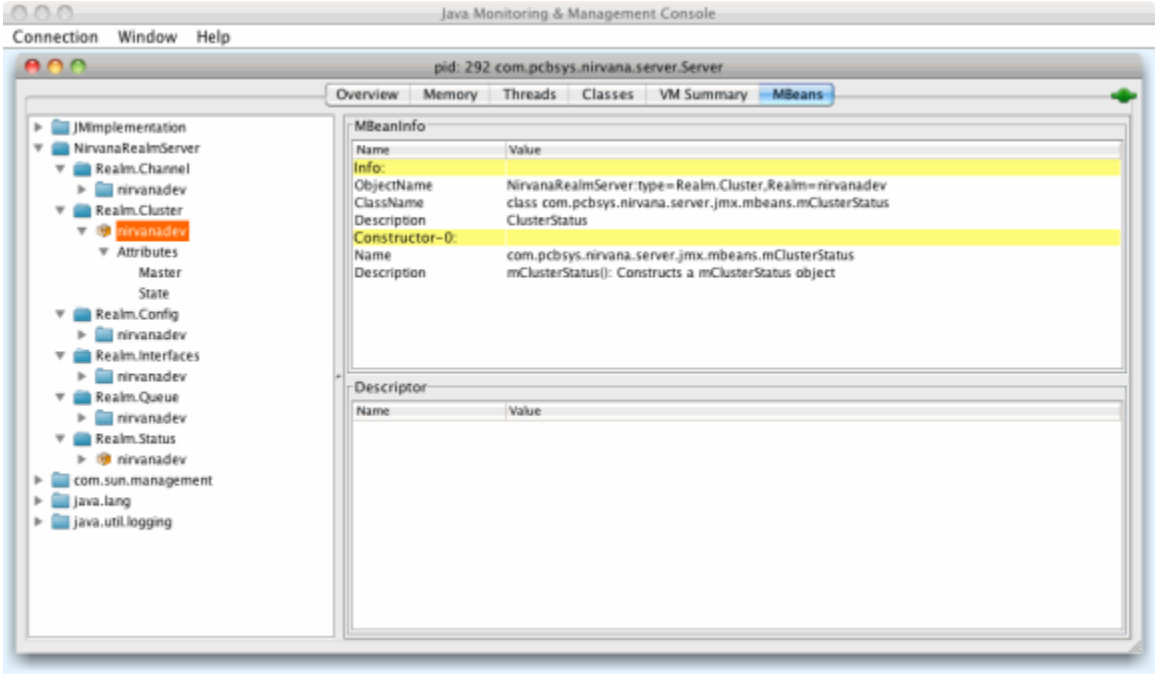
The Universal Messaging Realm Status JMX bean enables access to data visible in the Enterprise Manager Realm Status view. The following image illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



Universal Messaging Realm Cluster JMX bean

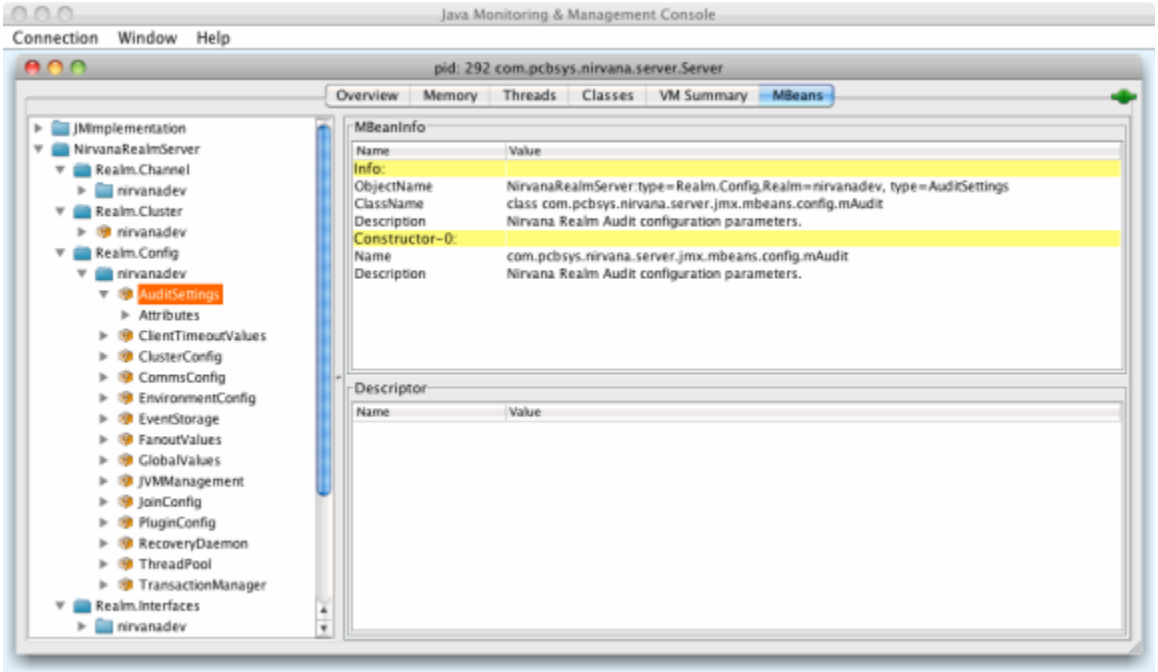
The Universal Messaging Realm Cluster JMX bean enables access to data visible in the Enterprise Manager Cluster Status view. The following image

illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



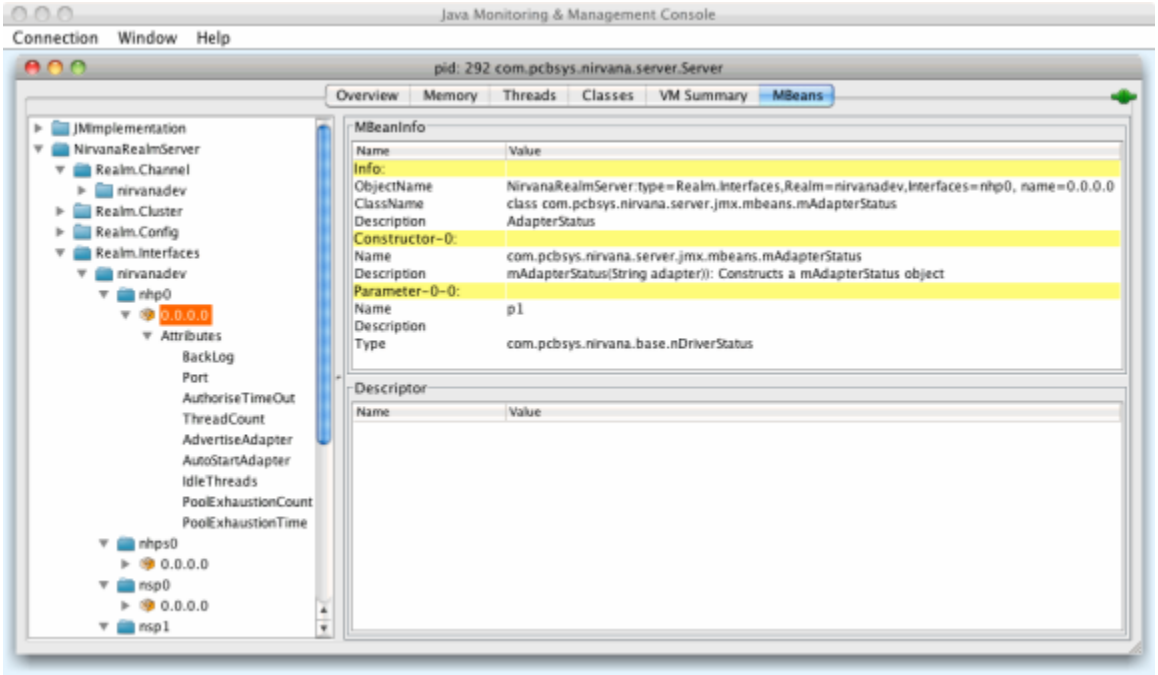
Universal Messaging Realm Configuration JMX bean

The Universal Messaging Realm Configuration JMX bean enables access to data visible in the Enterprise Manager Realm Configuration view. The following image illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



Universal Messaging Realm Interfaces JMX bean

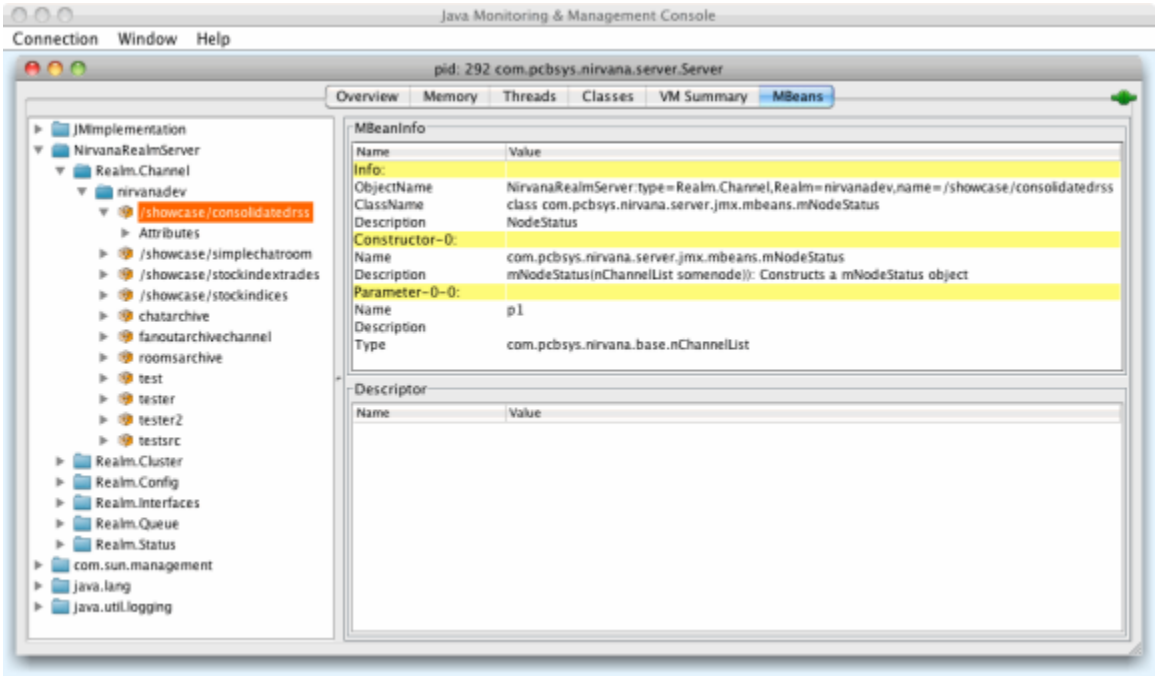
The Universal Messaging Realm Interfaces JMX bean enables access to data visible in the Enterprise Manager Realm Interfaces view. The following image illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



Universal Messaging Channel JMX bean

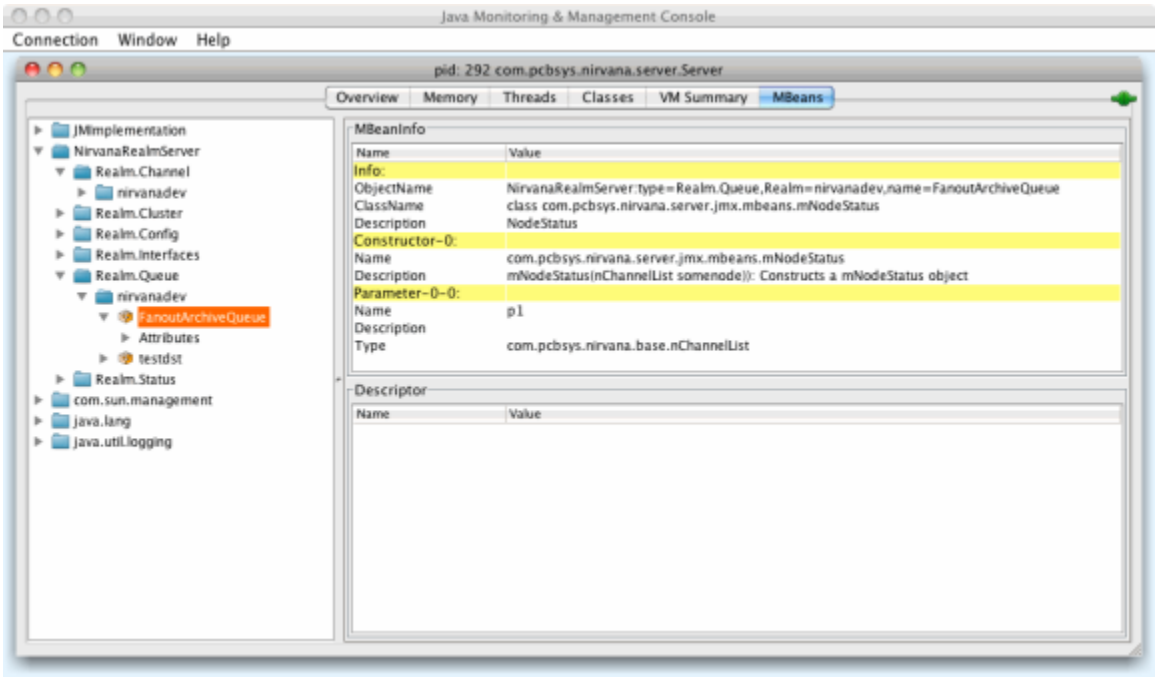
The Universal Messaging Channel JMX bean enables access to data visible in the Enterprise Manager Channel Statu view. The following image

illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



Universal Messaging Queue JMX bean

The Universal Messaging Queue JMX bean enables access to data visible in the Enterprise Manager Queue Status view. The following image illustrates how the JMX bean looks when accessed via JConsole on Mac OSX:



3 Performance, Scalability and Resilience

Performance, Scalabilty and Resilience

Performance, Scalability and Resilience are design themes that are followed in all areas of Universal Messaging's design and architecture. Specific implementation features have been introduced into server and client components to ensure that these themes remain constantly adhered to.

Performance

Universal Messaging is capable of meeting the toughest of Low Latency and High Throughput demands. Universal Messaging's server design, sophisticated threading models and heavily optimized IO subsystem ensures peak performance. Multicast and shared memory communication modes allow Universal Messaging to consistently achieve low microsecond latencies. Universal Messaging is constantly being benchmarked by clients and other 3rd parties and continues to come out on top.

The benchmarking section provides detailed information on performance in a variety of scenarios. Information on performance tuning is also available on the website, helping clients achieve optimal performance in their deployments.

Scalability

Scalability in terms of messaging Middleware typically means supporting large numbers of concurrent connections, something Universal Messaging does out of the box. However in defining truly global enterprise applications a single system often needs to scale across more than one processing core, often in more than one geographic location.

Universal Messaging servers can be configured in a variety of ways to suit scalability (and resilience) requirements. Multiple Universal Messaging servers can exist in a single federated name space. This means that although specific resources can be put onto specific Universal Messaging realm servers any number of resources can be managed and access centrally from a single entry point into Universal Messaging's federated namespace. In addition to high availability and resilience features Universal Messaging Clusters also offer a convenient way to replicate data and resources among a number of realm servers thus supporting higher rates of concurrent connections.

Resilience

Business contingency and disaster recovery planning demand maximum availability from messaging Middleware systems. Universal Messaging provides a number of server and client features to ensure data always remains readily accessible and that outages are transparent to clients.

Universal Messaging Clusters replicate all resources between realms. Channels, Topics, Queues and the data held within each one is always accessible from any realm server in the cluster.

Universal Messaging clients are given a list of Universal Messaging realms in the cluster and automatically move from one to another if a problem or outage occurs.

Clustering

Clusters: An Overview

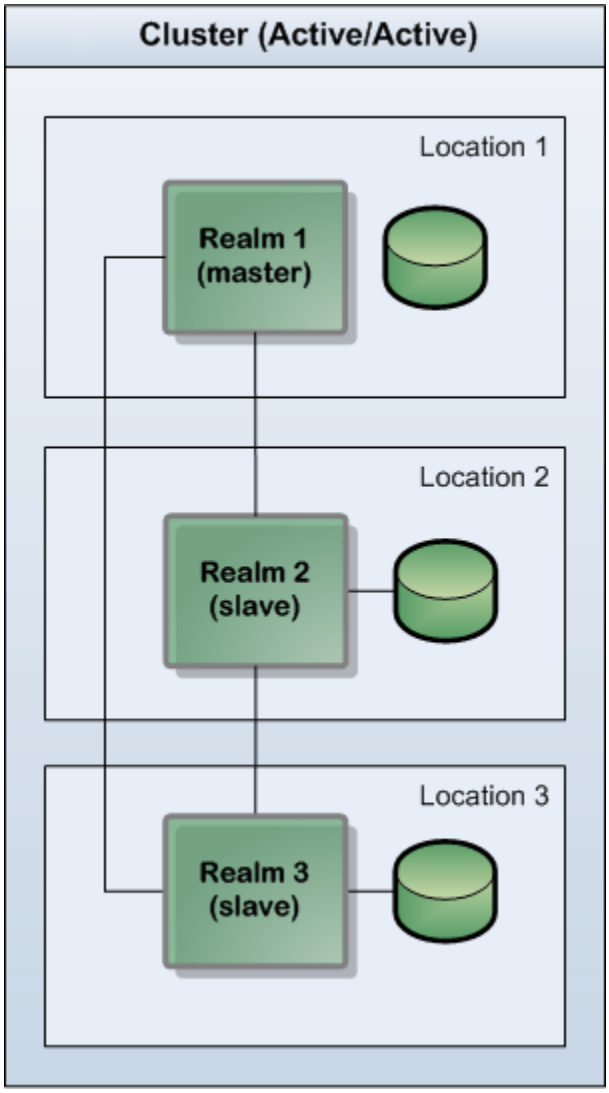
Universal Messaging provides built-in support for clustering in the form of Universal Messaging Clusters (see "Server Concepts" on page 42) and Universal Messaging Clusters with Sites (see "Clusters with Sites" on page 52).

Universal Messaging clusters provide support for business contingency and disaster recovery natively or in conjunction with existing BCP enterprise solutions.

From a client perspective a cluster offers *resilience* and *high availability*. Universal Messaging clients automatically move from realm to realm in a cluster as required or when specific realms within the cluster become unavailable to the client for any reason. The state of all client operations is maintained so a client moving will resume whatever operation they were previously carrying out.

Three Approaches to Clustering

As mentioned above, Universal Messaging provides built-in support for clustering in the form of Universal Messaging Clusters and Universal Messaging Clusters with Sites. Universal Messaging clients can also use the same clustering functionality to communicate with individual, non-clustered Universal Messaging Realms in Shared Storage (see "Shared Storage Configurations" on page 57) server configurations:

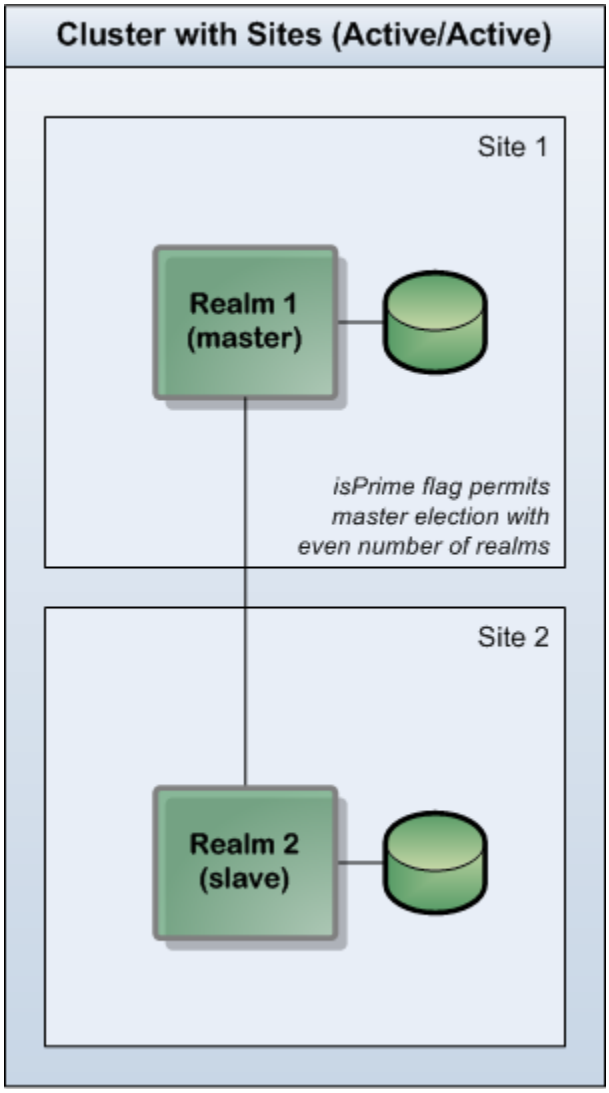


Universal Messaging Clusters

- Active/Active
- Transparent Client Failover
- Transparent Realm Failover
- Provides Load Balancing and Scalability

Universal Messaging Clusters are our recommended solution for high availability and redundancy. State is replicated across all active realms.

With 51% of realms required to form a functioning cluster, this is an ideal configuration for fully automatic failover across a minimum of three realms.

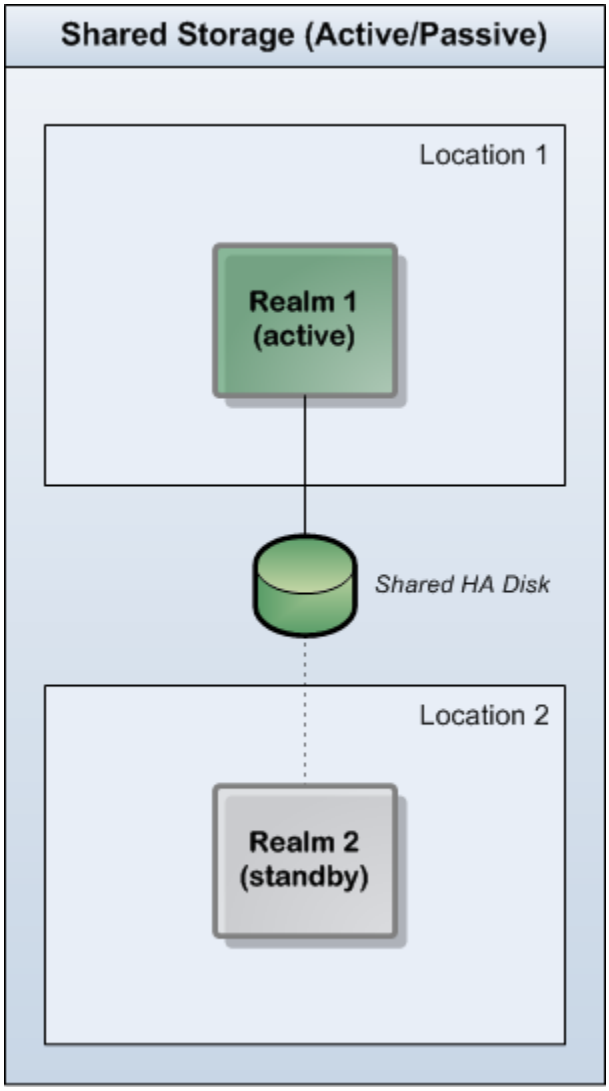


Universal Messaging Clusters with Sites

- Active/Active
- Transparent Client Failover
- Semi-Transparent Realm Failover
- Provides Load Balancing and Scalability

Universal Messaging Clusters with Sites provide most of the benefits of Universal Messaging Clusters but with less hardware and occasional manual intervention.

This configuration is designed for an even number of realms across two Sites (such as Production and DR). Failover is automatic should the "Non-Prime" Site fail, and requires manual intervention only if the Prime Site fails.



Shared Storage Configurations

- Active/Passive
- Transparent Client Failover
- Manual Realm Failover
- No Load Balancing Features

As an alternative to native Universal Messaging Clusters, Shared Storage (see "Shared Storage Configurations" on page 57) configurations can be deployed to provide disaster recovery options.

This approach does not make use of Universal Messaging's built-in Cluster features, but instead allows storage to be shared between multiple realms - of which only one is active at any one time.

In general, we recommend the use of Universal Messaging Clusters or Universal Messaging Clusters with Sites.

Client Concepts

A Universal Messaging client, whether using the Universal Messaging API or JMS, accesses Universal Messaging realms and their resources through a custom URL called an RNAME. When accessing resources in a cluster, clients use a comma separated array of RNAMES. This comma separated array can be given to the client dynamically when the client connects to any member of a cluster. If a connection is terminated unexpectedly the Universal Messaging client automatically uses the next RNAME in its array to carry on.

For example, if we have a cluster consisting of 3 realms, your nirvana nSession can be constructed using the 3 RNAME URLs associated with each of the realms in the cluster.

Once connected to a realm in a cluster, you can then obtain references to nChannel and nQueue objects (or in JMS, create a Session followed by a topic or queue).

Each event/message within Universal Messaging is uniquely identified by an event id regardless of whether it is stored on a channel, topic or queue. A clustered channel, topic or queue guarantees that every event published to it via any realm within the cluster will be propagated to every other realm in the cluster and will be identified with the same unique event id. This enables clients to seamlessly move from realm to realm after disconnection and ensure that they begin from the last event consumed based on this unique event id.

For scenarios where failover is handled at the network level, Universal Messaging sessions can be moved to alternate servers transparently without the use of multiple RNAMES.

Client Failover Using Multiple RNAMES

Using an array of RNAME URLs allows client applications to seamlessly failover to different cluster nodes without the use of any third party failover software.

For example, in a three realm clustered scenario, a client's RNAME string may contain the following RNAME URLs.:

```
nsp://host1:9000,nsp://host2:9000,nsp://host3:9000
```

When we first connect, the rnames[0] will be used by the session, and the client application will connect to this realm. However, should we disconnect from this realm, for example if host1 crashes, the client API will automatically reconnect the client application to the cluster member found at rnames[1].

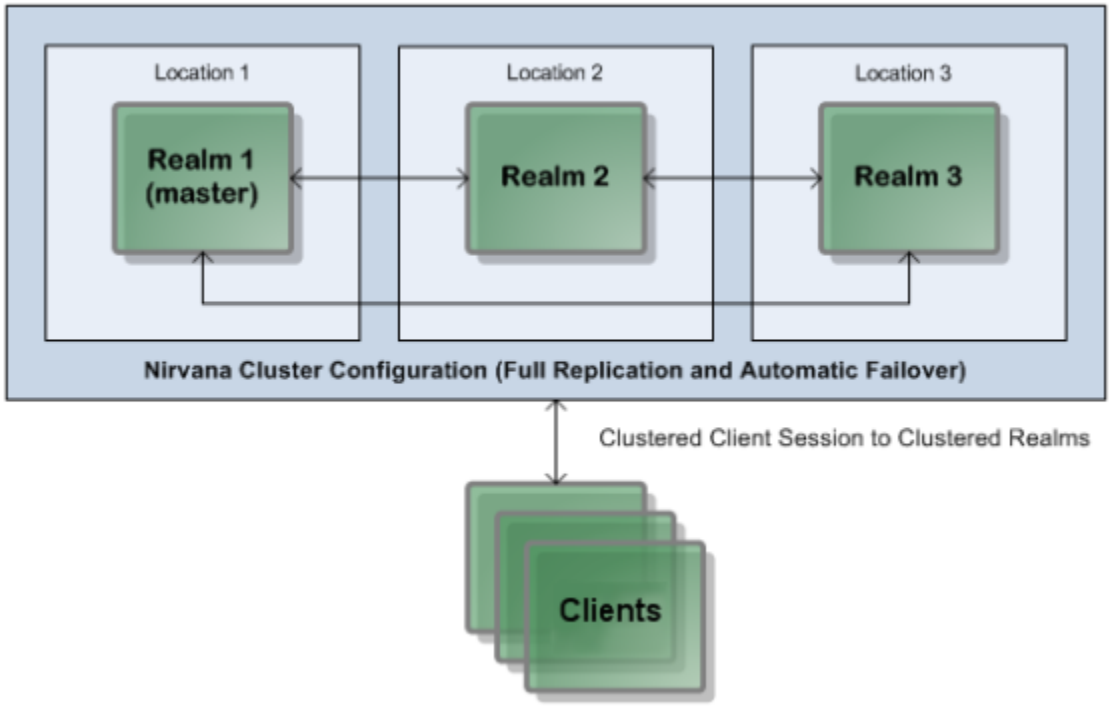
Clustered Server Concepts

Server Concepts

A Universal Messaging Cluster is a collection of Universal Messaging Realms (servers) that contain common messaging resources such as channels/topics or queues. Each clustered resource exists in every Realm within the cluster. Whenever the state of a clustered resource changes, the state change is updated on all realms in the cluster.

Clustering also offers a convenient way to replicate content between servers and ultimately offers a way to split large numbers of clients over different servers in different physical locations.

The following diagram represents a typical three-realm cluster distributed across three physical locations:



Three-realm cluster over three locations.

Clustered Resources

A Universal Messaging Realm server is a container for a number of messaging resources that can be clustered:

- Universal Messaging Channels
- JMS Topics

- Universal Messaging Queues
- JMS Queues
- DataGroups
- Access Control Lists
- Resource Atrributes including Type, Capacity, TTL
- Client Transactions on Universal Messaging Resources

Within the context of a cluster a single instance of a channel, topic or queue can exist on every node within the cluster. When this is the case all attributes associated with the resource are also propagated amongst every realm within the cluster. The resource in question can be written to or read from any realm within the cluster.

The basic premise for a Universal Messaging Cluster is that it provides a transparent entry point to a collection of realms that share the same resources and are, in effect, a mirror image of each other.

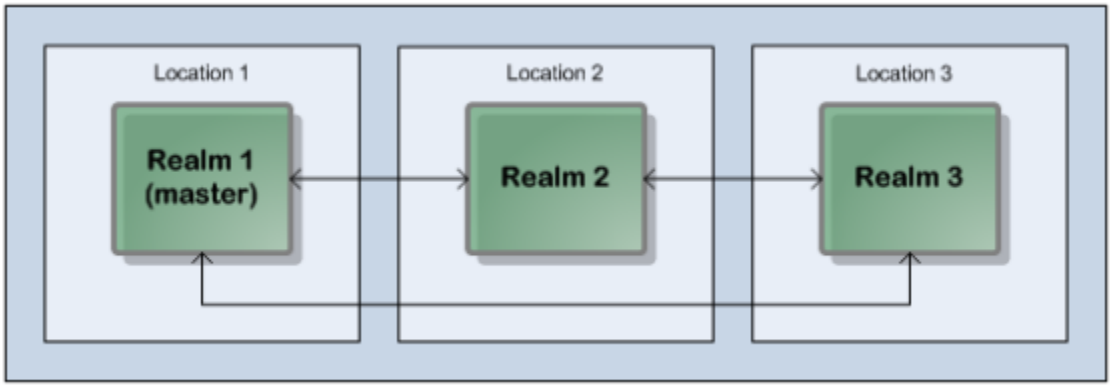
A Universal Messaging cluster achieves this by the implementation of some basic concepts described below:

- " Masters & Slaves" on page 43
- "Quorum" on page 45
- "Master Election" on page 46
- "Message Passing" on page 50
- " Outages & Recovery" on page 51
- " Clustered Resources" on page 51

Masters and Slaves

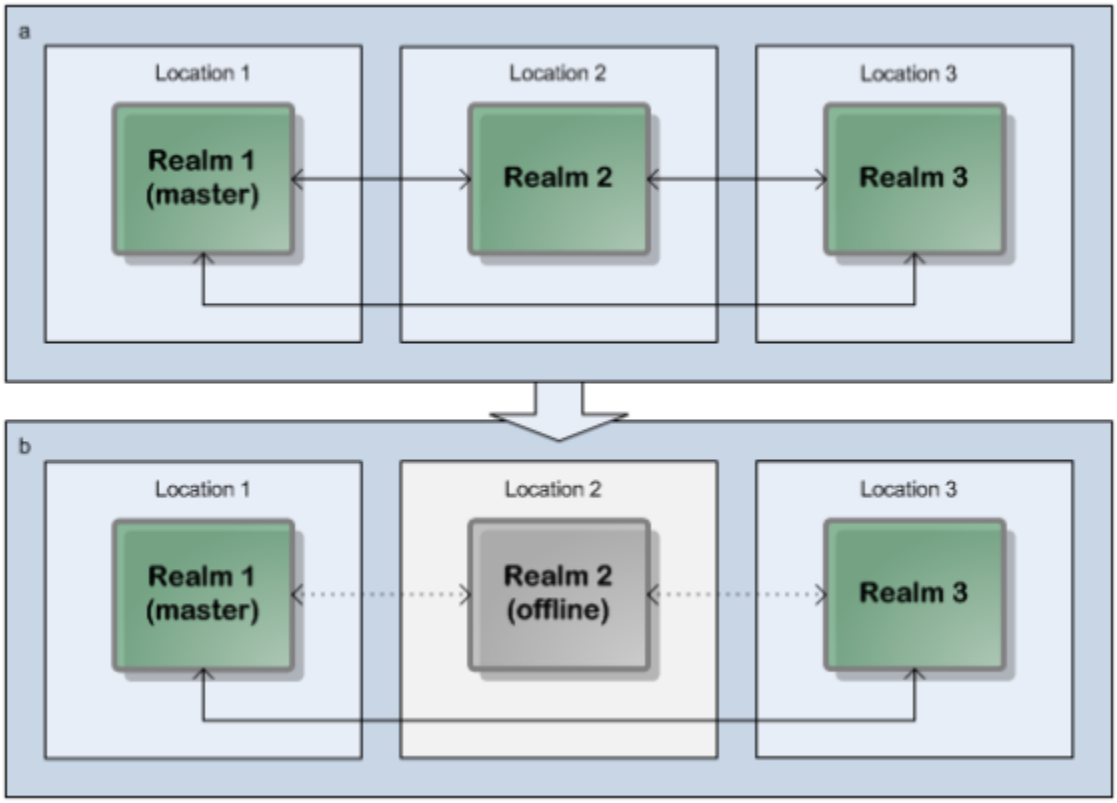
As explained in the Clustering Overview (see "Clusters: An Overview" on page 37), a *cluster* is a collection of Universal Messaging Realm Servers (realms).

Each cluster has one realm which is elected as master, and all other realms are deemed slaves. The master is the authoritative source of state for all resources within the cluster.



Three-realm cluster over three locations: one master and two slaves.

Should a realm or location become unavailable for any reason, the cluster's remaining realms should be able to carry on servicing clients:



Three-realm cluster over three locations: cluster continuation with one missing slave.

Note: Dotted lines represent interrupted communication owing to server or network outages.

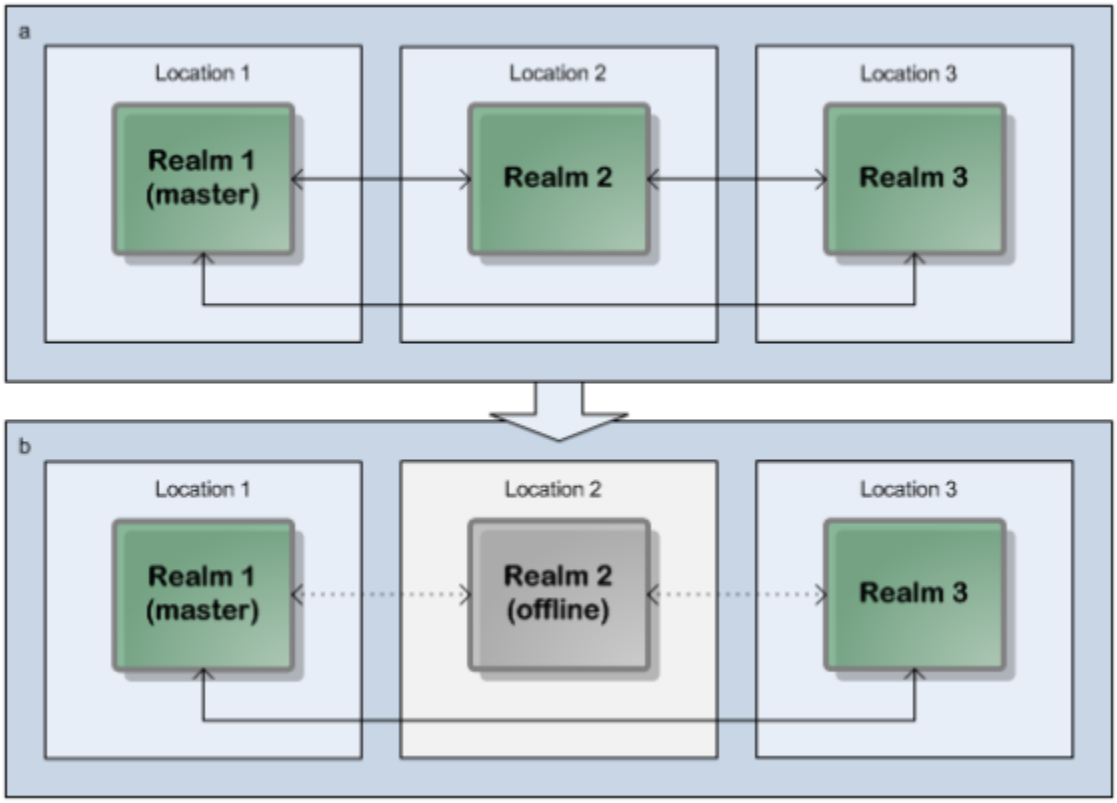
For publish/subscribe resources (see "Messaging Paradigms supported" on page 12), each published event will be allocated a unique event id by the master, which is then propagated to each slave.

Quorum

Quorum is the term used to describe the state of a fully formed cluster with an elected master. In order to achieve quorum, certain conditions need to be met. Most importantly, *51% or more* of the cluster nodes must be *online and intercommunicating* in order for quorum to be achieved.

Example: Quorum in a Three-Realm Cluster

In this example, we examine a three-realm cluster, distributed across three physical locations (such as a primary location and two disaster recovery locations). The 51% quorum requirement means there must always be a minimum of two realms active for the cluster to be online and operational:



Three-realm cluster over three locations: a 67% quorum is maintained if one location/realm fails.

Note: Dotted lines represent interrupted communication owing to server or network outages.

Split-Brain Prevention

Quorum, in conjunction with our deployment guidelines, prevents *split brain* (the existence of multiple masters) scenarios from occurring. By requiring at least 51% of realms to be online and intercommunicating before an election of a new master realm

can take place, it is impossible for two sets of online but not intercommunicating realms to both elect a new master.

To simplify the reliable achievement of quorum, we always recommend a cluster be created with an *odd number of member realms*, preferably in at least three separate locations. In the above three-realm/three-location example, should any one location become unavailable, sufficient realms remain available to achieve quorum and, if necessary, elect a new master (see "Election of a new Master" on page 46).

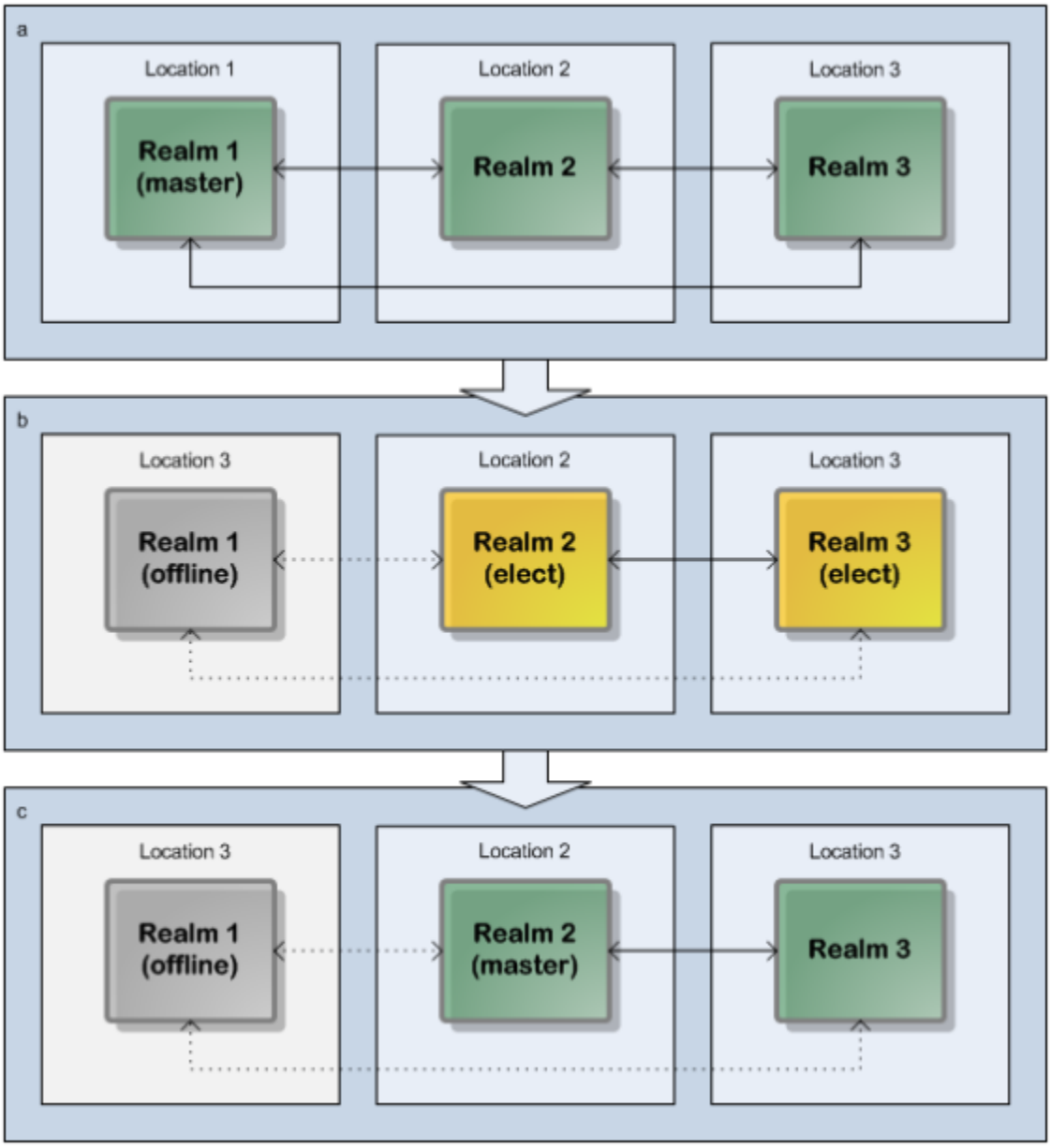
Election of a new Master

A master realm may unexpectedly exit or go offline owing to power or network failure. In this event, if the remaining cluster nodes "achieve 51% or greater quorum" on page 45, they will elect a new master realm between them and continue to function as a cluster.

The process of the master election involves all remaining realms in the cluster. Each remaining realm submits a vote across the cluster that results in the new master once all votes are received and the number of votes is greater than or equal to 51% of the total cluster members.

Example: Master Election in a Three-Realm Cluster

In this example, we examine a three-realm cluster, distributed across three physical locations (such as a primary location and two disaster recovery locations). The Master Realm has failed, but the reaming two realms achieve a quorum of 67% (which satisfies the 51% quorum minimum requirement), so will elect a new Master and continue operating as a cluster:



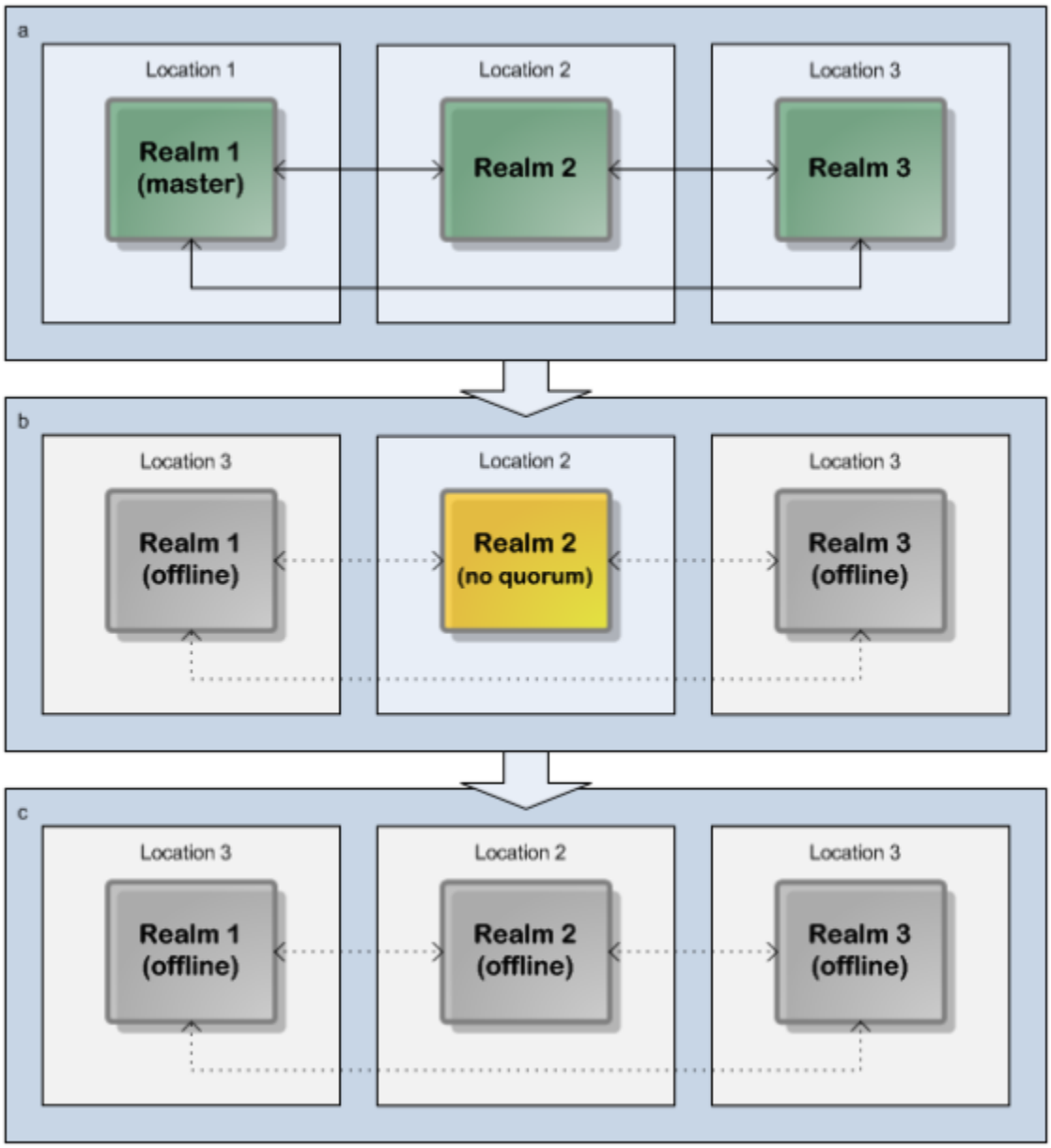
Three-realm cluster over three locations: quorum permits election of a new Master and cluster continuation.

Note: Dotted lines represent interrupted communication owing to server or network outages.

Examples: Insufficient Quorum for Master Election

In this example, we again examine a three-realm cluster, distributed across three physical locations. In this case, both the Master realm and one slave realm has failed, so the reaming realm represents only 33% of the cluster members (which does not satisfy the 51% quorum minimum requirement). As a result, it cannot elect a new Master, but

will instead disconnect its clients and attempt to re-establish communications with the other realms with the aim of reforming the cluster:

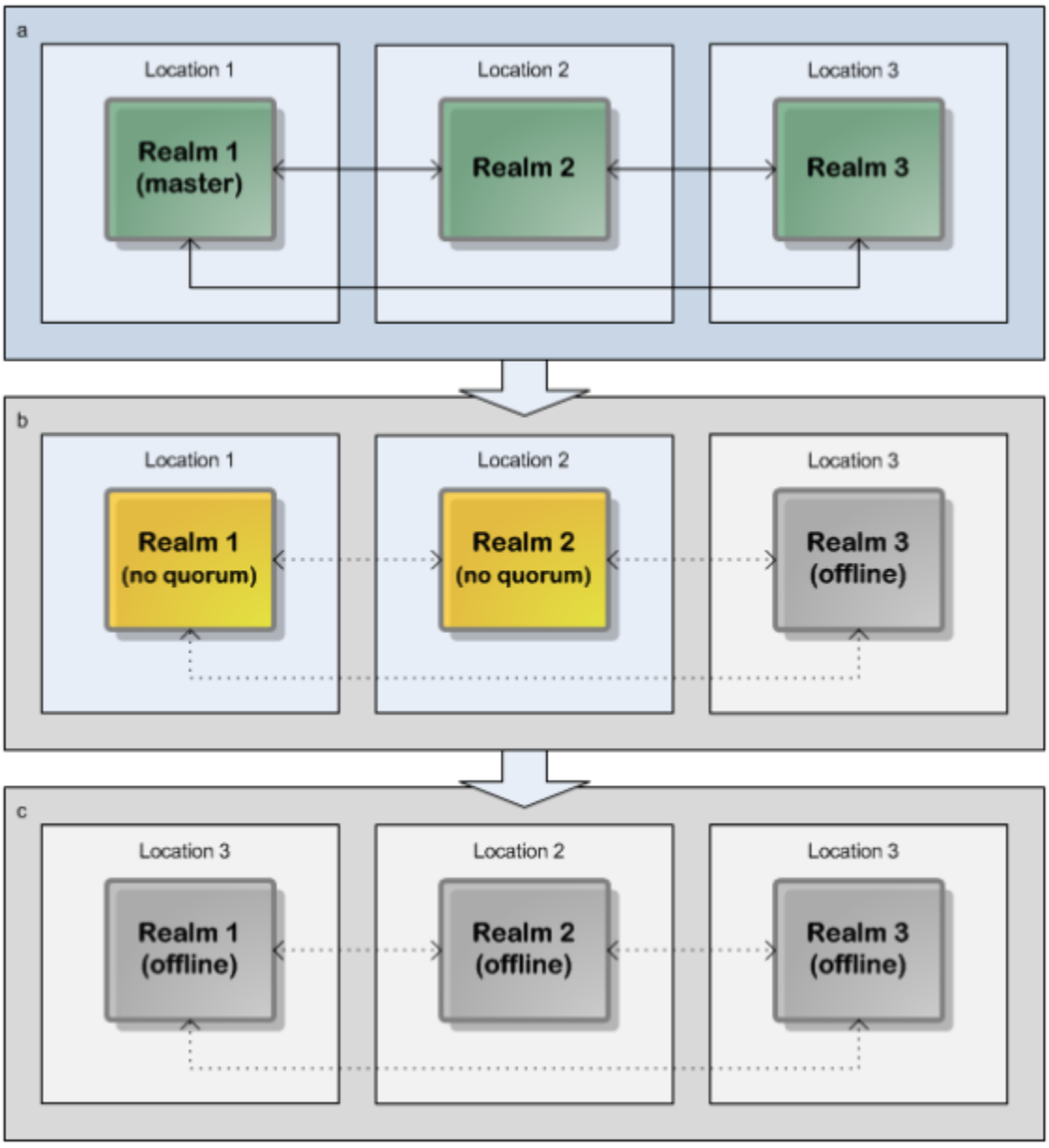


Three-realm cluster over three locations: insufficient quorum prevents election of a new Master or cluster continuation.

Note: Dotted lines represent interrupted communication owing to server or network outages.

A second example highlights both a realm's perspective of quorum, and prevention of split-brain (multiple masters) configurations. In this example, one realm server has failed, while two realms are still active. Also, in this particular example, we assume network connectivity between all realms has failed. As far as each active realm is concerned, therefore, it is the only functioning realm, representing only 33% of the

cluster members. As you might expect, this is insufficient for the 51% required quorum, and is thus also insufficient for the continued operation of the cluster. Both "active" realms will disconnect their clients and attempt to re-establish communications with the other realms, with the aim of achieving quorum and reforming the cluster:



Three-realm cluster over three locations: lack of network interconnectivity prevents cluster continuation.

Note: Dotted lines represent interrupted communication owing to server or network outages.

Note that in the above example, although the two active realms were unable to communicate with each other, it is possible that they were able to communicate with clients. Here, the importance of the 51% quorum rule can be clearly seen: without it,

both realms would have elected themselves master, which could have led to logically unresolvable conflicts once the cluster was reformed. It is therefore essential that the cluster was disabled until such time as a 51% quorum could be achieved.

Clearly, for the situation where we have a total cluster failure, i.e. all realms or locations are offline, the cluster is also deemed inaccessible.

Message Passing

Message passing between cluster realms enables state to be maintained across the member realms. The complexity of the message passing differs somewhat depending on the scenario.

Message Passing in Topics

It is possible to publish to topics on either master or slave realm nodes.

When we publish to the master and subscribe from both master and slave nodes, the master will simply pass the event onto each slave for delivery with the correct event id, and each slave will maintain the same event id as set by the master.

When publishing to a topic on a slave node, the slave has to contact the master for the correct event id assignment before the event is then propagated to each slave.

Message Passing in Queues

When using queues, the message passing is much more complex, since each read is destructive (i.e. it is immediately removed from the queue after it is delivered successfully).

Consider a situation where we have a cluster of 5 realms, and each realm has a consumer connected to a queue, *somequeue* . Assume we publish 2 events directly to the master realm's *somequeue* object.

If the first event happens to be consumed by a consumer on the master realm, each slave realm will be notified of the consumption of the event from *somequeue* and thus remove the event from its own local copy of *somequeue* .

If the next event is consumed by a consumer on some slave realm, then the slave realm will notify the master of the event being consumed. The master will update its local *somequeue* , and then propagate this change to all other slave realms (to update their own local copies of *somequeue*).

Both the Universal Messaging API and the JMS standard define transactional semantics for queue consumers which add to the complexity of the message passing. For example, a consumer may effectively roll back any number of events it has consumed but not acknowledged. When an event is rolled back, it must then be re-added to the queue for re-delivery to the next available queue consumer (which may exist on any of the slave realms). Each event that is rolled back requires each slave realm to maintain a cache of the events delivered to transactional consumers in order for the event to be effectively restored should it be required. The state of this cache must also be maintained identically by all cluster members. Once an event is acknowledged by the consumer (or

the session is committed), these events are no longer available to any consumer, and no longer exist in any of the cluster member's queues.

By now it should be clear that certain scenarios will require more message passing between cluster members than others. Nevertheless, there are a huge number of benefits associated with using clusters in terms of scalability.

Outages and Recovery

Should any cluster member realm exit unexpectedly or become disconnected from the remaining cluster realms, it needs to fully recover the current cluster state as soon as it restarts or attempts to rejoin the cluster.

When a cluster member rejoins the cluster, they automatically move into the *recovery state* until all local stores are recovered and its state is fully validated against the current master realm.

In order to achieve this, each clustered resource must recover the state from the master. This involves a complex evaluation of its own local stores against the master realm's stores to ensure that they contain the correct events, and that any events that no longer exist in any queues or topics are removed from its local stores. With queues for example, events are physically stored in sequence, but may be consumed non-sequentially (for example using message selectors that would consume and remove, say, every fifth event). Such an example would result in a fairly sparse and fragmented store, and adds to the complexity of recovering the correct state. Universal Messaging clusters will, however, automatically perform this state recovery upon restart of any cluster member.

Creating Clustered Resources

Channels, topics and queues can be created *cluster wide* , which ensures state is maintained across the cluster as described in the Clustering Overview (see "Clusters: An Overview" on page 37). Once a channel, topic or queue is created as cluster wide, any operations upon that resource are also propagated to all cluster members.

There are a number of ways to create cluster resources once you have created your cluster. The Enterprise Manager application is a tool that provides access to all resources on any realm within a cluster or on stand alone realms. This graphical tool is written using the Universal Messaging Client and nAdmin APIs and allows resources to be created, managed and monitored from one central point.

Because the Enterprise Manager tool is written using our own APIs, all operations you can perform using the tool are also available programmatically using the Universal Messaging APIs, allowing you to write customized applications for specific areas of interest within a realm or a cluster.

Once a realm is part of a cluster, you can centrally manage its resources and configuration. For example, realm access control lists (ACLs) can be updated on any member realm and the change will be propagated to all other member realms. Clustered channels and queue ACLs can also be changed on any cluster member and the change is then propagated to the other cluster members. Configuration changes, (in the Enterprise

Manager tool) can also be made on one realm and propagated to all other realms in the cluster.

This provides a powerful way of administering your entire Universal Messaging environment and its resources.

Inter-Cluster Connections

Inter-cluster connections can be formed between clusters in order to allow joins to be created between stores on these separate clusters. Inter-cluster connections are bi-directional, allowing joins to be formed between clusters in either direction once the inter-cluster connection has been set up.

In this way, these connections can facilitate inter-cluster routing of messages. Inter-cluster connections do not, however, provide full namespace federation across remote clusters. They are designed to support inter-cluster message propagation on explicitly joined stores, rather than mounting clusters in the namespace of remote clusters, as in realm federation.

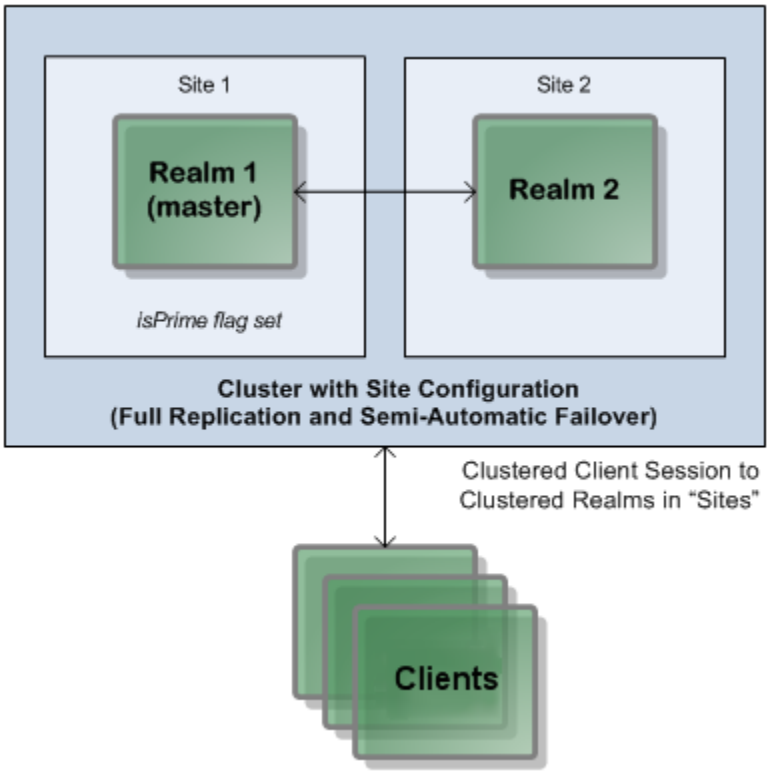
It is important to note that the use of inter-cluster connections is exclusive of the use of realm federation, and they *cannot be used together* in any combination.

Inter-Cluster connections can be added either using the Enterprise Manager or programmatically.

Clusters with Sites

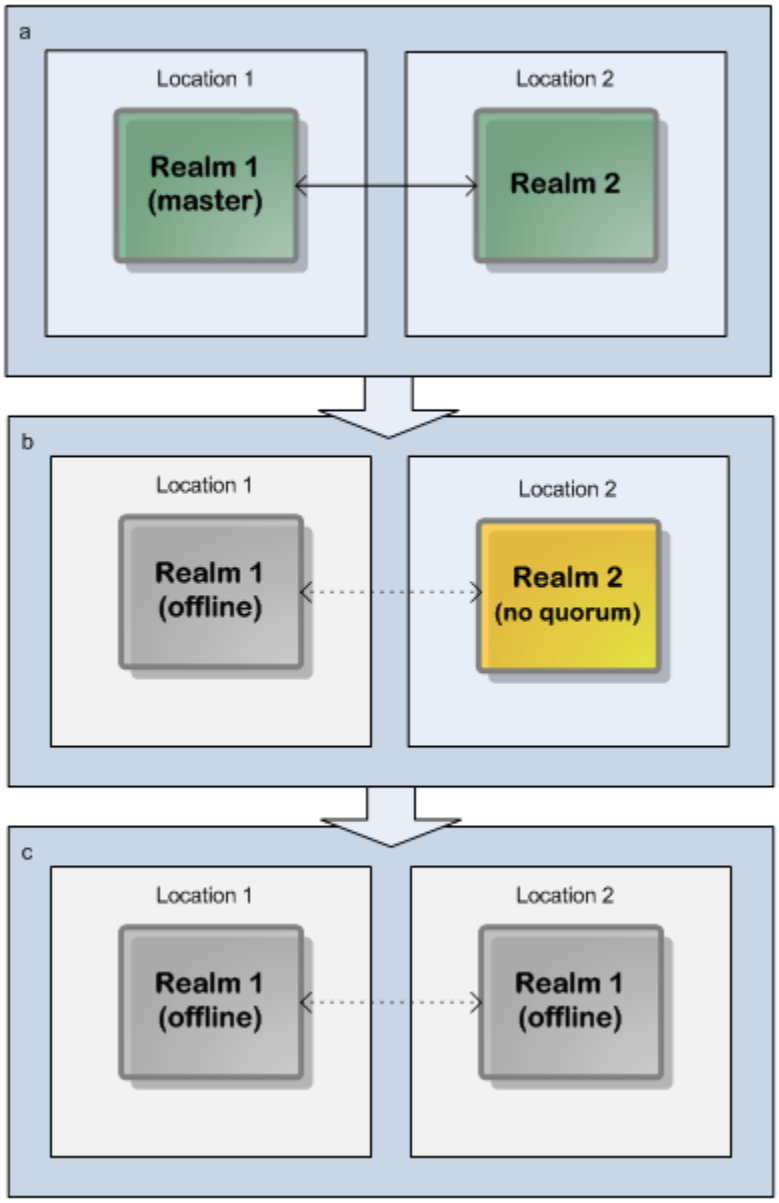
Sites - an exception to the Universal Messaging Cluster Quorum Rule (see "Quorum" on page 45*).*

Although our recommended approach to deploying a cluster is a minimum of three locations and an odd number of nodes across the cluster, not all organizations have three physical locations with the required hardware. In terms of BCP (Business Continuity Planning), or DR (Disaster Recovery), organizations may follow a standard approach with just two locations: a primary and backup:



Two-realm cluster over two sites, using Universal Messaging Clusters with Sites.

With only two physical sites available, the Quorum rule of 51% or more of cluster nodes being available is not reliably achievable, either with an odd or even number of realms split across these sites. For example, if you deploy a two-realm cluster, and locate one realm in each available location, then as soon as either location is lost, the entire cluster cannot function because of the 51% quorum rule:



Two-realm cluster over two locations: a 51% quorum is unachievable if one location/realm fails.

Note: Dotted lines represent interrupted communication owing to server or network outages.

Similarly, if you deployed a three-node cluster with one realm in Location 1 and two in Location 2, and then lost access to Location 1, the cluster would still be available; if, however, you lost Location 2, the cluster would not be available since only 33% of the cluster's realms would be available.

This problem has been addressed through the introduction of *Universal Messaging Clusters with Sites*. The basic concept of Sites is that if only two physical locations are available, and Universal Messaging Clustering is used to provide High Availability and DR, it should be possible for one of those sites to function with less than 51% of the

cluster available. This is achieved by allowing an additional vote to be allocated to either of the physical locations in order to achieve cluster quorum.

Example: Achieving Quorum using Universal Messaging Clusters with Sites

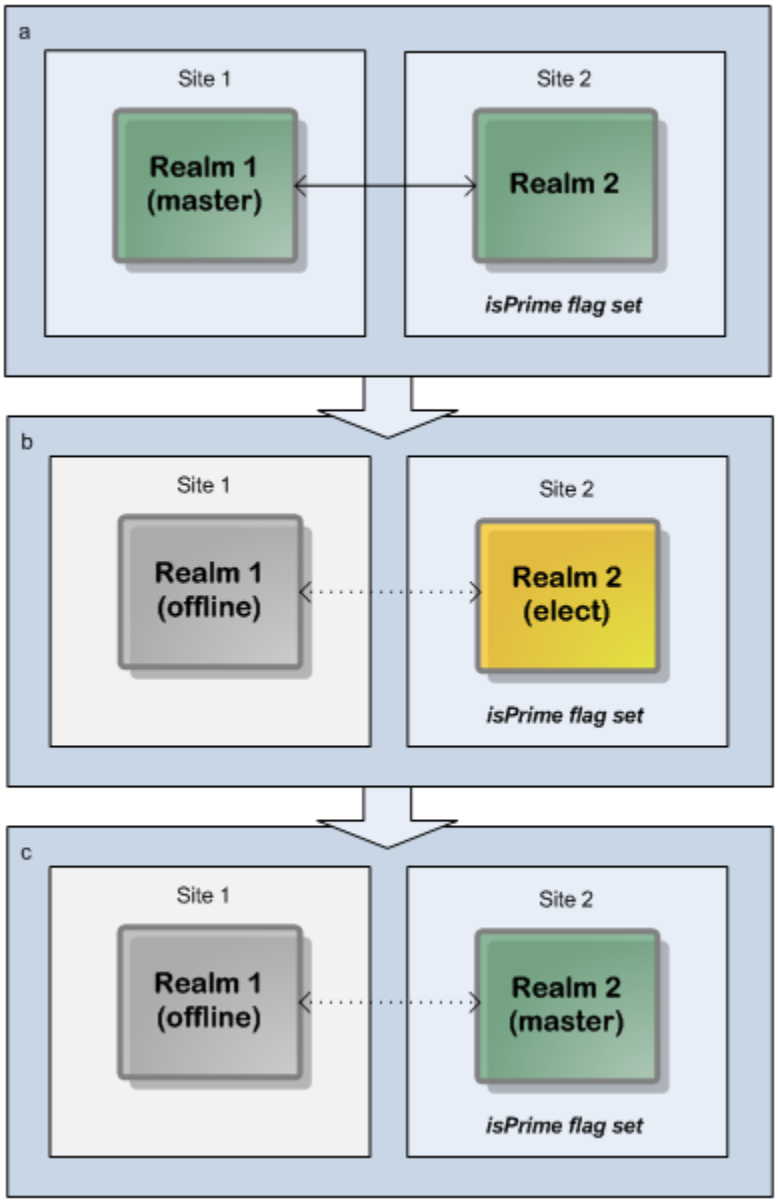
Consider an example scenario where there are two physical locations: the default production site, and a disaster recovery site.

We can deploy a cluster of two realms: one in the production site and one in the DR site. Normally this configuration wouldn't be able to satisfy the 51% quorum rule in the event of the loss of one location/realm.. However, within the Universal Messaging Admin API, and specifically within a cluster node, it is now possible to define individual *Site* objects and allocate each realm within the cluster to one of these physical sites. Each defined site contains a list of its members, and a flag to indicate whether the site as a whole can cast an additional vote. This flag is known as the *isPrime* flag.

We can thus define two sites in our cluster:

- production site
- disaster recovery site with *isPrime* flag set

In a disaster recovery situation, where the production site is lost, the DR site will achieve quorum with only one of the two nodes available because the *isPrime* flag provides an additional vote for the DR site:



Two-realm cluster over two sites: Sites make a 51% quorum achievable if one location/realm fails.

Note: Dotted lines represent interrupted communication owing to server or network outages.

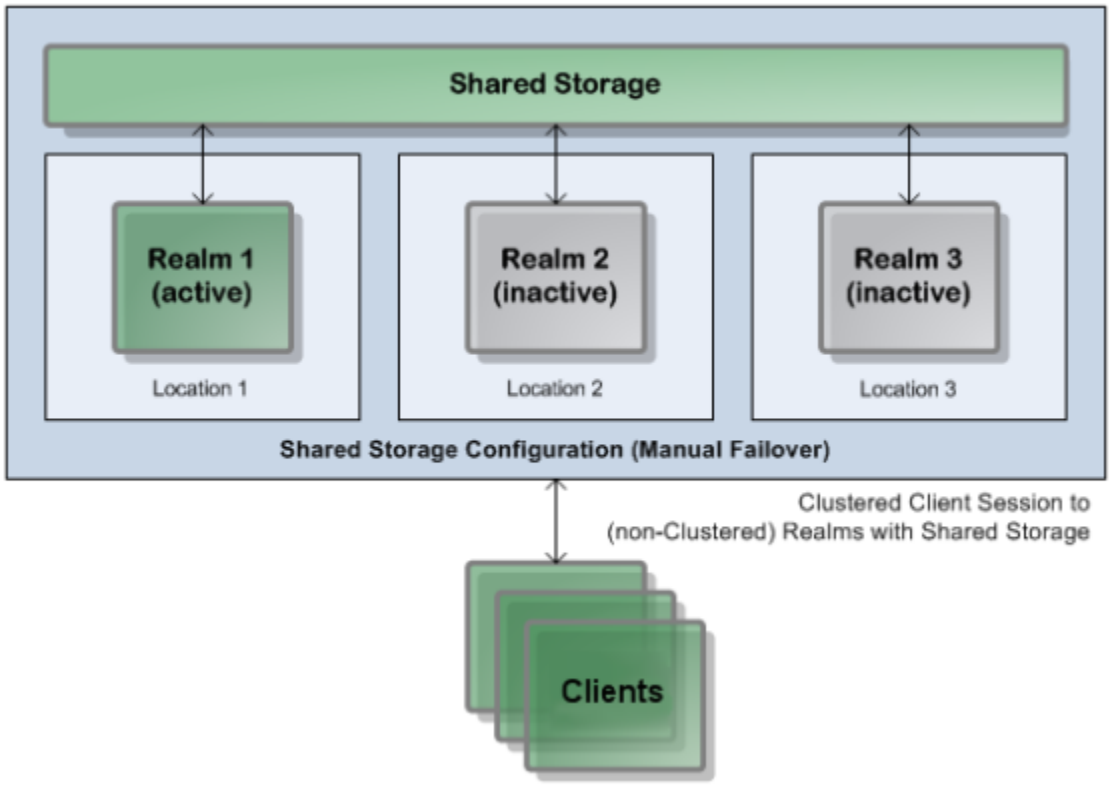
If, on the other hand, the disaster recovery site (which has the *isPrime* flag set) is lost, then manual intervention would be required to set the *isPrime* flag on the production site instead, so that the production site alone can achieve quorum and the cluster can still operate.

Note that this example uses only one realm per site for simplicity. The same technique can be used for sites with as many realms as required.

Setting the *isPrime* flag can be achieved programmatically using the Universal Messaging Admin API or using the Enterprise Manager tool. In these situations it is always advisable to discover the cause of the outage so any changes to configuration are made with the relevant facts at hand.

Shared Storage Configurations

Universal Messaging Realms may be set up to share a single virtual or physical disk, in *Shared Storage* configurations. While not technically a Cluster (see "Server Concepts" on page 42), Shared Storage configurations do provide a basic mechanism for rapid failover between Realms:



Single active realm with two inactive backup realms in a Shared Storage configuration.

Note that simple Shared Storage configurations do not provide any form of load balancing or scalability, and rely on administrators to perform manual or scripted failover processes. We therefore strongly recommend the use of native Universal Messaging Clusters (see "Server Concepts" on page 42) over Shared Storage configurations.

Setting Up a HA Failover Cluster

Universal Messaging servers can be clustered together to form part of a single logical High Availability (HA) server.

Server Configuration

As an example, let us look at the steps involved in creating a simple 2-node cluster:

- *Realm1* running on *host1.mycompany.com*
- *Realm2* running on *host2.mycompany.com*

Firstly, use the Enterprise Manager tool to create a cluster (see "Clusters: An Overview" on page 37) with Realm1 and Realm2.

Next, create cluster channels and cluster queues, which ensures these objects exist in both realm servers.

Client Configuration

The next step is to setup your clients so that they are configured to swap between Realm1 and Realm2 in case of failures.

When you initialise a client session with a Universal Messaging server, you provide an array of RNAME URLs (see "Communication Protocols and RNAMEs" on page 22) as the argument to the nSessionAttributes object. This ensures that if you lose the connection to a particular Universal Messaging Realm, the session will be automatically reconnected to the next realm in the RNAME array.

Using the configuration above where cluster channels exists on each Realm, disconnected clients will automatically continue publishing/subscribing to the channel or queue on the newly connected realm.

For example, to use the two Realms described above for failover you could use the following as your RNAME value using a comma separated list of individual RNAMEs:

```
RNAME=nhp://host1.mycompany.com:80,nsp://host2.mycompany.com:9000
```

In this example, note the optional use of different protocols and ports in the specified RNAMEs.

Failover/HA Scenarios

If all subscribers and publishers are configured in this way, then failover is provided in each of the following scenarios:

Scenario I: Subscriber loses connection to a Realm

if a subscriber is consuming data from the sales channel on Realm1 and loses its connection it will automatically attempt to connect to its additional RNAMES (in this case `nsp://host2.mycompany:9000`) and resume consuming from where it left off.

Scenario II: Publisher loses connection to a Realm

If a publisher loses a connection to its Realm, it will automatically reconnect to the alternative realm and continue publishing there.

Scenario III: Publisher and Subscriber are connected to different Realms

As the above channels on Realm1 and Realm2 are cluster channels, events published to a channels named, say, */sales* on either Realm will be passed to the */sales* channel on the other realm. As long as subscribers are consuming from the */sales* channel on one of the realms they will receive all events. Thus full guaranteed delivery is provided even if the publisher is publishing to Realm1 and subscribers are consuming from Realm2.

For more information on HA configuration options please contact the support team who will be happy to outline the pros and cons of the various HA configurations available.

Multicast: An Overview

Universal Messaging's ability to provide 'ultra-low latency' messaging has been further developed with the introduction of multicast options in addition to unicast to distribute messages to client applications.

With *unicast*, the server must physically write the message once for each destination client:

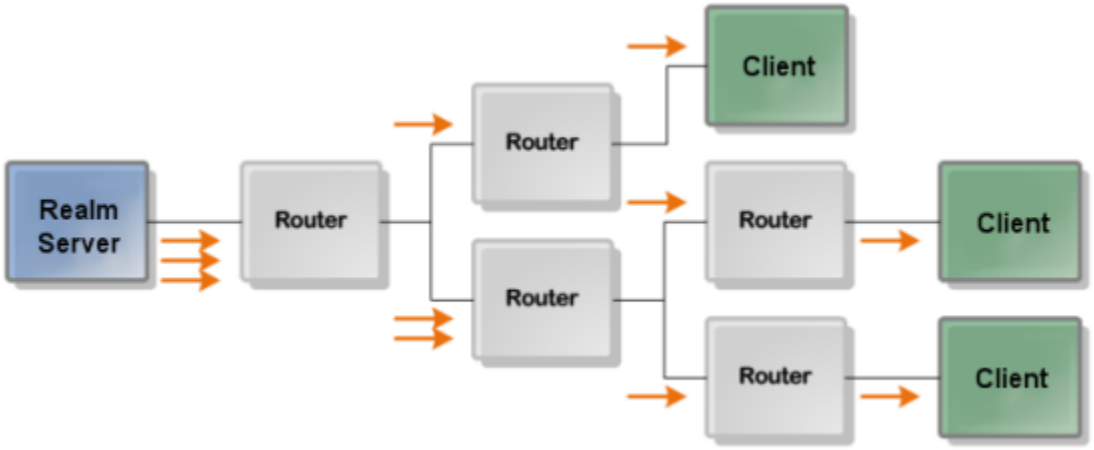


Figure 1: Universal Messaging in Unicast Mode

With *multicast*, a message is written to the network once where it is then routed to all connections in that multicast group:

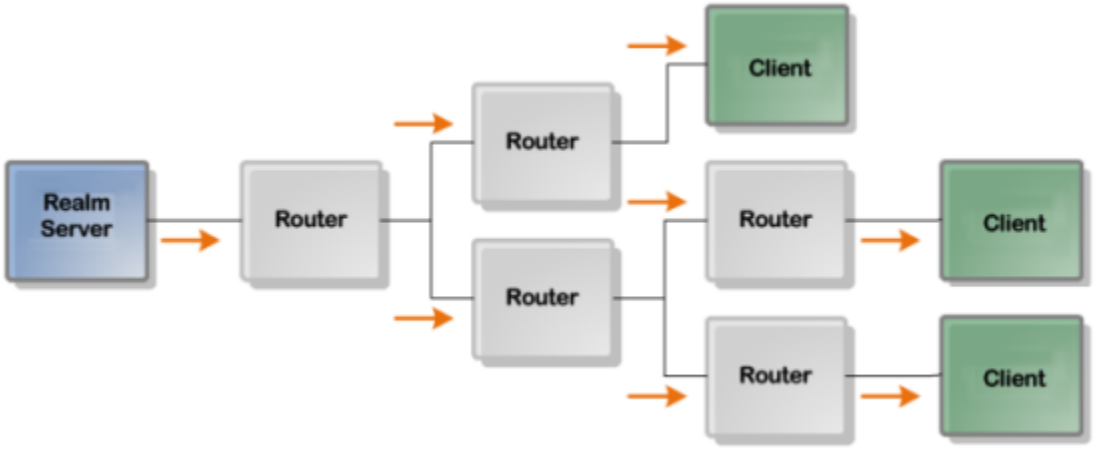


Figure 2: Universal Messaging in Multicast Mode

Multicast has clear performance improvements in terms of scalability. Where the performance of unicast gradually degrades as the number of destinations increases, multicast performance does not as the server still only has to write a message once per destination.

Universal Messaging supports multicast for cluster communication and can be enabled on individual DataGroups. Universal Messaging will then transparently operate in a multicast delivery mode for clients on networks capable of supporting the multicast protocol else continue to use unicast.

When you create a DataGroup you can pass a flag to enable multicast delivery. The Universal Messaging realm will automatically begin delivering events published to that group via multicast. When a client DataStream is added to a multicast enabled Data Group, it will transparently receive the information it needs to begin consuming via multicast. Multicast delivery may not be possible for all clients so initially the client will be sent events via both unicast and multicast. The user will only be delivered the data once however.

The Universal Messaging server will quickly detect whether or not the multicast packets are reaching the client as it does or does not receive acknowledgements. If the client does not support multicast, the server will simply continue to send events to that client via unicast. If the multicast packets do reach the client then after a period of time when both unicast and multicast are in sync, the Universal Messaging Server will stop sending unicast events to that DataStream.

Multicast addresses packets using the User Datagram Protocol (UDP) as opposed to TCP which is used in unicast. UDP is a connectionless protocol and does therefore not guarantee delivery or packet ordering in way that TCP does. However Universal Messaging still provides these guarantees by implementing the required checks at the application layer. These checks happen completely transparently to the user and any packet retransmissions can be monitored using the nAdminAPI or the Enterprise Manager.

Shared Memory (SHM)

Shared Memory (SHM) is our lowest latency communications driver, designed for inter-process communication (IPC). SHM can be used for client and inter-realm (cluster and join) communication. It is operating system independent as it does not depend on any OS specific libraries, making it simple to setup and use.

As the name suggests, shared memory allocates blocks of memory that other processes can access - allowing the same physical computer to make connections without network overhead. This has many advantages, one of which is that when the realm and the data source (publisher) are located on the same physical computer, there is no network latency added between them. This results in less latency for delivery to clients.

Advantages

- Lowest latency
- No network stack involved
- Efficient use of resources with no Network IO required

Disadvantages

- Same physical machine only
- Only currently supported by certain JVMs such as Oracle JDK1.6.0_32, JDK 7 and Azul Zing. On HP-UX systems, shared memory drivers are currently not supported.

Scalability

Performance, Scalability and Resilience

Performance, Scalability and Resilience are design themes that are followed in all areas of Universal Messaging's design and architecture. Specific implementation features have been introduced into server and client components to ensure that these themes remain constantly adhered to.

Performance

Universal Messaging is capable of meeting the toughest of Low Latency and High Throughput demands. Universal Messaging's server design, sophisticated threading models and heavily optimized IO subsystem ensures peak performance. Multicast and shared memory communication modes allow Universal Messaging to consistently achieve low microsecond latencies. Universal Messaging is constantly being benchmarked by clients and other 3rd parties and continues to come out on top.

The benchmarking section provides detailed information on performance in a variety of scenarios. Information on performance tuning is also available on the website, helping clients achieve optimal performance in their deployments.

Scalability

Scalability in terms of messaging Middleware typically means supporting large numbers of concurrent connections, something Universal Messaging does out of the box. However in defining truly global enterprise applications a single system often needs to scale across more than one processing core, often in more than one geographic location.

Universal Messaging servers can be configured in a variety of ways to suit scalability (and resilience) requirements. Multiple Universal Messaging servers can exist in a single federated name space. This means that although specific resources can be put onto specific Universal Messaging realm servers any number of resources can be managed and access centrally from a single entry point into Universal Messaging's federated namespace. In addition to high availability and resilience features Universal Messaging Clusters also offer a convenient way to replicate data and resources among a number of realm servers thus supporting higher rates of concurrent connections.

Resilience

Business contingency and disaster recovery planning demand maximum availability from messaging Middleware systems. Universal Messaging provides a number of server and client features to ensure data always remains readily accessible and that outages are transparent to clients.

Universal Messaging Clusters replicate all resources between realms. Channels, Topics, Queues and the data held within each one is always accessible from any realm server in the cluster.

Universal Messaging clients are given a list of Universal Messaging realms in the cluster and automatically move from one to another if a problem or outage occurs.

Realm Benchmarks

The benchmarks detailed in this section of the website are designed to give indications of the performance levels achievable with Universal Messaging.

These benchmarks and their accompanying results are presented as a guide only. They provide indications of the levels of performance for the given context in which they have been run. Performance in contexts or environments different to one we present are likely to give different results.

The performance of these benchmarks is limited primarily by the available network infrastructure and machine which the Universal Messaging Realm Server resides on. The results were produced by running the benchmark using commercially available machines which may typically be used to host services in an enterprise environment.

If you would like access to the benchmarking tools to run these tests yourselves, please email our support team.

Test Scenarios

Descriptions of several tests are detailed below, with each test running with a different configuration to provide indications of performance in varying types of context. These tests include:

- High Update Rates with Multi-cast delivery
- High Update Rates
- Medium Update Rates
- Low Update Rates

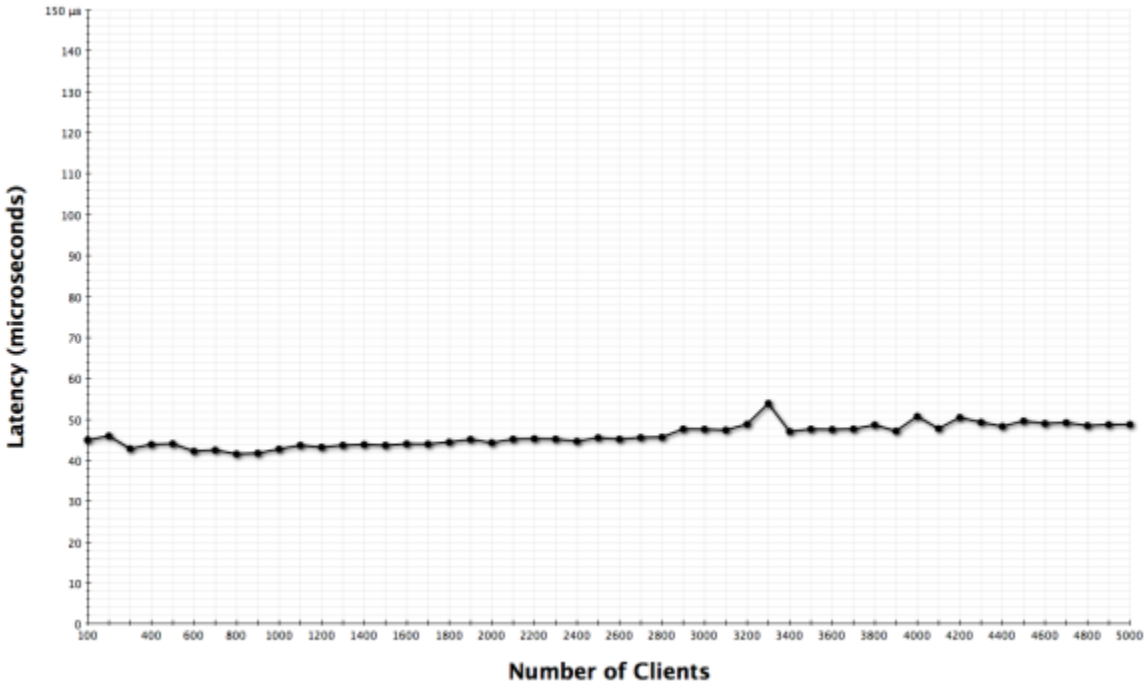
The low and medium rate tests are designed to model traffic which may be typical of non-time critical applications which may instead focus the number of concurrent connections a server can handle.

The high rate tests are designed to model latency critical applications which use small, frequent messages. This kind of traffic is more typical of trading systems which some of our customers deploy.

High Update Rate (Using Multi-Cast Delivery)

Multi-cast delivery is a new feature in Universal Messaging 7. This allows for ultra low latency delivery of streaming data to clients. The characteristics of the test are the same as the high update rate test, leveraging the benefits of multi-cast delivery to cope with the most demanding performance requirements.

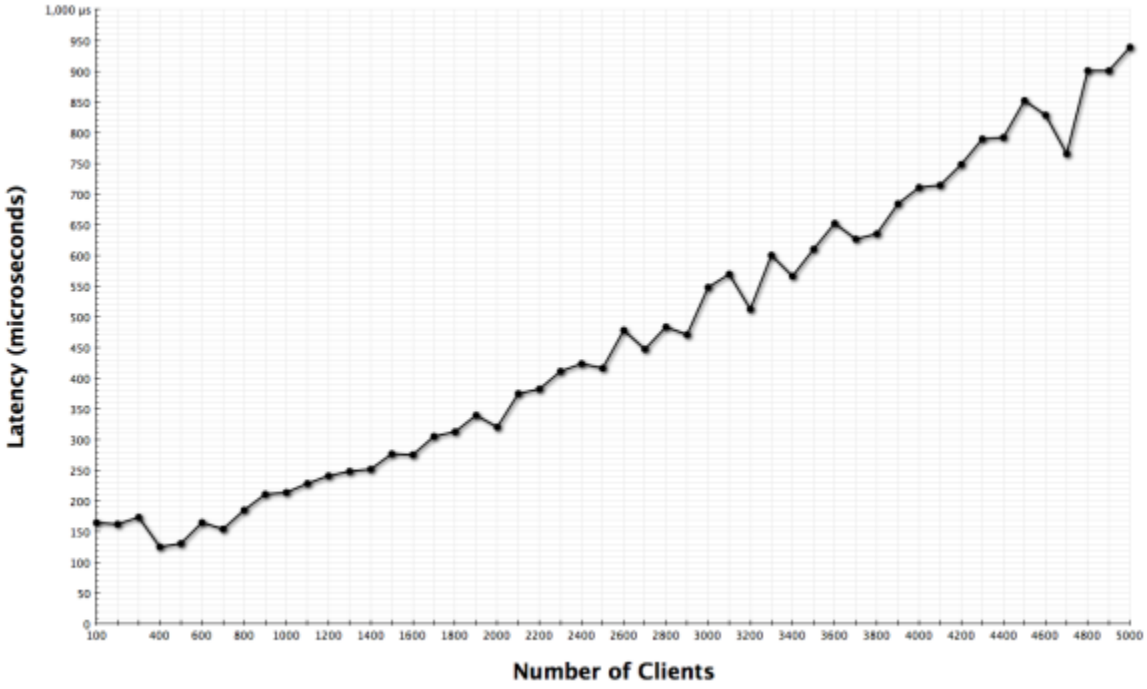
DataGroups	500
Clients	5,000
Increment Rate	100 every 60 seconds
Subscriptions per Client	50
Messages per second per Client	50
Peak Message Rate	250,000



High Update Rate

The high update rate test is designed to model applications which require low latency delivery times for data which is streamed in small intervals to a small set of clients. These characteristics are often found in trading systems.

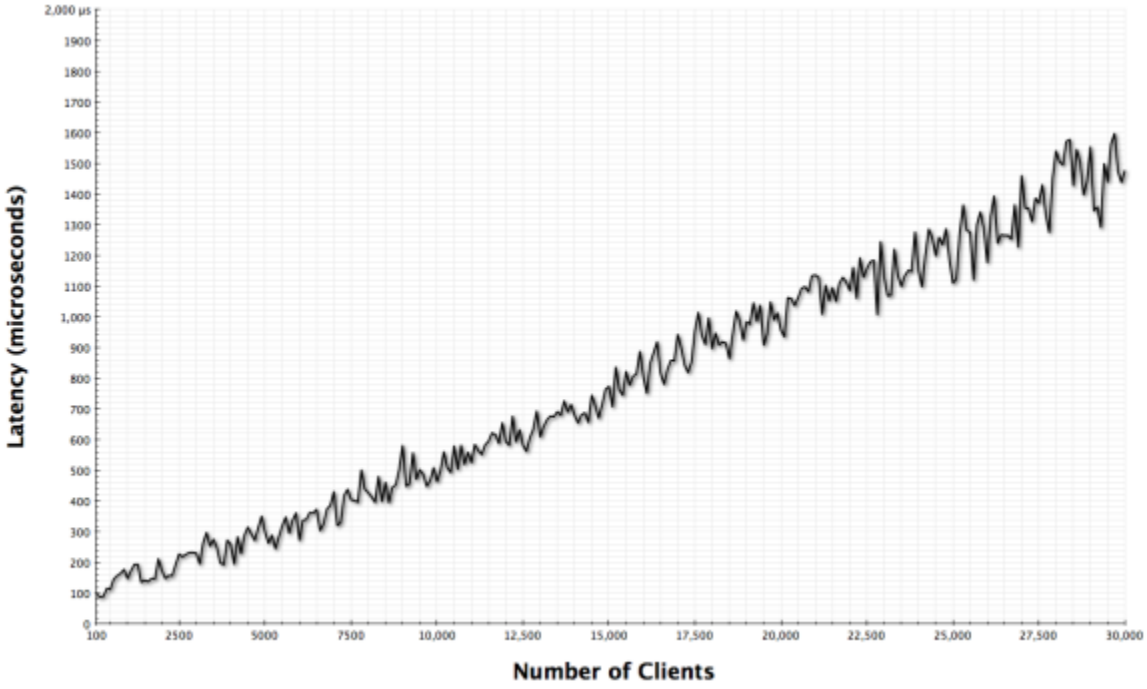
DataGroups	500
Clients	5,000
Increment Rate	100 every 60 seconds
Subscriptions per Client	50
Messages per second per Client	50
Peak Message Rate	250,000



Medium Update Rate

The medium update rate test is designed to model applications which require low latency delivery times to data which is streamed to clients at a moderate rate.

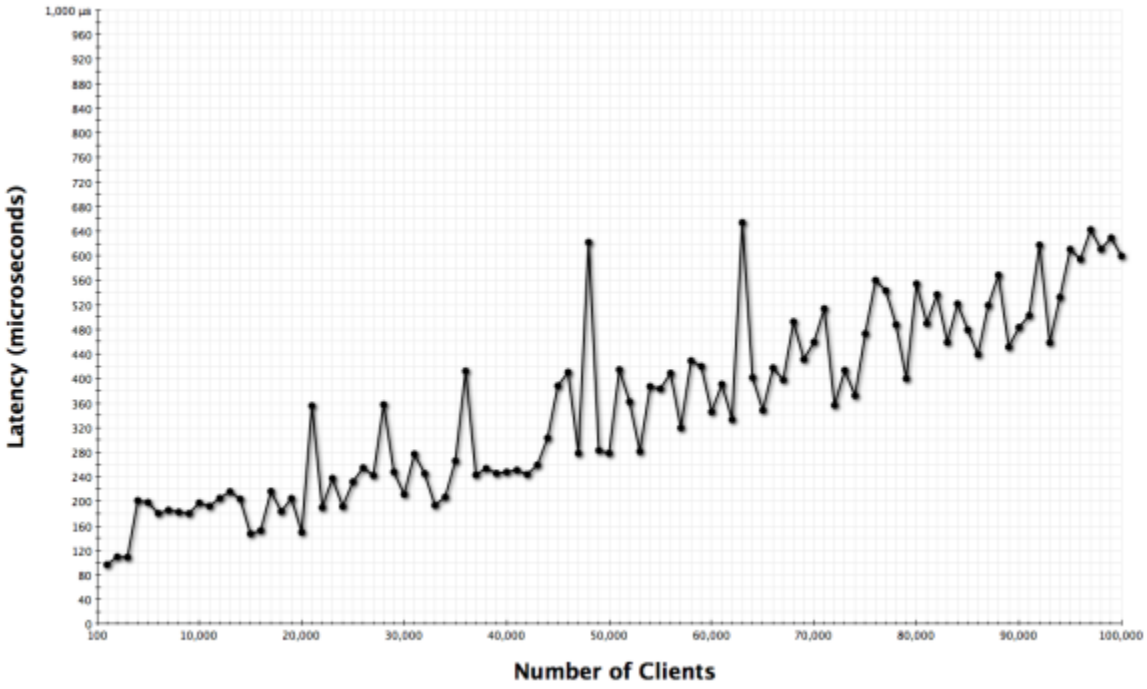
DataGroups	500
Clients	30,000
Increment Rate	100 every 60 seconds
Subscriptions per Client	10
Messages per second per Client	10
Peak Message Rate	300,000



Low Update Rate

The low update rate test is designed to model services which update infrequently with the focus on scaling delivery to large numbers of clients in a reasonable time frame.

DataGroups	500
Clients	100,000
Increment Rate	1,000 every 60 seconds
Subscriptions per Client	1
Messages per second per Client	1
Peak Message Rate	100,000



JavaScript High Update Rate

The JavaScript High Update Rate test uses headless simulation clients which communicate with the server using the WebSocket protocol. This provides an indication of the ability of the Universal Messaging Realm Server to scale to serve large numbers of web clients.

DataGroups	500
Clients	5,000
Increment Rate	100 every 60 seconds
Subscriptions per Client	50
Messages per second per Client	50
Peak Message Rate	250,000

Failover

Universal Messaging clients, whether server to client or server to server, automatically handle disconnect and reconnection to any number of alternative Universal Messaging Realm servers.

If a Universal Messaging server becomes unreachable the Universal Messaging session will automatically try to connect to the next defined Universal Messaging server in

the list of RNAME values provided when the nSession object was created. Universal Messaging clients can get this list of RNAMES from the server during the session handshake or programmatically at any time. The ability to pass RNAME values from server to client dynamically makes it very easy to manage client failover centrally.

Single Universal Messaging Realm

In a single server scenario upon receipt of an abnormal disconnection a Universal Messaging client will automatically try to reconnect. The back-off period and interfaces handling reconnection are all fully exposed in a public API. They can be used to trigger off any number of specific events.

Multiple Universal Messaging Realms

In a multiple server scenario Universal Messaging clustering (see "Clusters: An Overview" on page 37) allows for multiple Realms to act as live replicas of each other. This has huge benefits both in terms of load balancing and high availability.

Universal Messaging is also fully compatible with high availability and business contingency products implemented within the underlying operating system. This means that Universal Messaging is compatible with existing HA/BCP policies as well as providing its own in built fail over functionality.

Connections Scalability

Single server

A single Universal Messaging server has no hard limit set on the number of client connections that can be made to the server. Universal Messaging servers are implemented using Java's NIO (non-blocking IO) for all client communications including SSL. This means that there are no additional threads created and dedicated to each new client connection.

Multiple Servers

Universal Messaging messaging can be achieved across multiple server instances either by the use of a federated name space (see "Federation Of Servers" on page 73) or by using a Universal Messaging Cluster (see "Clusters: An Overview" on page 37).

Universal Messaging servers are aware of each other within a name space. Universal Messaging channels can appear in multiple places in a single name space allowing Universal Messaging to be infinitely scalable. In addition, Universal Messaging's licensing model places no constraints on the number of message servers run. Instead it is based on the number of channels used, thus allowing your client numbers to scale massively without additional cost.

4 Deployment

Deployment

The structure and target audience for any Enterprise Application determines the deployment topology for the underlying infrastructure that supports it. Universal Messaging provides a wide degree of flexibility when faced with stringent deployment requirements. Key features are discussed below

Persistence and Configuration

Universal Messaging uses its own persistent stores that remain relative to its installation location on your file system. Multiple realms can be configured from a single installation, each with their own configuration files and persistent stores for event storage.

Configuration Snapshots

All aspects of a Universal Messaging realms configuration can be stored in an XML file. Channels, ACL's, Interface configuration, Plugins etc. can all be included. New realms can quickly be bootstrapped from existing configuration files making the deployment to new environments quick and simple.

Interfaces

Universal Messaging Realms can support multiple communications interfaces, each one defined by a protocol a port. Universal Messaging Realms can be configured to bind to all network interfaces on a machine or specific ones depending on configuration requirements. Specific SSL certificate chains can be bound to specific interfaces thus insuring clients always authenticate and connect to specific interfaces

Web Deployment

As well as providing a wide range of client web technology support Universal Messaging's realm server provides useful features to aid web deployment. In addition to providing a messaging backbone for external clients Universal Messaging can also act as a web server delivering static and server-generated content to clients. This resolves security sandbox problems and port use.

Forward and reverse proxy server functionality is available for those that wish to host web content on a different server but stream real time data from Universal Messaging.

Server

Performance and Tuning

This section will provide some initial information and guidance on how to get the best out of Universal Messaging, as well as provide an explanation to understand the significance of certain steps when tuning applications for low latency.

General Advice

Much of the information given in this section of the Website is related to tuning a specific element of your system. In the list below are more general pieces of advice which may help improve performance. Links to more in depth tuning articles can be found further down the page.

- Ensure you are running the latest Universal Messaging release. We strive to enhance performance between releases, upgrading will ensure you are able to leverage the newest improvements
- Use the latest version of the Java Virtual Machine. JVM vendors often improve the performance of the virtual machine, or its garbage collector between releases.
- Collect monitoring information which will allow you to make informed decisions based on the origin of performance bottlenecks. Operating System provide statistics on memory consumption, processor and network utilization. Java Virtual Machines can output Garbage Collection statistics which can be a key part of diagnosing why an application may not be performing.

Detailed Topics

The links below provide additional information on tuning important aspects of a deployed application.

- "Realm Configuration" on page 88
- "JVM Tuning" on page 80
- "OS Tuning" on page 86
- "Network Tuning" on page 85

Validating Results

Much of the advice given here is based on our own observation by running our internal benchmarking suite. Your environment and needs may differ from those we model, so we would encourage that you validate any changes you make to your environment.

Many parameters, usually kernel parameters, are specific to an individual machine. Furthermore, it can be dangerous to change them without proper knowledge. It is encouraged to exercise caution when changing such settings.

Server Failover / High Availability

In order to provide your clients with a service that is highly available, clustering is recommended. Clusters enable transparency across your clients. If one server becomes unavailable, the client will automatically reconnect to another realm within the cluster. All cluster objects within the realm are replicated among all cluster realms and their state is maintained exactly the same across all realm members. Therefore whenever a client disconnects from one realm and reconnects to another, they will resume from the same position on the newly connected realm.

When a client provides a list of RNAMES as a comma separated list, if each entry in the list corresponds to realm that is a member of the cluster, then the client will reconnect to the next realm in the cluster list.

For more information on clustering, please see the clustering section in the Administrators Guide.

Data Routing

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

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

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

When creating a join 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 filter to be applied to the join.

Hop Count

Joins have an associated hop-count, which can optionally be defined when the join is created. The hop count allows a limit to be put on the number of subsequent joins an event can pass through if published over this join. If a hopcount is not defined for a join, it will default to 10.

The hop count is the number of *intermediate* stores between the source channel and the final destination. As an example, imagine we have 10 channels named "channel0" to "channel9" and all these channels are joined sequentially. When we publish to channel 0, if the join from channel0 to channel1 has a hopcount of 5 then the event will be found on channel0 (the source channel), channels 1 to 5 (the intermediate channels) and channel6 (the endpoint).

Loop Detection

Joins allow the possibility of defining a loop of joined channels. To prevent channels receiving multiple copies of the same event, Universal Messaging implements loop detection on incoming events. To illustrate this, imagine a simple example with two channels (channel0 and channel1) and we create a loop by joining channel0 to channel1 and channel1 to channel0. If we publish to channel0 the event will also be published to channel1 over the join. But channel1 is joined to channel0 too, so now the event would get published to channel0 again. Without Universal Messaging's loop detection, this cycle would repeat until the maximum hopcount has been reached.

To prevent this, Universal Messaging detects when a message which has already been published to a channel or queue and will not publish it a second time.

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.

Federation Of Servers

Universal Messaging supports the concept of a federated namespace, where realm servers may be located in different geographical locations but form part of the same logical namespace. A Universal Messaging name space can contain one or more Universal Messaging message servers, each one containing many topics, queues or Peer 2 Peer services.

Each Universal Messaging server is aware of others that have been added to the namespace and each one can redirect clients automatically to the required resource thus providing alternative routes when network outages occur. There is no single point of entry to a federated Universal Messaging namespace and it can be traversed in any direction from any point.

The entry into a Universal Messaging name space or server is via a custom URL called an RNAME (see "Communication Protocols and RNAMEs" on page 22). The RNAME provides the protocol, host and port required to access the Universal Messaging server. Universal Messaging clients can be passed an array of RNAME's. Should a connection fail to one of the realms the Universal Messaging client automatically moves onto the next.

The remote management of either clustered or federated realm servers is enabled via the Universal Messaging administration tool or administration API. There is no limit placed on the number of Universal Messaging Realms that can be managed from the Universal Messaging Enterprise Manager or using the Universal Messaging Administration API.

Proxy Servers and Firewalls

Universal Messaging transparently traverses modern proxy servers and firewall technology. Universal Messaging's HTTP and HTTPS drivers (see "Connecting Over HTTP/HTTPS" on page 89) support straight proxy servers as well as user authenticated proxy servers.

Universal Messaging self contained HTTP/HTTPS implementation ensures that if a remote client can access your web site the same client can access a Universal Messaging realm.

Server Memory for Deployment

Universal Messaging servers provide 3 different memory modes. Typically, the Universal Messaging Realm server will be deployed using the large memory mode. When deploying a Universal Messaging server, one of the considerations for memory consumption concerns the volatility of your data, and specifically the types of channels and queues you are using.

The channels that consume the most memory are those channels that keep the events in memory and do not write events to persistent store. These channels are known as *Simple* and *Reliable*.

If you have a simple channel with a TTL of say 1 day (86400000 milliseconds), and you expect to publish a 1k event per second, this channel alone will consume approximately 86.4MB of memory. However if your data has a very short lifespan defined by setting a low TTL, then the memory consumption would be much less than it would be with a 1 day TTL.

This kind of calculation will indicate to you how much maximum memory the Realm Server JVM needs to be allocated to avoid running out of memory.

If you follow these simple guidelines, you should be able to estimate the memory required for your channels.

Server Parameters

Introduction

When a Universal Messaging Realm Server is started, there are a number of parameters used in its initial startup sequence. These parameters are in the form of -D options that form part of the nserver.lax command line.

The Universal Messaging client API also supports a variety of different parameters that can be specified in the command line of any Universal Messaging Client application.

This section describes those -D parameters, what they are used for and their typical values.

Name	Required	Default	Description
javax.net.ssl.keyStore	N		Used to set the default KeyStore the server will use. If not supplied the client MUST set one when configuring an SSL interface
javax.net.ssl.keyStorePassword	N		Used to set the default password for the keystore. If not supplied the client must set one when configuring an SSL interface
javax.net.ssl.trustStore	N		Used to set the default trust store the server will use. If not supplied the client MUST set one when configuring an SSL interface
javax.net.ssl.trustStorePassword	N		Used to set the default Truststore password the server will use. If not supplied the client MUST set one when configuring an SSL interface
LOGLEVEL	N	4	Specifies the current log level to use
LOGFILE	N	System.out	Used to specify a log file to write the log entries to
LOGSIZE	N	100000	Specified in bytes before the log file is rolled
MaxMemory	N	Uses -Ms value	Used to specify the maximum amount of memory to use

Name	Required	Default	Description
SECURITYFILE	N		Used to specify the Super Users for this realm. Format is user@host (one per line). Note that this is only a bootstrap method on startup of a realm. If you had previously started the realm before specifying a SECURITYFILE, you will need to remove the files `realms.nst` and `realms.nst _old` from the `RealmSpecific` directory, then restart the realm with the -DSECURITYFILE setting in the lax file for the super user entries in the file to be added to the realm ACL.
DATADIR	Y		What directory to use to store files within
CKEYSTORE	N		Short hand for javax.net.ssl.keyStore
CKEYSTOREPASSWD	N		Short hand for javax.net.ssl.keyStorePassword
CAKEYSTORE	N		Short hand for javax.net.ssl.trustStore
CAKEYSTOREPASSWD	N		Short hand for javax.net.ssl.trustStorePassword
REALM	Y		Specifies the name of the Realm Server
CHANNELUMASK	N		Specifies the default channel protection mask

Name	Required	Default	Description
ADAPTER	N		Specifies an interface to use, eg nsp://0.0.0.0:9000/
ADAPTER_x	N		Specifies an interface to use, eg nsp://0.0.0.0:9000/ where x = 0 -> 9
mode	N		If set to IPAQ forces a small memory mode for the server
javax.net.ssl.debug	N		Useful to debug SSL issues, see www.javasoft.com for more information

Server Security for Deployment

Universal Messaging provides configurable security for authentication and entitlements. When a user connects using SSL, the server must have an SSL enabled interface configured. Once the interface is configured correctly, clients can connect to a realm using an SSL encrypted session.

Before clients can use the realm correctly, the correct permissions must be granted to each user within the ACLs for the realm, resources (see "Messaging Paradigms supported" on page 12) and services. For more information on this please see the security section.

Deployment

The structure and target audience for any Enterprise Application determines the deployment topology for the underlying infrastructure that supports it. Universal Messaging provides a wide degree of flexibility when faced with stringent deployment requirements. Key features are discussed below

Persistence and Configuration

Universal Messaging uses its own persistent stores that remain relative to its installation location on your file system. Multiple realms can be configured from a single installation, each with their own configuration files and persistent stores for event storage.

Configuration Snapshots

All aspects of a Universal Messaging realms configuration can be stored in an XML file. Channels, ACL's, Interface configuration, Plugins etc. can all be included. New realms can quickly be bootstrapped from existing configuration files making the deployment to new environments quick and simple.

Interfaces

Universal Messaging Realms can support multiple communications interfaces, each one defined by a protocol a port. Universal Messaging Realms can be configured to bind to all network interfaces on a machine or specific ones depending on configuration requirements. Specific SSL certificate chains can be bound to specific interfaces thus insuring clients always authenticate and connect to specific interfaces

Web Deployment

As well as providing a wide range of client web technology support Universal Messaging's realm server provides useful features to aid web deployment. In addition to providing a messaging backbone for external clients Universal Messaging can also act as a web server delivering static and server-generated content to clients. This resolves security sandbox problems and port use.

Forward and reverse proxy server functionality is available for those that wish to host web content on a different server but stream real time data from Universal Messaging.

Connecting to multiple realms using SSL

This Section describes how to connect to multiple Universal Messaging realms using SSL when different certificate hierarchies are used on each respective realm. The information below applies to any of the various wire protocols (see "Communication Protocols and RNAMEs" on page 22) that Universal Messaging supports, such as SSL enabled sockets (nsps) and HTTPS (nhps). Please note that the example programs contained in the Universal Messaging package will all work with SSL enabled on the realm server.

The certificate requirements differ depending on whether the realms require client certificate authentication or not. Let us assume that we want to connect to 2 realms over nsps, realmA and realmB. RealmA has interface nsps0 which uses a certificate signed by CA1, while RealmB has interface nsps0 which uses a certificate signed by CA2. . The next few paragraphs describe what needs to be done for each possible configuration.

Common requirements

The Universal Messaging client API uses JSSE so only 1 keystore file/keystore password and 1 truststore file/truststore password can be used. In order to achieve our goal then we will have to create a combined keystore and / or a combined truststore depending on our configuration.

Client certificate authentication NOT required

In the case where client certificate authentication is not required by both realms, your application needs to use a combined truststore / truststore password only using the -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters.

1. Both CA1 and CA2 are well known Root Certificate Authorities

   All well known Root CAs are already included in the JRE cacerts file which can be found in jre\lib\security\cacerts . Unless you have manually changed that keystore's password the default password is changeit. You have to use these values for your -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters.

2. CA1 is a well known Root Certificate Authority but CA2 is not (or vice versa)

   Two choices are available for this configuration. Either you add CA2's certificate to the JRE cacerts file or you create a combined keystore with CA2's certificate and CA1's certificate.You have to use these values for your -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters.

3. CA1 and CA2 are not well known Root Certificate Authorities

   In this instance you have to create a combined truststore file that contains both the CA1 and CA2 certificates. In order to do this export your CA certificates from their current JKS store files then create a new JKS file and import them. You can do this using the JDK keytool command line utility. Finally you have to use these values for your -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters.

Client certificate authentication required

In the case where client certificate authentication is not required by both realms, your application needs to use a combined keystore / keystore password and a combined truststore / truststore password using the -DCKEYSTORE, -DCKEYSTOREPASSWD, -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters respectively.

1. Both CA1 and CA2 are well known Root Certificate Authorities

   With regards to the truststore, all well known Root CAs are already included in the JRE cacerts file which can be found in jre\lib\security\cacerts . Unless you have manually changed that keystore's password the default password is changeit. You have to use these values for your -DCAKEYSTORE and -DCAKEYSTOREPASSWD parameters.

   With regards to the keystore, you need to create a combined keystore that contains both client certificates and then point the -DCKEYSTORE parameter to its path as well as set the -DCKEYSTOREPASSWD to the password of that combined keystore. In order to create a combined keystore, export the certificates and private keys in PKCS#12 format and then import them as trusted certificates in the same keystore file. You can do this using the JDK keytool command line utility.

2. CA1 is a well known Root Certificate Authority but CA2 is not (or vice versa)

   The easiest way for this configuration option is to create a single JKS file that contains the CA1 certificate, the CA1 signed client certificate, the CA2 certificate and

the CA2 client certificate. You then have to use the same values for CKEYSTORE, CAKEYSTORE and CKEYSTOREPASSWD, CAKEYSTOREPASSWD respectively.

3. CA1 and CA2 are not well known Root Certificate Authorities

   Again the easiest way for this configuration option is to create a single JKS file that contains the CA1 certificate, the CA1 signed client certificate, the CA2 certificate and the CA2 client certificate. You then have to use the same values for CKEYSTORE, CAKEYSTORE and CKEYSTOREPASSWD, CAKEYSTOREPASSWD respectively.

Environment Settings

The CKEYSTORE, CKEYSTOREPASSWD, CAKEYSTORE and CAKEYSTOREPASSWD system properties are used by the Universal Messaging sample apps, but are mapped to system properties required by a jsse enabled JVM by the utility program 'com.pcbsys.foundation.utils.fEnvironment', which all sample applications use. If you do not wish to use this program to perform the mapping between Universal Messaging system properties and those required by the JVM, you can specify the SSL properties directly. To do this in your own applications, the following system properties must be set:

```
-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, both the Universal Messaging sample apps and your own 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 (see "Communication Protocols and RNAMEs" on page 22) used by your client application must correspond to the correct type of SSL interface, and the correct hostname and port that was configured earlier.

The Java Virtual Machine

Selecting and Tuning a JVM is an important part in running any Java application smoothly. Applications with low latency requirements often require more attention paid to the JVM, as the JVM is often a big factor in performance.

This section outlines JVM selection, and advice on tuning for low latency applications on these JVMs. There are many different JVM vendors available and each JVM has

slightly different configurable parameters. This section outlines a few key vendors and important configuration parameters.

Selecting a Java Virtual Machine

As mentioned above, there are a variety of JVMs to choose from that come from different vendors. Some of these are free, some require a license to use. This section will outline the standard Oracle HotSpot VM, the Oracle JRockit VM and the Azul Zing VM.

Oracle HotSpot VM

The Oracle HotSpot VM is offered by Oracle, the JDK and JRE freely available from the Oracle Website. This JVM is suitable to fulfil most users needs for Universal Messaging.

Oracle JRockit VM

The Oracle JRockit VM is another offering from Oracle. This JVM was made free and publicly available in May 2011. It contains many of the same assets from the Oracle HotSpot VM. The performance capabilities of this VM are often advertised as greater than that of the HotSpot VM, however this varies depending on the usage scenario.

Oracle plan to eventually merge the code for the HotSpot and JRockit VMs in the future. This is not expected to happen until JDK8 at the earliest however.

Azul Zing VM

The Azul Zing VM is a commercial offering from Azul. Its primary feature is a 'Pauseless Garbage Collection'. This VM is well suited to applications which require the absolute lowest latency requirements. Applications which experience higher garbage collection pause times may also benefit from using this VM.

Configuring the Java Virtual Machine - Oracle HotSpot

This section covers parameters for the Oracle HotSpot VM which may help improve application performance. These settings can be applied to a Universal Messaging Realm Server by editing the nserver.lax file found under the server/realm/bin directory of your installation.

General Tuning Parameters

Below are some suggestions of general tuning parameters which can be applied to the HotSpot VM.

-Xmx	The maximum heap size of the JVM.
-Xms	The minimum heap size of the JVM. Set this as equal to the maximum heap size
-XX:+UseLargePages	Allows the JVM to use large pages. This may improve memory access performance. The system must be configured to use large pages.

-XX:+UseNUMA	Allows the JVM to use non uniform memory access. This may improve memory performance

Monitoring Garbage Collection Pause Times

It is important to collect proper monitoring information when tuning an application. This will allow you to quantify the results of changes made to the environment. Monitoring information about the Garbage collection can be collected from a JVM without any significant performance penalty.

We recommend using the most verbose monitoring settings. These can be activated by adding the following commands to the nserver.lax file.

```
-verbose:gc
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintGCApplicationStoppedTime
```

This will produce output similar to the following:

```
2012-07-06T11:42:37.439+0100:
        [GC
            [ParNew: 17024K->1416K(19136K), 0.0090341 secs]
                17024K->1416K(260032K), 0.0090968 secs]
                [Times: user=0.02 sys=0.01, real=0.01 secs]
```

The line starts by printing the time of the garbage collection. If Date Stamps are enabled, this will be the absolute time, otherwise it will be the uptime of the process. Printing the full date is useful for correlating information taken from the nirvana logs or other application logs.

The next line shows if this is a full collection. If the log prints GC, then this is a young generation collection. Full garbage collections are denoted by the output *Full GC (System)*. Full garbage collections are often orders of magnitude longer than young garbage collections, hence for low latency systems they should be avoided. Applications which produce lots of full garbage collections may need to undergo analysis to reduce the stress placed on the JVMs memory management.

The next line displays the garbage collectors type. In this example *ParNew* is the Parallel Scavenge collector. Detailed explanation of garbage collectors are provided elsewhere on this page. Next to the type, it displays the amount of memory this collector reclaimed, as well as the amount of time it took to do so. Young garbage collections will only produce one line like this, full garbage collections will produce one line for the young generation collection and another for the old generation collection.

The last line in this example shows the total garbage collection time in milliseconds. The user time is the total amount of processor time taken by the garbage collector in user mode. The system time is the total amount of processor time taken by the garbage collector running in privileged mode. The real time is the wall clock time that the garbage collection has taken, in single core systems this will be the user + system time. In multiprocessor systems this time is often less as the garbage collector utilizes multiple cores.

The last flag will also cause the explicit application pause time to be printed out to the console. This output will usually look like the following:

```
Total time for which application threads were stopped: 0.0001163 seconds
```

If you observe high client latencies as well as long application pause times, it is likely that the garbage collection mechanism is having an adverse affect on the performance of your application.

Tuning Garbage Collection

The Garbage Collector can be one of the most important aspects of Java Virtual Machine tuning. Large pause times have the capability to negatively impact an applications performance by a noticeable degree. Below are some suggestions of ways to combat specific problems observed by monitoring garbage collection pause times.

Frequent Full Garbage Collections

Full garbage collections are expensive, and often take an order of magnitude longer than a young generation garbage collection to complete. This kind of collection occurs when the old generation is full, and the JVM attempts to promote objects from the younger generation to the older generation. There are two scenarios where this can happen on a regular basis:

1. There are many live objects on the heap, which are unable to be cleaned up by the JVM.
2. The allocation rate of objects with medium-long lifespans is exceptionally high

If the information from garbage collection monitoring shows that full garbage collections are removing very few objects from the old generation, and that the old generation remains nearly full after a old generation collection, it is the case that there are many objects on the heap that cannot be cleaned up.

In the case of a Universal Messaging Realm Server exhibiting this symptom, it would be prudent to do an audit of data stored on the server. Stored Events, ACL Entries, DataGroups, Channels and Queues all contribute to the memory footprint of Universal Messaging. Reducing this footprint by removing unused or unnecessary objects will reduce the frequency of full collections.

If the information from garbage collection monitoring shows that young garbage collection results in many promotions on a consistent basis, then the JVM is likely to have to perform full garbage collections frequently to free space for further promotions.

This kind of heap behaviour is caused by objects which remain live for more than a short amount of time. After this short amount of time they are promoted from the young generation into the old generation. These objects pollute the old generation, increasing the frequency of old generation collections. As promotion is an expensive operation, this behaviour often also causes longer young generation pause times.

Universal Messaging will mitigate this kind of problem by employing a caching mechanism on objects. To further decrease the amount of objects with this lifespan it is important that the administrator perform an audit of creation of resources, such as events, acl entries, channels, datagroups or queues. Heavy dynamic creation and

removal of ACL Entries, Channels, DataGroups and Queues may induce this kind of behaviour.

If an administrator has done everything possible to reduce the static application memory footprint, as well as the allocation rate of objects in the realm server then changing some JVM settings may help achieve better results.

Increasing the maximum heap size will reduce the frequency of garbage collections. In general however larger heap sizes will increase the average pause time for garbage collections. Therefore it is important that pause times are measured to ensure they stay within an acceptable limit.

Long Young Generation Collection Pause Times

As mentioned above the primary cause of long young generation pauses is large amounts of object promotion. These objects often take the form of events, ACL entries, channels, datagroups and queues being created.

To minimise the amount of object creation during normal operating hours it is suggested to employ static creation of many channels, datagroups and queues at start up time. This will result in these objects being promoted once at the beginning of operation, remaining in the old generation. Analysing where possible events can be given short lifespans (possibly even made transient) will also reduce the amount of promotion, as these objects will become dereferenced before they are eligible to be moved to the old generation.

It is important to remember that the Java Virtual Machine's memory subsystem performs best when long living objects are created in the initialisation stage, while objects created afterwards die young. Therefore designing your system to create long lived objects like channels at startup and objects like events to be short lived allows Universal Messaging to harmoniously work with the underlying JVM.

Long Full Collection Pause Times

Full Garbage collections which take long periods of time can often be remedied by proper tuning of the underlying JVM. The two recommended approaches to reducing the amount of time spent in full garbage collections is detailed below.

The first approach would be to reduce the overall heap size of the application. Larger heaps often increase the amount of time for a garbage collection cycle to finish. Reducing the heap will lower the average time that a garbage collection cycle takes to complete. Smaller heap sizes will require garbage collecting more often however, so it is important to ensure that you balance the need for lower collection times with collection frequency.

If you are not able to reduce the heap size any further, because garbage collection frequency is increasing, it may be beneficial to change the type of garbage collector used. If you are experiencing high maximum latencies correlated with long GC times it may be beneficial to switch to using the CMS collector.

The Concurrent Mark Sweep (CMS) collector aims to minimize the amount of time an application is paused by doing many of its operations in parallel with the application. This collector can be enabled by adding the following parameter to nserver.lax

```
-XX:+UseConcMarkSweepGC
```

CMS Collections will usually take more time overall than those done with the Parallel Collector. Only a small fraction of the work done by the CMS collector requires the application to pause however, which will generally result in improved response times.

The Network

This page details important network settings that may improve network performance when using Universal Messaging. Many of the commands detailed on this page are specific to Red Hat Linux, though many of the concepts apply globally to all operating systems.

Stop the iptables Service

The iptables service is used to control packet filtering and NAT. In many cases it is not necessary to run this service and a minor performance gain can be seen by disabling this service. To disable this service use the following command:

```
service iptables stop
service ip6tables stop
```

Disable Adaptive Interrupts on Network Interfaces

Interrupts on a network interface notify the system that some network task is required to be run, for example reading some data from the network. Adaptive Interrupts control the rate at which interrupts are generated for these tasks. The delay in processing subsequent interrupts from interrupt coalescing may degrade performance.

```
ethtool -C eth0 adaptive-rx off
```

Disabling adaptive interrupts on an interface will make that interface use the set interrupt rate. This rate will delay interrupts by a set number of microseconds. The minimum value that this delay can be is 0 (immediate). To set this value on an interface use the command:

```
ethttool -C eth0 rx-usecs-irq 0
```

Kernel Settings

The Kernel has many network settings for TCP which can provide performance improvements when correctly tweaked. This section will outline a few suggestions, however care should be taken when changing these parameters. It is also important to validate results as your mileage may vary. These settings should be added to the sysctl.conf file.

Increase Socket Memory

The settings below will increase the amount of memory allocated by the kernel to tcp sockets.

Important: It is important to set these limits to a reasonable level for the amount of memory available on your machine.

```
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
```

```
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
net.ipv4.tcp_mem = 50576   64768   98152
```

Increase Backlog Queue Size

The command below will increase the queue size in packets waiting to be processed by the kernel. This queue fills up when the interface receives packets faster than the kernel can process them.

```
net.core.netdev_max_backlog = 2500
```

Increase the local Port Range

Applications which have to manage large numbers of current connections may find that they will run out of ports under the default settings. This default can be increased by using the following command:

```
net.ipv4.ip_local_port_range = 1024 65535
```

The maximum number of allocated ports are 65535, 1024 of these are reserved. Applications which manage extremely high numbers of connections will require more ports than this. One way to get around these limits would be to create multiple virtual network interfaces.

The Operating System

This page details important operating system and kernel settings that should be considered when optimising the server. The focus of this page is geared towards Red Hat Linux. Many of the suggestions here have synonymous commands under Solaris or Windows, which can be applied to have a similar effect.

Configuring User Limits

Unix has a configurable limit on the number of processes, file descriptors and threads available per user. This functionality is aimed to prevent a user from consuming all of the resources on a machine. These limits are often set to a reasonably low level, as a general purpose user will not consume many of these objects at any one time.

Application Servers like Universal Messaging may, if under considerable load, wish to consume a large number of these resources. Each open connection to a client for example consumes a file descriptor, and application servers which can support tens of thousands of concurrent connections will thus require as many file descriptors. It is therefore important to increase these limits for Universal Messaging.

Temporarily Increasing limits using the ulimit command

`ulimit` is a unix command which can be used to alter user limits. To increase the user limits which Universal Messaging consumes the following commands are recommended:

```
ulimit -n 250000
ulimit -u 10000
```

This will increase the number of file descriptors and the number of user processes allowed. Any processes spawned from the terminal this was entered on will inherit these limits.

Permanently Increasing User Limits

It is also possible to permanently increase the user limits by editing the relevant configuration file. This configuration file can usually be found in /etc/security/limits.conf.

```
user   soft   nofile   250000
user   hard   nofile   250000
user   soft   nproc    10000
user   hard   nproc    10000
```

Disabling Processor Power Saving States

Many new processors have mechanisms which allow them to dynamically turn individual cores on and off to save power. These mechanisms may sometimes degrade processor and memory performance. In applications that require consistent low latency performance it is recommended to disable this feature.

Many processors manage this by using the `cpuspeed` service. This service can be disabled, which on many machines and architectures will turn this functionality off.

```
service cpuspeed stop
```

Some processors however will require further work to disable power saving states. Whether or not your processor will require extra configuration and what those configuration steps are vary from processor to processor. Many Intel processors for example may require the following command to be appended to the boot options of your operating system

```
intel_idle.max_cstate=0
```

As mentioned above however, this will not be necessary for all processors. Consult with your processor specific documentation for information on disabling power saving states.

Stop the Interrupt Request Balance Service

Interrupts are signals generated, generally by devices, to notify a CPU that there is processing which needs to be done. Interrupt Request (IRQ) Balancing is the act of dividing these processes up between cores on a CPU. In some situations this may harm performance of applications running on the CPU, as these interrupts consume processor cycles and loads information into memory.

Disabling IRQ balancing will assign all interrupts to a single core by default. It is possible to assign interrupts to certain cores, but that is beyond the scope of this section. To disable IRQ balance, use the following command.

```
service irqbalance stop
```

The Realm Server

This page will detail important configuration options that can be changed on the server to improve performance. Important monitoring information that can be collected using the Admin API is also mentioned here. This monitoring information can be used to diagnose common problems.

Lowering the Log Level

Logging information can be useful for debugging a variety of problems that may occur. However particularly high logging levels can negatively effect the performance of your application. Logging creates extra objects which increases memory usage, as well as promotes contention between threads which wish to print to the logger.

The log level can be lowered by using the enterprise manager. The highest log level is 0 and produces the most output. The lowest log level is 7, which produces very little output. Unless attempting to gather logging information related to a particular issue it is recommended to keep the log level no lower than 4 (Quiet). Particularly demanding applications may wish to increase this up to 7 if necessary.

Increasing the Peak Watermark

The server is configured to enter a peak operating mode when a certain number of messages are being delivered through the server per second. Peak mode will batch messages in an effort to keep server load at an optimal level. This batching may increase average latencies for clients.

It is possible to raise the peak mode threshold so that the server does not utilise peak mode until a much higher load is reached. It is important to stress that beyond a certain point the non batching performance will suffer as machine limitations are reached.

Machines with good hardware will benefit from having this threshold raised, but slower machines may function better in batching mode after a certain message rate is reached.

Enable Low Latency Fanout Mechanism

By default, the most aggressive fanout mechanism Universal Messaging provides is disabled. This particular mechanism is capable of meeting extremely demanding latency and message rate requirements, however is very demanding on the resources of a system. It is disabled to prevent it consuming resources on machines with less resources (for example development machines).

If the hardware which the Universal Messaging server runs has greater than 8 cores it is recommended that you enable this fanout mechanism to produce the best latencies. This fanout mechanism will consume multiple cores entirely, so will therefore increase the load average of the machine. It is important that you have sufficient free cores free, as otherwise it is possible that this mode will cause Universal Messaging to starve other threads/processes running on the system.

The mechanism can be enabled by adding the following flag to the nserver.lax file under the server installation directory:

```
-DCORE_SPIN=true
```

There are further flags that can be applied to the lax file to customize the behaviour of this fanout setting. The first of these flags can be used to adjust the number of times that this fanout mode will spin on a core attempting to do work before switching to a less aggressive fanout mechanism.

```
-DSPIN_COUNT=1000000000
```

The default value for this spin count is one billion. Reducing this value will generally encourage the server to switch to a CPU intensive fanout mechanism, if the server is not busy. Reducing this value may result in a performance penalty which occurs as a result of using the less intensive fanout mechanism. The maximum value of this parameter is the same as Long.MAX_VALUE.

The less aggressive fanout mechanism employs a wait as opposed to spinning mechanism, the second flag can be used to specify the wait time between checking if work is available.

```
-DSPIN_WAIT=1
```

This parameter will alter the number of nanoseconds which the fanout mechanism will wait for between checking if it has tasks to complete. Increasing this number will decrease the CPU consumption of Universal Messaging, but at a cost to latency.

Client

Connecting Over HTTP/HTTPS

The Universal Messaging messaging APIs provides a rich set of functionality that can be used over sockets, SSL, HTTP and HTTPS. The code used to connect to the Universal Messaging server is the same regardless of which network protocol you are using to connect.

Under the Universal Messaging programming model there are a number of logical steps that need to be followed in order to establish a connection to a Universal Messaging sever (Realm). These involve establishing a session, obtaining a reference to a channel or a transaction, or registering an object as a subscriber.

Universal Messaging fully supports HTTP and HTTPS. Rather than tunnel an existing protocol through HTTP Universal Messaging has a pluggable set of communications drivers supporting TCP/IP Sockets, SSL enabled TCP/IP sockets, HTTP and HTTPS. Both the client and server make use of these pluggable drivers. From the server perspective different driver types can be assigned to specific Universal Messaging interfaces. From a client perspective a Universal Messaging session can be built on any one of the available drivers dynamically.

Please note that before making an HTTP/HTTPS connection to a Universal Messaging realm server you will first need to add a HTTP/HTTPS interface to the realm. See the Enterprise manager documentation for details.

After initialising your Universal Messaging session, you will be connected to the Universal Messaging Realm using HTTPS. 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.

For detailed information including code samples for connecting to Universal Messaging over HTTP/HTTPS please see our developer guides for the language you require.

Browser / Applet Deployment

Introduction

Universal Messaging client applications can run within stand alone applications as well as within Java applets loaded via a web browser such as Chrome, Firefox, MS Internet Explorer, Netscape, etc.

The Universal Messaging client APIs can be used with most Java Plugin versions.

Applet Sandbox / Host Machine Limitation

Applets run within a client's browser, and are subject to strict security limitations as defined by the Applet Model. These limitations need to be considered when deploying applets. One such limitation is that the applet is only allowed to communicate with the host machine from which the applet was downloaded. This restricts the applet to only being permitted to make connections to the applet host machine. This has a number of implications for an applet that uses Universal Messaging's APIs.

Universal Messaging's APIs communicate with a Realm Server (or potentially multiple servers in a cluster). This limitation means that the applet source host must be the same hostname as each Universal Messaging Realm in use by the applet. If the applet is served from a web server, such as Apache, and it is assumed the communication protocol required for Universal Messaging communication is nhp/nhps (http/https). The usual ports used by web servers running http and https are 80 and 443 respectively. Since the web server uses these ports and the realm servers need to run on the same machine with these ports there is obviously a problem since these ports are in use.

However, Universal Messaging provides 2 different methods for ensuring this is not a problem. The first is Universal Messaging's ability to act as a web server through its file plugin. Running a file plugin on an nhp or nhps interface enables the realm server to deliver the applet to the client browser, this removing the need for the web server and of course freeing up the ports for use by the realm server interfaces.

The second method can be used when the web server is apache. We can provide an apache module that acts similarly to mod.proxy for apache. This apache module called mod.Universal Messaging allows the web server to proxy all requests for a specific URL

to another host. This host can be the realm server running on any other port on the same machine or any other machine, and hence once again fixes this issue.

Another way to circumvent this restriction is to digitally sign the applet and thus allowing the applet to communicate with any host.

Browser Plugins

Universal Messaging can either run within a 4.0 browsers own Java Virtual Machine or run within a Java Virtual machine started using the Java plugin.

Universal Messaging does not require installation of the Java Plugin.

Client Jars

Depending on the functionality used by your Universal Messaging application, different jar files are required. This following table illustrates the deployment dependencies between the jar libraries installed by the Universal Messaging installer.

JAR File	Description	Dependency
nClient.jar	Provides Universal Messaging Client functionality (Pub/Sub & Queues)	None
nP2P.jar	Provides Universal Messaging Peer-to-Peer functionality	nClient.jar
nJMS.jar	Provides Universal Messaging Provider to support JMS functionality	nClient.jar
nJ2EE.jar	Provides Universal Messaging support for interacting with Application servers that support J2EE	nClient.jar & nJMS.jar
nAdminAPI.jar	Provides Universal Messaging Administration & Monitoring functionality	nClient.jar, nP2P.jar

JAR File	Description	Dependency
nAdminXMLAPI.jar	Provides Universal Messaging Configuration XML Import / Export functionality	nClient.jar, nP2P.jar, nAdminAPI.jar
nEnterpriseManager.jar	Contains the Enterprise Manager tool	nClient.jar, nP2P.jar, nAdminAPI.jar, nAdminXMLAPI.jar (Optional)
nServer.jar	Contains the Universal Messaging Realm Server	None
nPlugin.jar	Contains the Universal Messaging Server plugins	nServer.jar

Client Security

Universal Messaging makes use of JSSE for SSL enabled communication. Clients are able to connect using standard sockets (nsp), http (nhp), SSL enabled sockets (nsps) or https (nhps). Universal Messaging's client SSL communication uses the JVM's own SSL implementation.

Clients connecting using SSL (see "Client SSL Configuration" on page 138) will connect to a realm server that has an SSL enabled interface with either client authentication on or off.

Once authenticated using SSL, the client must have the desired permissions on the realm and its objects in order to perform the operations required. The entitlements are defined within the ACLs for the realm, channels, queues and services. The ACLs must contain the correct level of permissions for clients connecting to the realm.

Please also see the description of managing realm security ACLs in the documentation of the Enterprise Manager.

Client Parameters

The Universal Messaging client API supports a variety of different parameters that can be specified in the command line of any Universal Messaging Client application.

This section describes those -D parameters, what they are used for and their typical values.

Name	Required	Default	Description
LOGLEVEL	N	4	Specifies the current log level to use
LOGFILE	N	System.out	Used to specify a log file to write the log entries to
LOGSIZE	N	100000	Specified in bytes before the log file is rolled
user.name	N	Signed on name	Used to override the current username without coding it.
HPROXY	N		Used as a short hand to set http.proxyHost and http.proxyPort
http.proxySet	N		Set to true if the url handler is to use a proxy
http.proxyHost	N		Sets the Proxy host name to use
http.proxyPort	N		The proxy port to use
CKEYSTORE	N		Short hand for javax.net.ssl.keyStore
CKEYSTOREPASSWD	N		Short hand for javax.net.ssl.keyStorePassword
CAKEYSTORE	N		Short hand for javax.net.ssl.trustStore
CAKEYSTOREPASSWD	N		Short hand for javax.net.ssl.trustStorePassword
javax.net.ssl.debug	N		Useful to debug SSL issues, see www.javasoft.com for more information

Multiplexing Sessions

Universal Messaging supports the multiplexing of sessions to a specific host in Java, Flex and C#. This allows the circumvention of connection limit issues by packing multiple Universal Messaging sessions into one connection, and can be used to allow the same client to set up multiple subscriptions to a given channel or queue if required.

Multiplexing Sessions

To multiplex two sessions, first construct one session, and then create a new session by multiplexing the original session. These two sessions will now appear to act as normal sessions, but will, in fact, share a single connection.

This can be accomplished either by using the nSession object associated with the original session, or by using the nSessionAttributes used to create this original session. Below are examples of how to multiplex sessions via both methods:

```
//Construct first session
nsa = new nSessionAttributes(realmDetails, 2);
mySession = nSessionFactory.create(nsa, this);
mySession.init();
//Construct second session by multiplexing the first session.
otherSession = nSessionFactory.createMultiplexed(mySession);
otherSession.init();
//Construct a third session by multiplexing the first session's nSessionAttributes.
thirdSession = nSessionFactory.createMultiplexed(nsa);
thirdSession.init();
```

Examples of multiplexing a session are available for Java and C# .NET. The Tradespace demo application also provides an example of how to implement multiplexed sessions in Adobe Flex.

Language Deployment Tips

Adobe Flex Application Deployment

Adobe Flex and the Universal Messaging Policy File Server

Universal Messaging has a built in socket policy server for Flex applications that require tcp socket or ssl socket (NSP or NSPS) protocol.

The Universal Messaging Flex API also supports tcp sockets (nsp). In the same way that the Silverlight plugin will make a client access policy file request, so too will the Adobe Flash Player make a cross domain request when a socket connection is attempted by the Universal Messaging Flex API. This means that when a Flex application is loaded and tries to make a socket request, Flash will first of all attempt to make a cross domain file request on port 843 to the host the original socket request was being made to. For example, if i specify an RNAME of nsp://myhost.mydomain.com:9000, when the Universal Messaging Flex API attempts to construct a socket on port 9000, it will first

of all make a socket connect request to myhost.mydomain.com:843, and issue a cross domain file request.

So to enable successful deployment of socket based Flex Applications you need to run a policy file server on a socket interface that will automatically handle cross domain file requests. You would first need to create a socket interface on port 843, and select the "Enable Policy Server" check box under the Basic tab for an nsp interface. Once this is setup, you will also need a crossdomain.xml file in the /install/server/name/plugins/htdocs directory of the server. An example crossdomain.xml file might contain something like the following:

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all"/>
        <allow-access-from domain="*" to-ports="80,443,9000" secure="false"/>
        <allow-http-request-headers-from domain="*" headers="*"/>
</cross-domain-policy>
```

For further information on socket policies see Adobe's documentation on Flex Cross Domain Policy Files at http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

Silverlight Application Deployment

Silverlight and the Universal Messaging Policy File Server

Universal Messaging has a built in socket policy server for Silverlight applications that require tcp socket or ssl socket (NSP or NSPS) protocol.

Universal Messaging's Silverlight API enables clients to specify the communication protocol as nsp (ie tcp sockets). This means that when a Silverlight application is loaded and tries to make a socket request, Silverlight will first of all attempt to make a policy file request on port 943 to the host the original socket request was being made to. For example, if i specify an RNAME of nsp://myhost.mydomain.com:4502, when the Universal Messaging Silverlight API attempts to construct a socket on port 4502, it will first of all make a socket connect request to myhost.mydomain.com:943, and issue a policy file request.

With this in mind, Universal Messaging enables you to run a policy file server on a socket interface that will automatically handle these requests. You would first need to create a socket interface on port 943, and select the "Enable Policy Server" check box under the Basic tab for an nsp interface. Once this is setup, you will also need a clientaccesspolicy.xml file in the /install/server/name/plugins/htdocs directory of the server. This policy file might contain something like the following:

```
<?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"/>
                   <socket-resource port='4502-4534' protocol='tcp' />
              </grant-to>
```

```
            </policy>
        </cross-domain-access>
    </access-policy>
```

For further information on socket policies see Microsoft's Silverlight Policy Documentation at http://msdn.microsoft.com/en-us/library/cc645032%28v=vs.95%29.aspx.

JavaScript Application Deployment

JavaScript applications can be served directly from a Universal Messaging realm server, using a file plugin, or via a third party web server of your choice.

Serving Applications via a Universal Messaging File Plugin

For performance and security, we strongly recommend that applications are served from an SSL-encrypted file plugin. You may however choose to serve applications from a non-encrypted file plugin. See the description of using JavaScript for HTTP/HTTPS delivery in the Developer Guide.

Serving Applications via a third party Web Server

Most components of your JavaScript application can be served from any web server. A Universal Messaging File plugin is still required however, to serve certain parts of the JavaScript libraries. This is necessary to permit secure cross domain communication.

5 Security

Overview

Security

Universal Messaging provides a wide range of features to ensure that user access and data transmission is handled in a secure manner.

Universal Messaging includes built in authentication and entitlements functionality. Additionally Universal Messaging can drive 3rd party authentication and entitlements systems or be driven by organizations existing authentication and entitlements systems.

Universal Messaging makes use of standards based cryptography to provide encryption and the signing of events with digital signatures if required. Further information on Universal Messaging's security features can be found below.

Security Architecture

While distributed applications offer many benefits to their users the development of such applications can be a complex process. The ability to correctly authenticate users has been a complex issue and has lead to the emergence of standard Authentication and Authorisation frameworks, frameworks such as JAAS.

JAAS authentication is performed in a pluggable fashion. This permits applications to remain independent from underlying authentication technologies. New or updated authentication technologies can be plugged under an application without requiring modifications to the application itself.

Universal Messaging provides a wide variety of client APIs to develop enterprise, web and mobile applications. On the enterprise application front, Universal Messaging offers a transport protocol dependent authentication scheme while on the web and mobile application front a pluggable authentication framework is offered. The end result is that all applications can share the same Universal Messaging authorization scheme which requires a token@host based subject that access control lists can be defined upon.

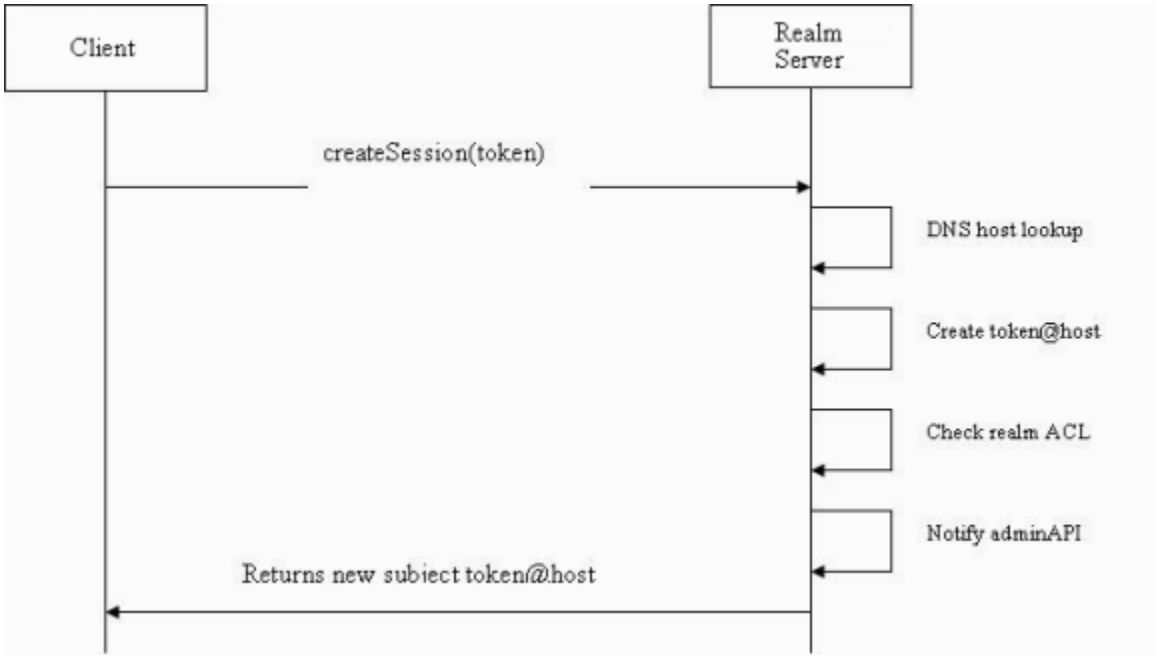
Enterprise Application Authentication

Universal Messaging enterprise applications can be written in a variety of programming languages. Each one of these client APIs offers connectivity using one of the 4 available transport protocols, namely nsp (TCP Sockets), nhp (HTTP), nsps (SSL Sockets) and nhps (HTTPS). The authentication scheme is transport protocol dependent therefore providing a basic authentication scheme for TCP based transport protocols (nsp, nhp) and an SSL authentication scheme for SSL based transport protocols (nsps, nhps).

Basic Authentication Scheme

Under this mode of authentication the client passes the username to the server as part of the initial connection handshake. The server then extracts the remote host name and creates the subject to be used by this connection.

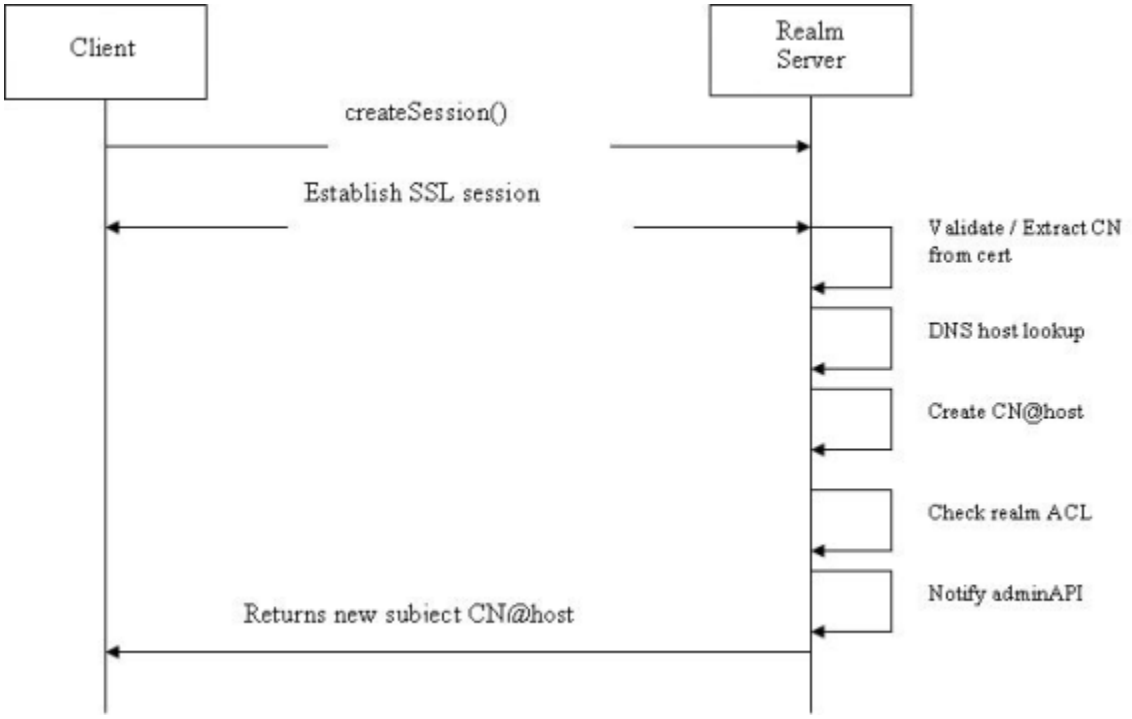
The client API can set the username component, however, the remote host is always set on the server. This stops clients from impersonating users from other hosts. The following diagram illustrates the basic authentication scheme's operation:



SSL Authentication Scheme

The Universal Messaging Realm server can be configured to perform Client Certificate authorisation or to allow anonymous SSL clients to connect. When the server is configured to allow anonymous clients to connect the subject is built up based on the previous authentication method. That is the username portion is passed to it from the client.

When the server is configured for client certificate processing the subject is constructed with the Common Name (CN) of the certificate and the remote host name. This allows the ACLs to be configured such that not only is the certificate valid but it can only access the Realm Server from a specific host. The following diagram illustrates the SSL authentication scheme's operation when using client certificates:



Web Application Authentication

Universal Messaging web applications can use a pluggable authentication framework that presents its self as basic http authentication as defined by RFC 1945. Basic authentication is supported by all popular web browsers and users have to enter a username and password in a browser provided login dialog before proceeding. The web browser then automatically includes the token in the Authorization HTTP header for all subsequent requests to the server's authentication realm, for the lifetime of the browser process. Please note that although Universal Messaging supports basic authentication on both nhp (HTTP) and nhps (HTTPS) interfaces, it is only advised to use it over HTTPS connections to secure your web application against man in the middle attacks and network sniffing tools.

In order to host your web application on Universal Messaging, a number of server side plugins are provided that you can configure and mount on the various URLs that your application expects connections on. These are the XML plugin, the Web Express plugin, the Servlet plugin, the Change Password plugin, the Realm Status plugin, the SOAP plugin, the File plugin and the Proxy Pass Through plugin.

Plugin Authentication Parameters

Each one of these plugins contains an identical set of configuration parameters that control its behavior towards authentication. These are described below:

- `Security Realm`: Name of the authentication realm
- `AddUserAsCookie`: Specifies if the authenticated username should be added as a cookie.

- `Authenticator`: Fully qualified class name of authenticator to use, or blank to use the default implementation provided.
- `AuthParameters`: A space delimited list of key=value definitions which are passed to the authenticator instance to initialize and configure it. These are passed to the Authenticator's init method.
- `GroupNames`: An optional comma separated list of groups. The user must be a member of at least one group to be granted access even if a valid username/password is provided. The groups are dependent on the authenticator implementation.
- `RoleNames`: An optional comma separated list of roles. The user must have at least one role to be granted access even if a valid username/password is provided. The roles are dependent on the authenticator implementation and are effectively the permissions defined.
- `ReloadUserFileDynamically`: If set to true, the reload method of the authenticator implementation will be called prior to serving each http request. If set to false, the reload will only be called once when the Universal Messaging interface starts.

Common AuthParameters

Irrespective of the authenticator implementation you use in your Universal Messaging server plugins, there are some AuthParameters that are also used by the server. These are:

- `NamedInstance`: This parameter requests that this authenticator configuration is bound to the specified named instance which will be shared across all plugins on this server that are configured to do so. Please note that the first plugin that accepts a connection will bind the name to the server together with the remaining configuration parameters. For this reason please make sure that configuration is always the same on all plugins that share the same instance.

Default Authenticator Implementation

Universal Messaging comes with a default authenticator implementation that uses a properties file to define users, groups and permissions (roles). In order to enable it on a Universal Messaging plugin, the Authenticator parameter needs to be left empty (this implies using the Default), the Authentication Realm set and one parameter needs to be set in AuthParameters.

The necessary parameter is called UserFile and should point to the full path of a java properties file, e.g. c:\users.txt. In order to get the Universal Messaging realm server to encrypt your user passwords, you need to add a property called initialise as shown below. This notifies the default authenticator that passwords are not encrypted so on the first load it will encrypt them, remove the initialise property and save your user file.

An example of a UserFile defining 3 permissions (roles), 3 groups and 3 users is shown below:

```
#Request password initialisation
initialise=true
#Permissions (Roles) Definition
```

```
perm_name_1=Guest
perm_name_2=User
perm_name_3=Admin
#Guests Group Definition
group_ID_Guests=1
group_desc_Guests=Guests Group
group_perm_Guests={1}
#Users Group Definition
group_ID_Users=2
group_desc_Users=Users Group
group_perm_Users={2}
#Admins Group Definition
group_ID_Admins=3
group_desc_Admins=Admins Group
group_perm_Admins={3}
#Example Guest User Definition
user_desc_someguest=Some Guest User
user_pass_someguest=password
user_perm_someguest={1}
user_home_id_someguest=Guests
user_group_someguest=Guests
#Example Regular User Definition
user_desc_someuser=Some User
user_pass_someuser=password
user_perm_someuser={1,2}
user_home_id_someuser=Users
user_group_someuser=Users
user_group_0_someuser=Guests
#Example Admin User Definition
user_desc_someadmin=Some Admin User
user_pass_someadmin=password
user_perm_someadmin={1,2,3}
user_home_id_someadmin=Admins
user_group_someadmin=Admins
user_group_0_someadmin=Guests
user_group_1_someadmin=Users
```

Custom Authenticator Implementations

The interface for creation of custom authenticator implementations is defined in the following 3 classes of the com.pcbsys.foundation.authentication package:

```
fAuthenticator: Represents the Authenticator Implementation and
                has the following methods
public void init(Hashtable initParams);
public String getName();
public synchronized void close();
public void reload();
public fPermission addPermission(int permNo, String name) ;
public fUser addUser(String username, String description,
                     String plainPassword, String groupName);
public fUser copyUser(fUser user) ;
public fUser getUser(String username);
public void delUser(fUser user);
public fGroup addGroup(int id, String name, String description);
public fPermission getPermission(int id);
public fPermission getPermission(String name);
public fGroup getGroup(String name);
public void delPermission(int id);
public void delGroup(fGroup group);
public void saveState() throws IOException
fGroup: Represents the user groups and contains the following methods:
public void reload(int id, String name, String description);
```

```
public boolean isModified();
public void setModified(boolean flag);
public String getName();
public int getId();
public String getDescription();
public BitSet getPermissions();
public void addUser(fUser aUser);
public Enumeration getUsers();
public Hashtable getUserHash();
public void setUserHash(Hashtable newhash);
public void delUser(fUser aUser);
public int getNoUsers();
public void setPermission(fPermission perm);
public void clearPermission(fPermission perm);
public void resetPermission();
public BitSet getPermissionBitSet();
public boolean can(fPermission perm);
fUser : Represents the authentication users and
        has the following methods:
public void reload(String name, String description, String password,
                   fGroup group);
public void createUser(String name, String description, String password,
                       fGroup group) ;
public void setPassword(String pass);
public BitSet getPermissions();
public BitSet getTotalPermissions();
public boolean can(fPermission perm);
public String login(byte[] password, boolean requestToken, Hashtable params);
public String login(String password, boolean requestToken, Hashtable params);
public String getHomeId();
public void setHomeId(String myHomeId);
public void setGroup(fGroup group);
public void delGroup(fGroup group);
public String getName();
public String getDescription();
public String getPassword();
public fGroup getGroup();
public Enumeration getGroups();
public Hashtable getGroupHash();
public void setGroupHash(Hashtable newhash);
public int getNumGroups();
public void setPermission(fPermission perm);
public void setDescription(String desc);
public void clearPermission(fPermission perm);
public void setPermissionBitSet(BitSet newperms);
public BitSet getPermissionBitSet();
public void resetPermission();
public boolean isModified();
public void setModified(boolean flag);
```

Example Database Authenticator

As discussed in the previous section the default implementation is based on an optionally encrypted text file, with passwords being MD5 digested. It is however possible to use different storage mechanisms for users, groups and permissions such as a relational database. There are no restrictions on the design of the database schema as Universal Messaging simply needs a set of classes that comply to the fAuthenticator, fGroup and fUser interfaces. Please note that not all classes need to be subclassed but only the ones that you need to modify the default behaviour.

In the context of this example we are going to use a mysql database running on localhost and containing a users table with the following columns:

- varchar
- varchar
- int
- varchar

In order to keep the example simple we are going to statically define the groups and permissions within the authenticator source code. We will use the group functionality on the base fGroup class and therefore will only subclass fAuthenticator and fUser as shown below:

DBAuthenticator

```
package com.myapp;
import com.pcbsys.foundation.authentication.*;
import com.pcbsys.nirvana.client.*;
import com.mysql.jdbc.Driver;
import java.io.*;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DriverManager;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
public class DBAuthenticator extends fAuthenticator {
  private static fGroup MYAPP_GROUP =null;
  private static fGroup MYCOMPANY_GROUP =null;
  protected static fPermission CLIENT_PERMISSION=null;
  protected static fPermission ADMIN_PERMISSION=null;
  private static fPermission GUEST_PERMISSION=null;
  private boolean initialised=false;
  private String myName="DBAuthenticator";
  private static int myUniqueID=0;
  private static Connection myConnection;
  private static String jdbcurl = "jdbc:mysql://localhost:3306/test";
  private static String myDBUser="root";
  private static String myDBPassword="";
    //Lets statically define the groups and permissions
    static {
        //Company Group
        MYCOMPANY_GROUP =new fGroup();
        MYCOMPANY_GROUP.reload(2,"mycompany", "MyCompany Group");
        //Application Group
        MYAPP_GROUP =new fGroup();
        MYAPP_GROUP.reload(0,"mycompany/myapp", "MyApp Group");
        GUEST_PERMISSION=new fPermission();
        GUEST_PERMISSION.reload(0,"Guest");
        CLIENT_PERMISSION=new fPermission();
        CLIENT_PERMISSION.reload(1,"Client");
        ADMIN_PERMISSION=new fPermission();
        ADMIN_PERMISSION.reload(4,"Admin");
    }
  public void close(){
    super.close();
    if(getUsageCount() == 0){
```

```
      fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
          "Closing Authenticator ["+getUsageCount()+"]");
      //release connection pool
      if (myConnection!=null){
          try {
              myConnection.close();
          } catch (SQLException e) {}
          myConnection=null;
      }
      initialised=false;
  }
  else {
      fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
            "Closing Authenticator ["+getUsageCount()+"]");
  }
}
public DBAuthenticator() {
  super();
  addGroup(MYCOMPANY_GROUP);
  addGroup(MYAPP_GROUP);
  getPermissionsCollection().put("Client", CLIENT_PERMISSION);
  getPermissionsCollection().put("Admin", ADMIN_PERMISSION);
  getPermissionsCollection().put("Guest", GUEST_PERMISSION);
}
protected static Connection getDBConnection() throws SQLException{
    if (myConnection==null){
      try {
          Class.forName("com.mysql.jdbc.Driver").newInstance();
          myConnection = DriverManager.getConnection(jdbcurl,myDBUser,
             myDBPassword);
      } catch (InstantiationException e) {
          e.printStackTrace();
          //To change body of catch statement use
          // File | Settings | File Templates.
      } catch (IllegalAccessException e) {
          e.printStackTrace();
          //To change body of catch statement use
          // File | Settings | File Templates.
      } catch (ClassNotFoundException e) {
          e.printStackTrace();
          //To change body of catch statement use
          // File | Settings | File Templates.
      }
    }
  return myConnection;
}
  public String getName() {
      return myName;
  }
  private static String getNextId() {
      return ""+myUniqueID++;
  }
  public void init(Hashtable initParams) throws IOException {
      if (!initialised){
          if (initParams.containsKey("NamedInstance")){
                  myName=(String)initParams.get("NamedInstance");
              }
              else {
                  myName=myName+"_"+getNextId();
                  fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
                    "Default Instance Requested ");
              }
              if (initParams.get("DBUser")!=null){
```

```
                myDBUser=(String)initParams.get("DBUser");
            }
            if (initParams.get("DBPassword")!=null){
                myDBPassword=(String)initParams.get("DBPassword");
            }
            if (initParams.get("JDBCURL")!=null){
                jdbcurl=(String)initParams.get("JDBCURL");
            }
            initialised=true;
        }
}
private DBUser createDBUserInstance(String username, String description,
       String password, int permissions, int home){
    DBUser someuser=new DBUser();
    someuser.setAuthenticator(this);
    if (home==1){ //MyCompany/MyApp Users
       someuser.reload(username,description,password, MYAPP_GROUP);
       //Set home
       someuser.setHomeId(MYAPP_GROUP.getName());
       //Group association logic
       someuser.setGroup(MYAPP_GROUP);
       //Set outer group
       someuser.setGroup(MYCOMPANY_GROUP);
       MYCOMPANY_GROUP.addUser(someuser);
       //Set inner group
       MYAPP_GROUP.addUser(someuser);
       //Permissions association logic
       switch (permissions){
            case 1:
                someuser.setPermission(CLIENT_PERMISSION);
                break;
            case 10:
                someuser.setPermission(ADMIN_PERMISSION);
                break;
            default:{
                someuser.setPermission(GUEST_PERMISSION);
                break;
            }
        }
    }
    else {
        fAuthenticator.logAuthenticatorMessage("WARNING: User "+
           username+" has a home value of "+home+
           ". User will be ignored!");
        return null;
    }
    return someuser;
}
private DBUser LoadUserFromDatabase(String username){
    DBUser someuser=null;
    Connection conn =null;
    java.sql.Statement stmt = null;
    java.sql.ResultSet rset=null;
    String name=null;
    try{
    conn=getDBConnection();
    stmt = conn.createStatement();
    rset = stmt.executeQuery(
      "select name,password, rights,home from USERS where name='"+
       username.toLowerCase()+"'" );
        while(rset.next())
        {
           int permissions= rset.getInt("RIGHTS");
```

```
            int home = rset.getInt("HOME"); // home desk association
            name=rset.getString("name").toLowerCase();
            String password=rset.getString("password");
            if (password==null || password.trim().length()==0 ||
               password.equals("null")) password="nopassword";
            String description="A "+name+" user";
            //safeguard for users without a description!
            someuser=createDBUserInstance(name,description,password,
               permissions,home);
            if (someuser==null) continue;
            //In case we have invalid data to create this user object
             //Cache instance
             getUsersCollection().put(someuser.getName(),someuser);
        }
        rset.close();
        rset=null;
        stmt.close();
        stmt=null;
    }
    catch (Throwable t){
        logAuthenticatorException(
           "DBAuthenticator: Error obtaining details for user "+name);
        logAuthenticatorException(t);
        t.printStackTrace();
    }
    finally {
        if (rset!=null){
            try {
                rset.close();
            } catch (SQLException e) {
            }
            rset=null;
        }
        if (stmt!=null){
            try {
                stmt.close();
            } catch (SQLException e) {
            }
            stmt=null;
        }
    }
    return someuser;
}
private void LoadUsersFromDatabase(){
    Connection conn =null;
    java.sql.Statement stmt=null;
    java.sql.ResultSet rset=null;
    String name=null;
    try{
    conn=getDBConnection();
        stmt = conn.createStatement();
        rset = stmt.executeQuery(
           "select name,password, rights,home from USERS order by name" );
        while(rset.next())
        {
           int rights= rset.getInt("RIGHTS");
           int home = rset.getInt("HOME"); // home desk association
           name=rset.getString("name").toLowerCase();
           String password=rset.getString("password");
           if (password==null || password.trim().length()==0 ||
               password.equals("null")) password="nopassword";
           String description="A "+name+" user";
           DBUser someuser=createDBUserInstance(name,description,
```

```
                password,rights,home);
            if (someuser==null) continue;
            getUsersCollection().put(someuser.getName(),someuser);
        }
        rset.close();
        rset=null;
        stmt.close();
        stmt=null;
    }
    catch (Throwable t){
        logAuthenticatorException("Error obtaining details for user "+
            name);
        logAuthenticatorException(t);
        t.printStackTrace();
    }
    finally {
        if (rset!=null){
            try {
                rset.close();
            } catch (SQLException e) {
            }
            rset=null;
        }
        if (stmt!=null){
            try {
                stmt.close();
            } catch (SQLException e) {
            }
            stmt=null;
        }
    }
}
public void reload() throws IOException {
    LoadUsersFromDatabase();
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
        "Reload called");
}
/**
 * Creates a new fPermission with the unique ID and name supplied.
 *
 * The implementation should save the new permission to the relevant
 * technology used.
 *
 * @param permNo Unique ID from 0 to 63.
 * @param name   Name describing this new permission.
 * @return the new fPermission.
 * @throws java.io.IOException If unable to create the new fPermission.
 */
public fPermission addPermission(int permNo, String name) throws
   IOException {
if(getPermissionsCollection().get(""+permNo) == null){
  fPermission perm = new fPermission();
  perm.reload(permNo,name);
  getPermissionsCollection().put(""+permNo, perm);
  return perm;
}
fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
     "Added Permission "+name+"("+permNo+")");
return (fPermission) super.getPermissionsCollection().get(""+permNo);
}
public fUser addUser(String username, String description,
    String plainPassword, String groupName) throws IOException {
fGroup group = null;
```

```
if (groupName != null) {
  group = (fGroup) getGroupsCollection().get(groupName);
  if (group == null) throw new IOException("No known group " + groupName);
}
fUser user = createUser(username, description, plainPassword, group);
getUsersCollection().put(user.getName(), user);
if(group != null) group.addUser(user);
fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+"Added User "+
     username+" NOTE: This is not currently persisted in the database!");
return user;
}
/**
 * Creates a new fUser with the supplied values.
 * The password field is passed as plain text but it is up to the
 * implementation to ensure the password
 * is secure.
 *
 * The implementation should save the new user to the relevant
 * technology used.
 *
 * @param user The user to copy.
 * @return The new fUser created.
 * @throws java.io.IOException If there where any errors during the
 * construction of the user.
 */
public fUser copyUser(fUser user) throws IOException {
    fGroup group = null;
    group = (fGroup) getGroupsCollection().get(user.getGroup().getName());
    fUser aUser = createUser(user.getName(),user.getDescription(),
       user.getPassword(),user.getGroup());
    getUsersCollection().put(aUser.getName(), aUser);
    if(group != null) group.addUser(aUser);
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
        "Copied User "+user.getName());
    return aUser;
}
/**
 * Adds a new group with the supplied values.
 *
 * The implementation should save the new group to the relevant
 * technology used.
 *
 * @param id          Unique ID for the group.
 * @param name        Name of the new group.
 * @param description Description of the new group.
 * @return The new fGroup object.
 * @throws java.io.IOException If unable to create the new fGroup object.
 */
public fGroup addGroup(int id, String name, String description) throws
   IOException {
    fGroup group = new fGroup();
    group.reload(id, name, description);
    addGroup(group);
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
       "Added Group "+group.getName());
    return group;
}
/**
 * Returns the permission with the ID supplied or null if not found.
 *
 * @param id fPermission Id to search for.
 * @return the fPermission or null if not found.
 */
```

```
 public fPermission getPermission(int id) {
   Enumeration perms = getPermissionsCollection().elements();
   while (perms.hasMoreElements()) {
     fPermission fPermission = (fPermission) perms.nextElement();
     if (fPermission.getId() == id) return fPermission;
   }
   return null;
 }
 /**
  * Returns the permission with the name supplied or null if not found.
  *
  * @param name fPermission name to search for.
  * @return the fPermission or null if not found.
  */
 public fPermission getPermission(String name) {
     return (fPermission)getPermissionsCollection().get(name);
 }
public Enumeration getUsers(){
    return  getUsersCollection().elements();
}
 public fUser getUser(String username) {
   return (fUser) LoadUserFromDatabase(username.toLowerCase());
 }
 public fGroup getGroup(String name) {
     return (fGroup)getGroupsCollection().get(name);
 }
 /**
  * Removes the permission with the ID supplied.
  *
  * The implementation should remove the permission from the relevant
  * technology used.
  *
  * @param id of the permission to delete.
  * @throws java.io.IOException if unable to delete the permission.
  */
 public void delPermission(int id) throws IOException {
 getPermissionsCollection().remove(""+id);
   fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
       "Deleted permission ("+id+")");
 }
 /**
  * Removes the user supplied.
  *
  * The implementation should remove the user from the relevant
  * technology used.
  *
  * @param user fUser object to remove.
  * @throws java.io.IOException If unable to remove the user.
  */
 public void delUser(fUser user) throws IOException {
 if (user.getGroup() != null) {
   user.getGroup().delUser(user);
 }
 getUsersCollection().remove(user.getName());
   fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
      "Deleted User "+user.getName());
 }
 /**
  * Removes the supplied fGroup object.
  *
  * Any user currently a member of this group will have the group reset
  * to null meaning no group membership.
  *
```

```
     * The implementation should remove the group from the relevant
     * technology used.
     *
     * @param group Group to remove.
     * @throws java.io.IOException If unable to remove the group.
     */
    public void delGroup(fGroup group) throws IOException {
        Enumeration enm = group.getUsers();
        while (enm.hasMoreElements()) {
          fUser user = (fUser) enm.nextElement();
          user.delGroup(group);
        }
        getGroupsCollection().remove(group.getName());
        fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
           "Deleted Group "+group.getName());
    }
    /**
     * Requests that the implementation save the current state.
     *
     * This should include all users, groups and permissions.
     *
     * @throws java.io.IOException if the save failed.
     */
    public void saveState() throws IOException {
       //TODO: Implement saving of data to the database
    }
    public void roll() throws IOException {
        //TODO: Implement any log file rolling
    }
    protected fUser createUser(String name, String desc, String password,
      fGroup group){
      //  System.out.println("CreateUSer being called");
      DBUser usr = new DBUser();
      usr.reload(name, desc, password, group);
      usr.setHomeId(group.getName());
      usr.setAuthenticator(this);
      return usr;
    }
}
```

DBUser

```
package com.myapp;
import com.pcbsys.foundation.authentication.fUser;
import com.pcbsys.foundation.authentication.fGroup;
import com.pcbsys.foundation.authentication.fAuthenticator;
import java.util.Hashtable;
import java.sql.SQLException;
public class DBUser extends fUser {
  private static fAuthenticator myAuthenticator;
  protected DBUser(){
    super();
    super.setGroupHash(new Hashtable());
  }
  //Allow setting a reference to the authenticator instance so that we can
  // obtain its DB connection for
  // user authentication purposes
  public static void setAuthenticator (fAuthenticator authenticator){
      myAuthenticator=authenticator;
  }
 protected DBUser(String name, String desc, String password){
    this(name, desc, password, null);
  }
```

```
  protected DBUser(String name, String desc, String password, fGroup group){
    super(name,desc,password,group);
  }
 public String login(byte[] password, boolean requestToken){
     return login(password,requestToken,null);
 }
public String login(String password, boolean requestToken){
    return login(password,requestToken,null);
}
 public String login(byte[] password, boolean requestToken, Hashtable params){
   return login(new String(password), requestToken, params);
  }
 public String login(String password, boolean requestToken, Hashtable params){
        java.sql.Connection conn =null;
        java.sql.Statement stmt=null;
        java.sql.ResultSet rset=null;
        String name=null;
        try{
        conn=((DBAuthenticator)myAuthenticator).getDBConnection();
        stmt = conn.createStatement();
        rset = stmt.executeQuery(
           "select password, rights from USERS where name ='"+
            this.getName()+"'" );
            while(rset.next())
            {
               int rights= rset.getInt("RIGHTS");
               String thepassword=rset.getString("password");
               if (thepassword.equals(password)){
                   if (rights > 0) return "true";
               }
            }
            rset.close();
            rset=null;
            stmt.close();
            stmt=null;
        }
        catch (Throwable t){
            myAuthenticator.logAuthenticatorException(
                "DBAuthenticator: Error obtaining details for user "+name);
            myAuthenticator.logAuthenticatorException(t);
        }
        finally {
            if (rset!=null){
                try {
                    rset.close();
                } catch (SQLException e) {
                }
                rset=null;
            }
            if (stmt!=null){
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt=null;
            }
        }
     return null;
  }
}
```

Web Application Single Sign On

Single sign-on (SSO) is a method of access control that enables a user to log in once and gain access to the required application and its resources without being prompted to log in again. When developing multi node Universal Messaging web applications, you have to take into consideration that incidents such a network failure, could cause the user's browser to fail over to a different node than the one initially authenticated with. In order to prevent the application user from authenticating again, or to integrate your Universal Messaging web application to a 3d party authentication mechanism and provide alternative authentication user interfaces, you can use Single Sign On Interceptors (SSI).

A Single Sign On Interceptor (SSI) is a class that conforms to a specific interface and gets invoked by the Universal Messaging realm prior to authentication in order to decide which of the following 3 outcomes should occur:

- If the user meets the criteria required, allow access to the plugin content as if they where normally authenticated, optionally generating a unique session id.
- If the user does not meet the criteria required, but a redirection is configured, redirect their browsers to the specified URL in order to authenticate.
- If the user does not meet the criteria required, and no redirection is configured, then fall back to the regular authenticator configured.

The interface for creation of a nirvana SSI implementations is defined in the following 2 classes of the com.pcbsys.foundation.authentication package:

fSSInterceptor

```
//Return an fSSUser with a null username to fall back to authenticator
// (or redirect if a URL is set)
public abstract fSSUser getSSUser(Hashtable httpHeaders, Hashtable urlParameters);
public abstract void setParameters(Hashtable params);
public abstract void clear();
public abstract String getName();
```

fSSUser

```
public fSSUser(String username, String redirectURL);
public fSSUser(String username, String redirectURL, String token);
public String getUsername();
public String getRedirectURL();
public String getToken()
```

Plugin Single Sign On Interceptor Parameters

- SSInterceptor: Fully qualified class name of SSI to use. If not specified, no interceptor will be used
- SSOAppendToken: Setting this parameter to true instructs the SSI object to generate and return a unique session ID when an affirmative single sign on decision is reached. Please note that the absence of a session ID is irrelevant to the single sign on decision.

Common Single Sign On AuthParameters

Irrespective of the Single Sign On implementation you use in your Universal Messaging server plugins, there are some AuthParameters that are also used by the server. These are:

- `SSONamedInstance`: This optional parameter requests that this SSI object is bound to the specified named instance which will be shared across all plugins on this server that are configured to do so. Please note that the first plugin that accepts a connection will bind the name to the server together with the remaining configuration parameters. For this reason please make sure that configuration is always the same on all plugins that share the same SSI instance.
- `REDIRECT_URL`: This optional parameter specifies the URL that a web client should be redirected to should the interceptor's criteria are not met. This allows the creation of alternative authentication methods such as form based authentication or others.

Mobile Application Authentication

When developing Universal Messaging based mobile applications, authentication is dependent on your mobile technology of choice. This is because the Universal Messaging Blackberry API authentication works exactly like any Universal Messaging enterprise application (with the exception of SSL client certificates) while the Universal Messaging IPhone application works exactly like any Universal Messaging web application.

Using HTTP/HTTPS

The Universal Messaging messaging APIs provides a rich set of functionality that can be used over sockets, SSL, HTTP and HTTPS. The code used to connect to the Universal Messaging server is the same regardless of which network protocol you are using to connect.

Under the Universal Messaging programming model there are a number of logical steps that need to be followed in order to establish a connection to a Universal Messaging sever (Realm). These involve establishing a session, obtaining a reference to a channel or a transaction, or registering an object as a subscriber.

Universal Messaging fully supports HTTP and HTTPS. Rather than tunnel an existing protocol through HTTP Universal Messaging has a pluggable set of communications drivers supporting TCP/IP Sockets, SSL enabled TCP/IP sockets, HTTP and HTTPS. Both the client and server make use of these pluggable drivers. From the server perspective different driver types can be assigned to specific Universal Messaging interfaces. From a client perspective a Universal Messaging session can be built on any one of the available drivers dynamically.

Please note that before making an HTTP/HTTPS connection to a Universal Messaging realm server you will first need to add a HTTP/HTTPS interface to the realm.

To create a connect to a Universal Messaging Realm over HTTPS you would use an RNAME (see "Communication Protocols and RNAMEs" on page 22) that specific the Universal Messaging HTTPS protocol (nhps) as follows:

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

```
//use an RNAME indicating the wire protocol you are using (HTTPS in this case)
//you can pass an array of up to four values for RNAME for added robustness
String[] RNAME=( {"nhps://remoteHost:443" } );
nSessionAttrib nsa = new nSessionAttrib( RNAME );
```

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

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

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

```
Public class myReconnectHandler implements nReconnectHandler {
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();
```

After initialising your Universal Messaging session, you will be connected to the Universal Messaging Realm using HTTPS. 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.

Authentication

Authentication

While distributed applications offer many benefits to their users the development of such applications can be a complex process. The ability to correctly authenticate users has been a complex issue and has lead to the emergence of standard Authentication and Authorisation frameworks, frameworks such as JAAS.

JAAS authentication is performed in a pluggable fashion. This permits applications to remain independent from underlying authentication technologies. New or updated authentication technologies can be plugged under an application without requiring modifications to the application itself.

Universal Messaging provides a wide variety of client APIs to develop enterprise, web and mobile applications. On the enterprise application front, Universal Messaging offers a transport protocol dependent authentication scheme while on the web and mobile application front a pluggable authentication framework is offered. The end result is that all applications can share the same Universal Messaging authorization scheme which requires a token@host based subject that access control lists can be defined upon.

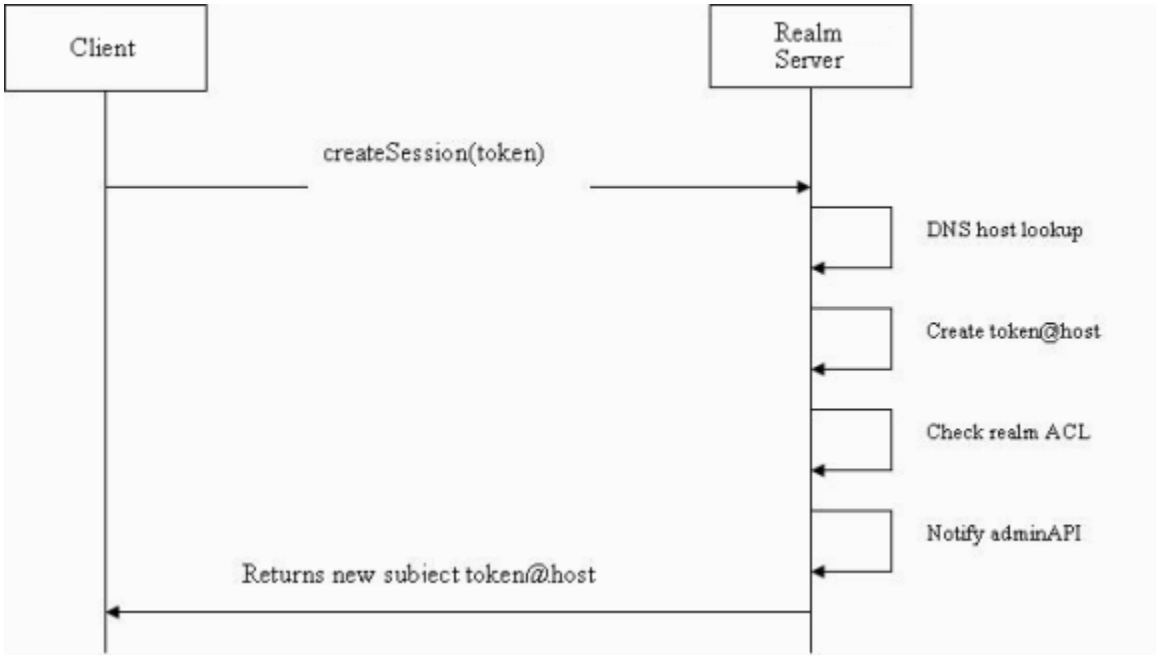
Enterprise Application Authentication

Universal Messaging enterprise applications can be written in a variety of programming languages. Each one of these client APIs offers connectivity using one of the 4 available transport protocols, namely nsp (TCP Sockets), nhp (HTTP), nsps (SSL Sockets) and nhps (HTTPS). The authentication scheme is transport protocol dependent therefore providing a basic authentication scheme for TCP based transport protocols (nsp, nhp) and an SSL authentication scheme for SSL based transport protocols (nsps, nhps).

Basic Authentication Scheme

Under this mode of authentication the client passes the username to the server as part of the initial connection handshake. The server then extracts the remote host name and creates the subject to be used by this connection.

The client API can set the username component, however, the remote host is always set on the server. This stops clients from impersonating users from other hosts. The following diagram illustrates the basic authentication scheme's operation:

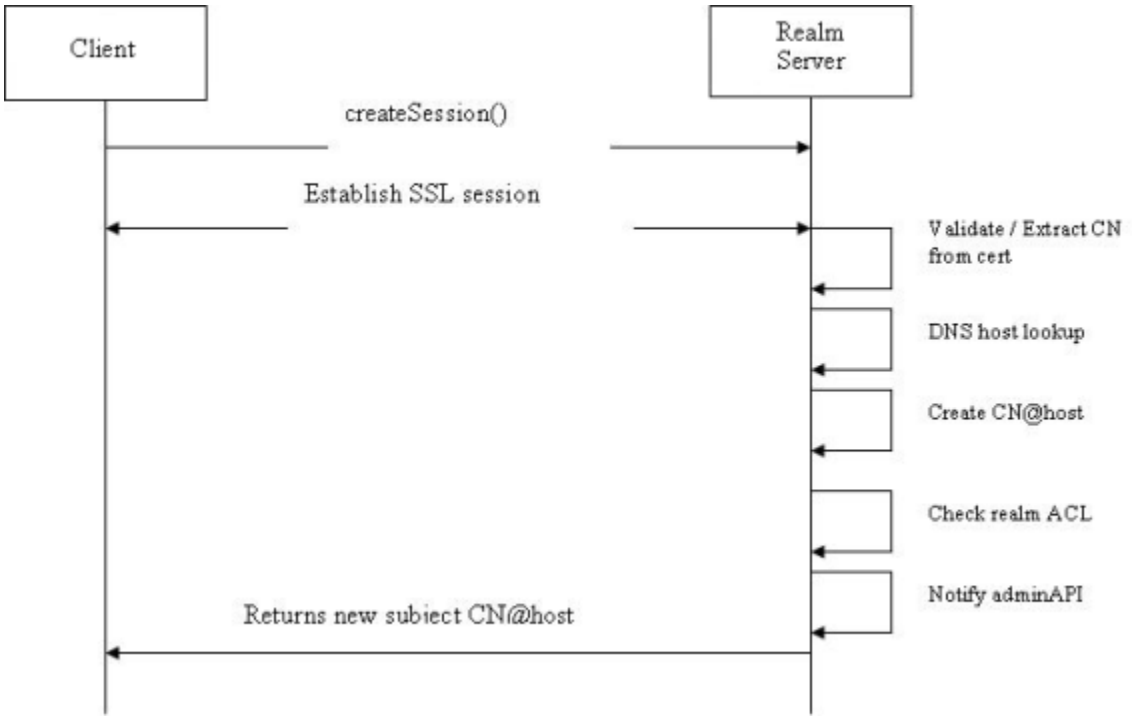


SSL Authentication Scheme

The Universal Messaging Realm server can be configured to perform Client Certificate authorisation or to allow anonymous SSL clients to connect. When the server is configured to allow anonymous clients to connect the subject is built up based on the previous authentication method. That is the username portion is passed to it from the client.

When the server is configured for client certificate processing the subject is constructed with the Common Name (CN) of the certificate and the remote host name. This allows the ACLs to be configured such that not only is the certificate valid but it can only

access the Realm Server from a specific host. The following diagram illustrates the SSL authentication scheme's operation when using client certificates:



Web Application Authentication

Universal Messaging web applications can use a pluggable authentication framework that presents its self as basic http authentication as defined by RFC 1945. Basic authentication is supported by all popular web browsers and users have to enter a username and password in a browser provided login dialog before proceeding. The web browser then automatically includes the token in the Authorization HTTP header for all subsequent requests to the server's authentication realm, for the lifetime of the browser process. Please note that although Universal Messaging supports basic authentication on both nhp (HTTP) and nhps (HTTPS) interfaces, it is only advised to use it over HTTPS connections to secure your web application against man in the middle attacks and network sniffing tools.

In order to host your web application on Universal Messaging, a number of server side plugins are provided that you can configure and mount on the various URLs that your application expects connections on. These are the XML plugin, the Servlet plugin, the Change Password plugin, the Realm Status plugin, the SOAP plugin, the File plugin and the Proxy Pass Through plugin.

Plugin Authentication Parameters

Each one of these plugins contains an identical set of configuration parameters that control its behavior towards authentication. These are described below:

- `Security Realm`: Name of the authentication realm

- `AddUserAsCookie`: Specifies if the authenticated username should be added as a cookie.
- `Authenticator`: Fully qualified class name of authenticator to use, or blank to use the default implementation provided.
- `AuthParameters`: A space delimited list of key=value definitions which are passed to the authenticator instance to initialize and configure it. These are passed to the Authenticator's init method.
- `GroupNames`: An optional comma separated list of groups. The user must be a member of at least one group to be granted access even if a valid username/password is provided. The groups are dependent on the authenticator implementation.
- `RoleNames`: An optional comma separated list of roles. The user must have at least one role to be granted access even if a valid username/password is provided. The roles are dependent on the authenticator implementation and are effectively the permissions defined.
- `ReloadUserFileDynamically`: If set to true, the reload method of the authenticator implementation will be called prior to serving each http request. If set to false, the reload will only be called once when the Universal Messaging interface starts.

Common AuthParameters

Irrespective of the authenticator implementation you use in your Universal Messaging server plugins, there are some AuthParameters that are also used by the server. These are:

- `NamedInstance`: This parameter requests that this authenticator configuration is bound to the specified named instance which will be shared across all plugins on this server that are configured to do so. Please note that the first plugin that accepts a connection will bind the name to the server together with the remaining configuration parameters. For this reason please make sure that configuration is always the same on all plugins that share the same instance.

Default Authenticator Implementation

Universal Messaging comes with a default authenticator implementation that uses a properties file to define users, groups and permissions (roles). In order to enable it on a Universal Messaging plugin, the Authenticator parameter needs to be left empty (this implies using the Default), the Authentication Realm set and one parameter needs to be set in AuthParameters.

The necessary parameter is called UserFile and should point to the full path of a java properties file, e.g. c:\users.txt. In order to get the Universal Messaging realm server to encrypt your user passwords, you need to add a property called initialise as shown below. This notifies the default authenticator that passwords are not encrypted so on the first load it will encrypt them, remove the initialise property and save your user file.

An example of a UserFile defining 3 permissions (roles), 3 groups and 3 users is shown below:

```
#Request password initialisation
initialise=true
#Permissions (Roles) Definition
perm_name_1=Guest
perm_name_2=User
perm_name_3=Admin
#Guests Group Definition
group_ID_Guests=1
group_desc_Guests=Guests Group
group_perm_Guests={1}
#Users Group Definition
group_ID_Users=2
group_desc_Users=Users Group
group_perm_Users={2}
#Admins Group Definition
group_ID_Admins=3
group_desc_Admins=Admins Group
group_perm_Admins={3}
#Example Guest User Definition
user_desc_someguest=Some Guest User
user_pass_someguest=password
user_perm_someguest={1}
user_home_id_someguest=Guests
user_group_someguest=Guests
#Example Regular User Definition
user_desc_someuser=Some User
user_pass_someuser=password
user_perm_someuser={1,2}
user_home_id_someuser=Users
user_group_someuser=Users
user_group_0_someuser=Guests
#Example Admin User Definition
user_desc_someadmin=Some Admin User
user_pass_someadmin=password
user_perm_someadmin={1,2,3}
user_home_id_someadmin=Admins
user_group_someadmin=Admins
user_group_0_someadmin=Guests
user_group_1_someadmin=Users
```

Custom Authenticator Implementations

The interface for creation of custom authenticator implementations is defined in the following 3 classes of the com.pcbsys.foundation.authentication package:

```
fAuthenticator: Represents the Authenticator Implementation and
                has the following methods
public void init(Hashtable initParams);
public String getName();
public synchronized void close();
public void reload();
public fPermission addPermission(int permNo, String name) ;
public fUser addUser(String username, String description, String plainPassword,
       String groupName);
public fUser copyUser(fUser user) ;
public fUser getUser(String username);
public void delUser(fUser user);
public fGroup addGroup(int id, String name, String description);
public fPermission getPermission(int id);
```

```
public fPermission getPermission(String name);
public fGroup getGroup(String name);
public void delPermission(int id);
public void delGroup(fGroup group);
public void saveState() throws IOException
fGroup: Represents the user groups and contains the following methods:
public void reload(int id, String name, String description);
public boolean isModified();
public void setModified(boolean flag);
public String getName();
public int getId();
public String getDescription();
public BitSet getPermissions();
public void addUser(fUser aUser);
public Enumeration getUsers();
public Hashtable getUserHash();
public void setUserHash(Hashtable newhash);
public void delUser(fUser aUser);
public int getNoUsers();
public void setPermission(fPermission perm);
public void clearPermission(fPermission perm);
public void resetPermission();
public BitSet getPermissionBitSet();
public boolean can(fPermission perm);
fUser : Represents the authentication users and has the following methods:
public void reload(String name, String description, String password,
       fGroup group);
public void createUser(String name, String description, String password,
       fGroup group) ;
public void setPassword(String pass);
public BitSet getPermissions();
public BitSet getTotalPermissions();
public boolean can(fPermission perm);
public String login(byte[] password, boolean requestToken, Hashtable params);
public String login(String password, boolean requestToken, Hashtable params);
public String getHomeId();
public void setHomeId(String myHomeId);
public void setGroup(fGroup group);
public void delGroup(fGroup group);
public String getName();
public String getDescription();
public String getPassword();
public fGroup getGroup();
public Enumeration getGroups();
public Hashtable getGroupHash();
public void setGroupHash(Hashtable newhash);
public int getNumGroups();
public void setPermission(fPermission perm);
public void setDescription(String desc);
public void clearPermission(fPermission perm);
public void setPermissionBitSet(BitSet newperms);
public BitSet getPermissionBitSet();
public void resetPermission();
public boolean isModified();
public void setModified(boolean flag);
```

Example Database Authenticator

As discussed in the previous section the default implementation is based on an optionally encrypted text file, with passwords being MD5 digested. It is however possible to use different storage mechanisms for users, groups and permissions such as a relational database. There are no restrictions on the design of the database schema

as Universal Messaging simply needs a set of classes that comply to the fAuthenticator, fGroup and fUser interfaces. Please note that not all classes need to be subclassed but only the ones that you need to modify the default behaviour.

In the context of this example we are going to use a mysql database running on localhost and containing a users table with the following columns:

- varchar
- varchar
- int
- varchar

In order to keep the example simple we are going to statically define the groups and permissions within the authenticator source code. We will use the group functionality on the base fGroup class and therefore will only subclass fAuthenticator and fUser as shown below:

DBAuthenticator

```
package com.myapp;
import com.pcbsys.foundation.authentication.*;
import com.pcbsys.nirvana.client.*;
import com.mysql.jdbc.Driver;
import java.io.*;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DriverManager;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
public class DBAuthenticator extends fAuthenticator {
  private static fGroup MYAPP_GROUP =null;
  private static fGroup MYCOMPANY_GROUP =null;
  protected static fPermission CLIENT_PERMISSION=null;
  protected static fPermission ADMIN_PERMISSION=null;
  private static fPermission GUEST_PERMISSION=null;
  private boolean initialised=false;
  private String myName="DBAuthenticator";
  private static int myUniqueID=0;
  private static Connection myConnection;
  private static String jdbcurl = "jdbc:mysql://localhost:3306/test";
  private static String myDBUser="root";
  private static String myDBPassword="";
    //Lets statically define the groups and permissions
    static {
        //Company Group
        MYCOMPANY_GROUP =new fGroup();
        MYCOMPANY_GROUP.reload(2,"mycompany", "MyCompany Group");
        //Application Group
        MYAPP_GROUP =new fGroup();
        MYAPP_GROUP.reload(0,"mycompany/myapp", "MyApp Group");
        GUEST_PERMISSION=new fPermission();
        GUEST_PERMISSION.reload(0,"Guest");
        CLIENT_PERMISSION=new fPermission();
        CLIENT_PERMISSION.reload(1,"Client");
        ADMIN_PERMISSION=new fPermission();
        ADMIN_PERMISSION.reload(4,"Admin");
```

```
  }
public void close(){
  super.close();
  if(getUsageCount() == 0){
      fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
         "Closing Authenticator ["+getUsageCount()+"]");
      //release connection pool
      if (myConnection!=null){
          try {
              myConnection.close();
          } catch (SQLException e) {}
          myConnection=null;
      }
      initialised=false;
  }
  else {
      fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
         "Closing Authenticator ["+getUsageCount()+"]");
  }
}
public DBAuthenticator() {
  super();
  addGroup(MYCOMPANY_GROUP);
  addGroup(MYAPP_GROUP);
  getPermissionsCollection().put("Client", CLIENT_PERMISSION);
  getPermissionsCollection().put("Admin", ADMIN_PERMISSION);
  getPermissionsCollection().put("Guest", GUEST_PERMISSION);
}
protected static Connection getDBConnection() throws SQLException{
    if (myConnection==null){
      try {
          Class.forName("com.mysql.jdbc.Driver").newInstance();
          myConnection = DriverManager.getConnection(jdbcurl,myDBUser,
                         myDBPassword);
      } catch (InstantiationException e) {
          e.printStackTrace();
          //To change body of catch statement use
          // File | Settings | File Templates.
      } catch (IllegalAccessException e) {
          e.printStackTrace();
          //To change body of catch statement use
          //File | Settings | File Templates.
      } catch (ClassNotFoundException e) {
          e.printStackTrace();
         //To change body of catch statement use
         // File | Settings | File Templates.
      }
    }
  return myConnection;
}
  public String getName() {
      return myName;
  }
  private static String getNextId() {
      return ""+myUniqueID++;
  }
  public void init(Hashtable initParams) throws IOException {
      if (!initialised){
          if (initParams.containsKey("NamedInstance")){
                  myName=(String)initParams.get("NamedInstance");
              }
              else {
                  myName=myName+"_"+getNextId();
```

```
                fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
                   "Default Instance Requested ");
            }
            if (initParams.get("DBUser")!=null){
                myDBUser=(String)initParams.get("DBUser");
            }
            if (initParams.get("DBPassword")!=null){
                myDBPassword=(String)initParams.get("DBPassword");
            }
            if (initParams.get("JDBCURL")!=null){
                jdbcurl=(String)initParams.get("JDBCURL");
            }
            initialised=true;
        }
}
private DBUser createDBUserInstance(String username, String description,
      String password, int permissions, int home){
    DBUser someuser=new DBUser();
    someuser.setAuthenticator(this);
    if (home==1){ //MyCompany/MyApp Users
       someuser.reload(username,description,password, MYAPP_GROUP);
       //Set home
       someuser.setHomeId(MYAPP_GROUP.getName());
       //Group association logic
       someuser.setGroup(MYAPP_GROUP);
       //Set outer group
       someuser.setGroup(MYCOMPANY_GROUP);
       MYCOMPANY_GROUP.addUser(someuser);
       //Set inner group
       MYAPP_GROUP.addUser(someuser);
       //Permissions association logic
       switch (permissions){
            case 1:
                someuser.setPermission(CLIENT_PERMISSION);
                break;
            case 10:
                someuser.setPermission(ADMIN_PERMISSION);
                break;
            default:{
                someuser.setPermission(GUEST_PERMISSION);
                break;
            }
        }
    }
    else {
        fAuthenticator.logAuthenticatorMessage("WARNING: User "+username+
         " has a home value of "+home+". User will be ignored!");
        return null;
    }
    return someuser;
}
private DBUser LoadUserFromDatabase(String username){
    DBUser someuser=null;
    Connection conn =null;
    java.sql.Statement stmt = null;
    java.sql.ResultSet rset=null;
    String name=null;
    try{
    conn=getDBConnection();
    stmt = conn.createStatement();
    rset = stmt.executeQuery(
        "select name,password, rights,home from USERS where name='"+
           username.toLowerCase()+"'" );
```

```
            while(rset.next())
            {
               int permissions= rset.getInt("RIGHTS");
               int home = rset.getInt("HOME"); // home desk association
               name=rset.getString("name").toLowerCase();
               String password=rset.getString("password");
               if (password==null || password.trim().length()==0 ||
                   password.equals("null")) password="nopassword";
               String description="A "+name+" user";
                  //safeguard for users without a description!
               someuser=createDBUserInstance(name,description,password,
                   permissions,home);
               if (someuser==null) continue;
                //In case we have invalid data to create this user object
                //Cache instance
                getUsersCollection().put(someuser.getName(),someuser);
            }
            rset.close();
            rset=null;
            stmt.close();
            stmt=null;
        }
        catch (Throwable t){
            logAuthenticatorException(
                "DBAuthenticator: Error obtaining details for user "+name);
            logAuthenticatorException(t);
            t.printStackTrace();
        }
        finally {
            if (rset!=null){
                try {
                    rset.close();
                } catch (SQLException e) {
                }
                rset=null;
            }
            if (stmt!=null){
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt=null;
            }
        }
        return someuser;
    }
    private void LoadUsersFromDatabase(){
        Connection conn =null;
        java.sql.Statement stmt=null;
        java.sql.ResultSet rset=null;
        String name=null;
        try{
        conn=getDBConnection();
            stmt = conn.createStatement();
            rset = stmt.executeQuery(
                 "select name,password, rights,home from USERS order by name" );
            while(rset.next())
            {
               int rights= rset.getInt("RIGHTS");
               int home = rset.getInt("HOME"); // home desk association
               name=rset.getString("name").toLowerCase();
               String password=rset.getString("password");
               if (password==null || password.trim().length()==0 ||
```

```
                password.equals("null")) password="nopassword";
            String description="A "+name+" user";
            DBUser someuser=createDBUserInstance(name,description,password,
               rights,home);
            if (someuser==null) continue;
            getUsersCollection().put(someuser.getName(),someuser);
        }
        rset.close();
        rset=null;
        stmt.close();
        stmt=null;
    }
    catch (Throwable t){
        logAuthenticatorException("Error obtaining details for user "+name);
        logAuthenticatorException(t);
        t.printStackTrace();
    }
    finally {
        if (rset!=null){
            try {
                rset.close();
            } catch (SQLException e) {
            }
            rset=null;
        }
        if (stmt!=null){
            try {
                stmt.close();
            } catch (SQLException e) {
            }
            stmt=null;
        }
    }
}
public void reload() throws IOException {
    LoadUsersFromDatabase();
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
       "Reload called");
}
/**
 * Creates a new fPermission with the unique ID and name supplied.
 *
 * The implementation should save the new permission to the relevant
 * technology used.
 *
 * @param permNo Unique ID from 0 to 63.
 * @param name   Name describing this new permission.
 * @return the new fPermission.
 * @throws java.io.IOException If unable to create the new fPermission.
 */
public fPermission addPermission(int permNo, String name) throws
     IOException {
if(getPermissionsCollection().get(""+permNo) == null){
  fPermission perm = new fPermission();
  perm.reload(permNo,name);
  getPermissionsCollection().put(""+permNo, perm);
  return perm;
}
fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
    "Added Permission "+name+"("+permNo+")");
return (fPermission) super.getPermissionsCollection().get(""+permNo);
}
public fUser addUser(String username, String description,
```

```
     String plainPassword, String groupName) throws IOException {
fGroup group = null;
if (groupName != null) {
  group = (fGroup) getGroupsCollection().get(groupName);
  if (group == null) throw new IOException("No known group " + groupName);
}
fUser user = createUser(username, description, plainPassword, group);
getUsersCollection().put(user.getName(), user);
if(group != null) group.addUser(user);
fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+"Added User "+
     username+" NOTE: This is not currently persisted in the database!");
return user;
}
/**
 * Creates a new fUser with the supplied values.
 * The password field is passed as plain text but it is up to the
 * implementation to ensure the password
 * is secure.
 *
 * The implementation should save the new user to the relevant
 * technology used.
 *
 * @param user The user to copy.
 * @return The new fUser created.
 * @throws java.io.IOException If there where any errors during the
 * construction of the user.
 */
public fUser copyUser(fUser user) throws IOException {
    fGroup group = null;
    group = (fGroup) getGroupsCollection().get(user.getGroup().getName());
    fUser aUser = createUser(user.getName(),user.getDescription(),
       user.getPassword(),user.getGroup());
    getUsersCollection().put(aUser.getName(), aUser);
    if(group != null) group.addUser(aUser);
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
        "Copied User "+user.getName());
    return aUser;
}
/**
 * Adds a new group with the supplied values.
 *
 * The implementation should save the new group to the relevant
 * technology used.
 *
 * @param id          Unique ID for the group.
 * @param name        Name of the new group.
 * @param description Description of the new group.
 * @return The new fGroup object.
 * @throws java.io.IOException If unable to create the new fGroup object.
 */
public fGroup addGroup(int id, String name, String description) throws
      IOException {
    fGroup group = new fGroup();
    group.reload(id, name, description);
    addGroup(group);
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
        "Added Group "+group.getName());
    return group;
}
/**
 * Returns the permission with the ID supplied or null if not found.
 *
 * @param id fPermission Id to search for.
```

```
   * @return the fPermission or null if not found.
   */
  public fPermission getPermission(int id) {
    Enumeration perms = getPermissionsCollection().elements();
    while (perms.hasMoreElements()) {
      fPermission fPermission = (fPermission) perms.nextElement();
      if (fPermission.getId() == id) return fPermission;
    }
    return null;
  }
  /**
   * Returns the permission with the name supplied or null if not found.
   *
   * @param name fPermission name to search for.
   * @return the fPermission or null if not found.
   */
  public fPermission getPermission(String name) {
      return (fPermission)getPermissionsCollection().get(name);
  }
 public Enumeration getUsers(){
     return  getUsersCollection().elements();
 }
  public fUser getUser(String username) {
    return (fUser) LoadUserFromDatabase(username.toLowerCase());
  }
  public fGroup getGroup(String name) {
      return (fGroup)getGroupsCollection().get(name);
  }
  /**
   * Removes the permission with the ID supplied.
   *
   * The implementation should remove the permission from the relevant
   * technology used.
   *
   * @param id of the permission to delete.
   * @throws java.io.IOException if unable to delete the permission.
   */
  public void delPermission(int id) throws IOException {
  getPermissionsCollection().remove(""+id);
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
       "Deleted permission ("+id+")");
  }
  /**
   * Removes the user supplied.
   *
   * The implementation should remove the user from the relevant
   * technology used.
   *
   * @param user fUser object to remove.
   * @throws java.io.IOException If unable to remove the user.
   */
  public void delUser(fUser user) throws IOException {
  if (user.getGroup() != null) {
    user.getGroup().delUser(user);
  }
  getUsersCollection().remove(user.getName());
    fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
       "Deleted User "+user.getName());
  }
  /**
   * Removes the supplied fGroup object.
   *
   * Any user currently a member of this group will have the group reset
```

```
     * to null meaning no group membership.
     *
     * The implementation should remove the group from the relevant
     * technology used.
     *
     * @param group Group to remove.
     * @throws java.io.IOException If unable to remove the group.
     */
    public void delGroup(fGroup group) throws IOException {
        Enumeration enm = group.getUsers();
        while (enm.hasMoreElements()) {
          fUser user = (fUser) enm.nextElement();
          user.delGroup(group);
        }
        getGroupsCollection().remove(group.getName());
        fAuthenticator.logAuthenticatorMessage("{"+getName()+"} "+
            "Deleted Group "+group.getName());
    }
    /**
     * Requests that the implementation save the current state.
     *
     * This should include all users, groups and permissions.
     *
     * @throws java.io.IOException if the save failed.
     */
    public void saveState() throws IOException {
       //TODO: Implement saving of data to the database
    }
    public void roll() throws IOException {
        //TODO: Implement any log file rolling
    }
    protected fUser createUser(String name, String desc, String password,
        fGroup group){
      //  System.out.println("CreateUSer being called");
      DBUser usr = new DBUser();
      usr.reload(name, desc, password, group);
      usr.setHomeId(group.getName());
      usr.setAuthenticator(this);
      return usr;
    }
}
```

DBUser

```
package com.myapp;
import com.pcbsys.foundation.authentication.fUser;
import com.pcbsys.foundation.authentication.fGroup;
import com.pcbsys.foundation.authentication.fAuthenticator;
import java.util.Hashtable;
import java.sql.SQLException;
public class DBUser extends fUser {
  private static fAuthenticator myAuthenticator;
  protected DBUser(){
    super();
    super.setGroupHash(new Hashtable());
  }
  //Allow setting a reference to the authenticator instance so that we can
  // obtain its DB connection for
  //  user authentication purposes
  public static void setAuthenticator (fAuthenticator authenticator){
      myAuthenticator=authenticator;
  }
 protected DBUser(String name, String desc, String password){
```

```
    this(name, desc, password, null);
  }
  protected DBUser(String name, String desc, String password, fGroup group){
    super(name,desc,password,group);
  }
 public String login(byte[] password, boolean requestToken){
     return login(password,requestToken,null);
 }
public String login(String password, boolean requestToken){
    return login(password,requestToken,null);
}
 public String login(byte[] password, boolean requestToken, Hashtable params){
   return login(new String(password), requestToken, params);
  }
 public String login(String password, boolean requestToken, Hashtable params){
        java.sql.Connection conn =null;
        java.sql.Statement stmt=null;
        java.sql.ResultSet rset=null;
        String name=null;
        try{
        conn=((DBAuthenticator)myAuthenticator).getDBConnection();
        stmt = conn.createStatement();
        rset = stmt.executeQuery(
            "select password, rights from USERS where name ='"+
            this.getName()+"'" );
            while(rset.next())
            {
               int rights= rset.getInt("RIGHTS");
               String thepassword=rset.getString("password");
               if (thepassword.equals(password)){
                   if (rights > 0) return "true";
               }
            }
            rset.close();
            rset=null;
            stmt.close();
            stmt=null;
        }
        catch (Throwable t){
            myAuthenticator.logAuthenticatorException(
               "DBAuthenticator: Error obtaining details for user "+name);
            myAuthenticator.logAuthenticatorException(t);
        }
        finally {
            if (rset!=null){
                try {
                    rset.close();
                } catch (SQLException e) {
                }
                rset=null;
            }
            if (stmt!=null){
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt=null;
            }
        }
     return null;
  }
}
```

Web Application Single Sign On

Single sign-on (SSO) is a method of access control that enables a user to log in once and gain access to the required application and its resources without being prompted to log in again.When developing multi node Universal Messaging web applications, you have to take into consideration that incidents such a network failure, could cause the user's browser to fail over to a different node than the one initially authenticated with. In order to prevent the application user from authenticating again, or to integrate your Universal Messaging web application to a 3d party authentication mechanism and provide alternative authentication user interfaces, you can use Single Sign On Interceptors (SSI).

A Single Sign On Interceptor (SSI) is a class that conforms to a specific interface and gets invoked by the Universal Messaging realm prior to authentication in order to decide which of the following 3 outcomes should occur:

- If the user meets the criteria required, allow access to the plugin content as if they where normally authenticated, optionally generating a unique session id.
- If the user does not meet the criteria required, but a redirection is configured, redirect their browsers to the specified URL in order to authenticate.
- If the user does not meet the criteria required, and no redirection is configured, then fall back to the regular authenticator configured.

The interface for creation of a nirvana SSI implementations is defined in the following 2 classes of the com.pcbsys.foundation.authentication package:

fSSInterceptor

```
//Return an fSSUser with a null username to fall back to authenticator
// (or redirect if a URL is set)
public abstract fSSUser getSSUser(Hashtable httpHeaders,
       Hashtable urlParameters);
public abstract void setParameters(Hashtable params);
public abstract void clear();
public abstract String getName();
```

fSSUser

```
public fSSUser(String username, String redirectURL);
public fSSUser(String username, String redirectURL, String token);
public String getUsername();
public String getRedirectURL();
public String getToken()
```

Plugin Single Sign On Interceptor Parameters

- SSInterceptor: Fully qualified class name of SSI to use. If not specified, no interceptor will be used
- SSOAppendToken: Setting this parameter to true instructs the SSI object to generate and return a unique session ID when an affirmative single sign on decision is reached. Please note that the absence of a session ID is irrelevant to the single sign on decision.

Common Single Sign On AuthParameters

Irrespective of the Single Sign On implementation you use in your Universal Messaging server plugins, there are some AuthParameters that are also used by the server. These are:

- `SSONamedInstance`: This optional parameter requests that this SSI object is bound to the specified named instance which will be shared across all plugins on this server that are configured to do so. Please note that the first plugin that accepts a connection will bind the name to the server together with the remaining configuration parameters. For this reason please make sure that configuration is always the same on all plugins that share the same SSI instance.
- `REDIRECT_URL`: This optional parameter specifies the URL that a web client should be redirected to should the interceptor's criteria are not met. This allows the creation of alternative authentication methods such as form based authentication or others.

Mobile Application Authentication

When developing Universal Messaging based mobile applications, authentication is dependent on your mobile technology of choice. This is because the Universal Messaging Mobile APIs work exactly like any Universal Messaging enterprise or web API (with the exception of SSL client certificates).

Using SASL

Overview

The entire set of session creation methods of the Universal Messaging client API for Java (nsp/ nsps/ nhp/ nhps, native and JMS) have been extended in Universal Messaging 9.6 with overloaded variants that accept username/password credentials which are then supplied to the UM server, and the UM server has been enhanced to enable those credentials to be authenticated against pluggable Directory backends, ranging from LDAP to flat files.

The exchange of user credentials can be performed by means of either SASL (embedded within the Universal Messaging SDK) or JAAS (controlled by user-configurable pluggable modules).

The configuration is determined by a set of Java system properties on both the client and server side, the latter typically centralized in the nserver.conf configuration file.

Note that authentication does not supplant the traditional Universal Messaging ACLs and is merely an additional security step performed before the relevant ACLs are evaluated and applied.

Client

If the pre-existing session connection methods with no username/password parameters are used, then the client will continue to use unauthenticated sessions as before

(assuming the server is configured to allow that), i.e. by defaulting the user identity to the username under whose identity the client process is running (as reflected in the Java "user.name" system property). The client API is controlled by two main Java system properties.

- *Nirvana.auth.client.jaaskey*

  If set, this means that any authentication should be performed via JAAS and its value specifies the name of the entry to use in the JAAS login configuration file. JAAS is an established Java standard which is beyond the scope of the UM documentation and the login configuration file's pathname is specified by the usual JAAS system property, java.security.auth.login.config. The Nirvana client SDK supplies the username and password to the JDK's built-in JAAS subsystem, and the JAAS login config file specifies one or more pluggable JAAS modules that will perform the authentication. The precise modules to use are a matter of site-specific policies determined by the Nirvana admins, and the JAAS modules configured into a client should obviously be aligned with those configured on the server. If Nirvana.auth.client.jaaskey is not set, then the authentication mechanism defaults to SASL.

- *nirvana.sasl.client.mech*

  This specifies which SASL mechanism to use, and the supported options are Plain or CRAM-MD5. The mechanism defaults to Plain if this system property is not set, and the usual SASL trade-offs apply. SASL-Plain transmits the user password in cleartext, so it is advisable to only use it over an SSL connection, while Cram-MD5 does not transmit it in cleartext, but does require it to be stored in plaintext on the server (whereas Plain can work with stored passwords in either plain or encrypted format).

One of the JAAS modules available is the Nirvana class, com.pcbsys.foundation.security.sasl.fSaslClientLoginModule, which will result in the authentication being performed via SASL after all, despite initially being routed via JAAS. From the server's perspective, the authentication negotiation is conducted entirely in SASL. The fSaslClientLoginModule class is integrated with the Software AG family of JAAS modules, and one reason you might opt for this JAAS-SASL hybrid is to chain it with other SAG modules.

Server

There is a much broader range of configuration options on the server, controlling every aspect of authentication from whether it's enabled in the first place, to how to look up user credentials.

Client Negotiation

Authentication is disabled by default on the server for backward compatibility, meaning that even if clients do supply user credentials, they will be accepted without verification. This is controlled by the `Nirvana.auth.enabled` system property, which must be explicitly set to "Y" or "y" to enable authentication. System admins have to set up various other config options when enabling authentication, so they

would set Nirvana.auth.enabled as part of that effort. Even when authentication is enabled, authenticating clients can exist side-by-side with non-authenticating ones, meaning it is entirely optional for clients to supply any user credentials, and if they don't they will be handled in the traditional Universal Messaging manner. The `Nirvana.auth.mandatory` system property controls this behaviour, and should be explicitly set to "Y" or "y" to make authentication mandatory, meaning clients that don't supply a username and password will be rejected (with the exception of the super-user on localhost, so that Enterprise Manager doesn't get locked out). When a client does authenticate, the UM client-server protocol automatically signals the server whether they're using SASL or JAAS and if JAAS, then the `Nirvana.auth.server.jaaskey` system property must be set on the server, and it specifies the name of the entry to use in the JAAS login configuration file. As in the client case, the pathname of this file is specified by the standard JAAS `java.security.auth.login.config` property. If the `Nirvana.auth.server.jaaskey` system property is not set, then all attempts to authenticate via JAAS will be rejected.

Directory Backend

The UM server can make use of a variety of backend Directory servers or mechanisms, as controlled by the `Nirvana.directory.provider` system property, which specifies the pluggable Java class representing the Directory.

Usernames are case-sensitive and are used in the form supplied to do the Directory lookup. This is the authentication step, and is followed by an authorization step in which the username is normalized to lowercase to match against Nirvana ACLs. Nirvana ACLs are case-insensitive but expressed in lower-case and any ACLs created via the Enterprise Manager will be forced to lower case.

Internal User Repository

If the Nirvana.directory.provider system property is set to `com.pcbsys.foundation.security.auth.fSAGInternalUserRepositoryAdapter`, then usernames will be looked up in a standard Software AG store called the 'Internal User Repository', which is a flat file maintained by the SAG command-line utility internaluserrepo.sh (found in INSTALLROOT/common/bin).

This mechanism is the default Directory, if the Nirvana.directory.provider property is not set.

The location of the file is given by the system property, `Nirvana.auth.sagrepo.path`, and would default to ./users.txt (relative to runtime directory of UM server), but the nserver.conf shipped with UM overrides this as ../users.txt, locating it in the same INSTALLROOT/nirvana/server/umserver directory as the licence.xml file. The nserver.conf file may of course be edited as usual to move the users.txt file into a location that is shared by all the realms of an installed UM instance.

LDAP

If the `Nirvana.directory.provider` system property is set to `com.pcbsys.foundation.security.auth.fLDAPAdapter`, then LDAP will be used as the source of user information.

Interaction with the LDAP server is configured via the following Java system properties:

- `Nirvana.ldap.provider`: The LDAP client class - defaults to the JDK's built-in provider, com.sun.jndi.ldap.LdapCtxFactory
- `Nirvana.ldap.url` : The address of the LDAP server. This has no default and must be specified, using syntax such as ldap://localhost:389/dc=sag,dc=com
- `Nirvana.ldap.suffix` : The suffix to apply to LDAP queries. This has no default and may be null, but if non-null it qualifies the URL above. Eg. Nirvana.ldap.url=ldap://localhost:389/dc=sag and Nirvana.ldap.suffix=dc=com will result in the same effective query root as Nirvana.ldap.suffix=ldap://localhost:389/dc=sag,dc=com when the `Nirvana.ldap.suffix` property is not set.
- `Nirvana.ldap.rootcreds`: The privileged-admin login credentials to use on the LDAP server, in order to perform user queries. There is no default and if not set it means there is no need to specify any such credentials, but if present the format must be username:password.

The remaining system properties relate to the LDAP schema and default to the standard COSINE schema:

- `Nirvana.ldap.attribute.username`: This specifies the LDAP attribute which represents the username, and defaults to the standard schema convention of "cn".
- `Nirvana.ldap.attribute.password`: This specifies the LDAP attribute which represents the password, and defaults to the standard schema convention of "userPassword".
- `Nirvana.ldap.search.username`: This specifies the search expression to use for a given username, and defaults to cn=%U%, where %U% gets substituted by the username.

Converting a .jks Key Store to a .pem Key Store

In order to convert a Java key store into a Privacy Enhanced Mail Certificate, you will need to use two tools :

1. keytool.exe - to import the keystore from JKS to PKCS12 (supplied with Java)
2. openssl.exe - to convert the PCKS12 to PEM (supplied with OpenSSL)

Neither keytool or openssl can be used to convert a jks directly into a pem. First we must use keytool to convert the JKS into PKCS:

```
keytool -importkeystore -srckeystore client.jks -destkeystore client.pkcs
  -srcstoretype JKS -deststoretype PKCS12
```

You will be prompted to enter passwords for the key stores when each of these programs are run. The password used for the keystores created using the generator is "nirvana". Next you need to use openssl.exe to convert the PKCS into PEM.

```
openssl pkcs12 -in client.pkcs -out client.pem
```

Repeat the above code for the any other jks key stores. After this you will have the required key stores in pem format.

Access Control Lists

Security Policies

Universal Messaging offers complete control over security policies. Universal Messaging can either store security policies locally or be driven by an external entitlements service.

Universal Messaging's rich set of entitlements ensure that everything from a network connection through to topic and queue creation can be controlled on a per user basis.

Every component of a Universal Messaging server has a set of entitlements associated with it. These entitlements can be set programmatically or through the Universal Messaging Enterprise Manager.

For more information on the components that entitlements can be set against please refer to the Universal Messaging ACL Guide (see "Access Control Lists (ACLs)" on page 135).

Access Control Lists (ACLs)

Universal Messaging's Access Control List (ACL) controls client connection requests and subsequent Universal Messaging operations. By default access control checks are performed within a realm.

The Universal Messaging Administration API exposes the complete security model of the Universal Messaging Realm Server, remotely allowing customer specific security models to be created. This means that it is easy to integrate Universal Messaging into an existing authentication and entitlement service.

It is also possible to manage Universal Messaging ACLs using the enterprise manager GUI.

The Universal Messaging realm has an ACL associated with it. The acl contains a list of subjects and the operations that each subject can perform on the realm.

Users are given entitlements based on their subject. A subject is made up of a username and a host.

The username part of the subject is the name of the user taken from either the operating system of the machine they are connecting from or the certificate name if they are using an ssl protocol.

The host part of the subject is either the ip address or the hostname of the machine they are connecting from.

The subject takes the form of :

```
username@host
```

For example:

```
johnsmith@192.168.1.2
```

Each channel, queue and service also has an associated acl that defines subjects and the operations the subjects can perform.

A subject corresponds to the user information for a realm connection

Each type of acl entry has a number of flags that can be set to true or false in order to specify whether the subject can or can't perform the operation.

General ACL permissions

The following flags apply to every ACL.

- Modify - Allows the subject to add/remove ACL entries
- List - Allows the subject to get a list of ACL entries
- Full Privileges - Has complete access to the secured object

Universal Messaging Realm Server ACL permissions

The Realm Access Control Entry has the following controllable flags

- Use Admin API - Can use the nAdminAPI package
- Manage Realm - Can add / remove realms from this realm
- Manage Joins - Can add/delete channel joins
- Manage P2P Services - Can create/destroy P2P services
- Manage Channels - Can add/delete channels on this realm
- Access The Realm - Can actually connect to this realm
- Override Connection Count - Can bypass the connection count on the realm
- Configure Realm - Can set run time parameters on the realm
- Cluster - perform cluster operations, such as create, delete or modify cluster information
- Manage DataGroups - Can add, delete and manage any DataGroup on the realm
- Publish to DataGroups - Can publish to any DataGroup on the realm
- Own DataGroups - Can take ownership of any DataGroup on the realm

Channel ACL permissions

The Channel Access Control Entry has the following controllable flags

- Write - Can publish events to this channel
- Read - Can subscribe for events on this channel
- Purge - Can delete events on this channel

- Get Last EID - Can get the last event Id on this channel
- Named - Can the user connect using a named (durable) subscriber

Queue ACL permissions

The Queue Access Control Entry has the following controllable flags

- Write - Can push events to this queue
- Read - Can peek the events on this queue
- Purge - Can delete events on this queue
- Pop - Can pop events from the queue

P2P Service permissions

The Service Access Control Entry has the following controllable flags

- Connect - Can access this service

Wildcard Support

As well as being able to specify an access control entry for a specific subject the subject itself can contain wildcards. In this way you can specify access control based on hostname or on username.

The subject *@* is provided in all ACL objects by default, and corresponds to the default permission that all subjects inherit who connect but do not individually appear within the ACL. If a subject is listed in the ACL, then the entitlements given to that subject overrides that of any wildcarded entry, including the *@* default subject.

Example Wildcard ACLs :

ACL Entry	Description
@	Represents all users from all nodes
*@client1.com	Represents all users from the node client1.com
username@nodename	Represents the user "username" on the node "nodename"
username@*	Represents the user "username" on all nodes

SSL

SSL Encryption

Universal Messaging fully supports SSL and offers support for SSL enabled TCP/IP sockets as well as HTTPS. When SSL is used the subject used for entitlements can be extracted from the clients certificate CN attribute.

Universal Messaging's SSL implementation supports both client side and anonymous (Server side) SSL. Different SSL certificate chains can be assigned to different Universal Messaging interfaces, each one supporting its own set of cryptographic algorithms. There is no limit to the number of interfaces and therefore different SSL certificate chains that can be supported on a single Universal Messaging Realm server.

For more information on configuring Universal Messaging interfaces to use SSL encryption with Universal Messaging please see the Enterprise Manager guide.

To learn more about SSL please see the SSL Concepts section ("SSL Concepts" on page 140).

Client SSL Configuration

This Section describes how to use SSL in your Universal Messaging client applications. Universal Messaging supports various wire protocols (see "Communication Protocols and RNAMEs" on page 22) including SSL enabled sockets and HTTPS. The example programs contained in the Universal Messaging package will all work with SSL enabled on the realm server.

This guide explains client SSL configuration on Java. Guides for C++ and C# are also available.

Once you have created an SSL enabled interface for your realm you need to ensure that your client 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.

If you would like to add your own client certificates please see our developers guide.

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

Using the example keystores, the following system properties are required by the Universal Messaging sample apps and must be specified in the command line as follows:

```
-DCKEYSTORE=%INSTALLDIR%\client\Universal Messaging\bin\client.jks
-DCKEYSTOREPASSWD=password
-DCAKEYSTORE=%INSTALLDIR%\client\Universal Messaging\bin\nirvanacacerts.jks
-DCAKEYSTOREPASSWD=password
```

where:

CKEYSTORE is the client keystore location

CKEYSTOREPASSWD is the password for the client keystore

CAKEYSTORE is the CA keystore file location

CAKEYSTOREPASSWD is password for the CA keystore

The above system properties are used by the Universal Messaging sample apps, but are mapped to system properties required by a jsse enabled JVM by the utility program 'com.pcbsys.foundation.utils.fEnvironment', which all sample applications use. If you do not wish to use this program to perform the mapping between Universal Messaging system properties and those required by the JVM, you can specify the SSL properties directly. To do this in your own applications, the following system properties must be set:

```
-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, both the Universal Messaging sample apps and your own 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 (see "Communication Protocols and RNAMEs" on page 22) used by your client application must correspond to the correct type of SSL interface, and the correct hostname and port that was configured earlier.

JMS Clients

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.

SSL Concepts

SSL With Client Certificate Validation

When a client requests a secure connection to a server, both the client and the server must verify each other's certificate to ensure that the source is trusted. A certificate will generally contain the source's name, the certificate authority that signed it and the public key for that source. The certificate can be authenticated by validating the signature.

- *Certificate signed by well known certification authority*

  Several certification authorities exist (such as Verisign) that will review applications for certificates and digitally sign them if they have sufficient proof that the company is what it claims to be. These authorities are trusted as default by web browsers and applications supporting SSL.

  When a connection attempt is made, the certificate is checked with these well known authorities. The signature is created using the private key of the certificate authority, therefore it is known to have been encrypted by that authority only if it can be decrypted by it's public key. If this is the case then it is known that the certificate is valid.

- *Self signed certificates*

  To acquire a certificate from a well known certification authority is expensive and time consuming as the company will have to prove itself by providing certain documentation. It is possible however to create and sign your own certificates. This essentially makes the company that created the certificate also act as the certification authority for that certificate. This means that the certificate will not be validated by checking with the well known authorities.

  If the certificate is not validated by the well known authorities then it is checked against the user created trust store. This store contains certificates for companies that are not registered with the well known authorities but the user has chosen to trust.

  Because the sample certificates created are self signed, it is important that the trust store of the client/server contains the certificates of the server/client respectively otherwise an exception will be thrown as the system cannot verify the source.

Once the certificates have been validated, the client and server will have each others public keys. When the client communicates with the server, it will encrypt the data with the server's public key. Public keys are asymmetric which means that it cannot be decrypted using the same key, instead it requires the server's private key which only the server holds. This means that only the server can read the data sent. Similarly, any response from the server will be encrypted with the client's public key.

Anonymous SSL

Universal Messaging also supports anonymous (server-side) SSL. Anonymous SSL does not validate the client. This means that the client does not need to have a certificate as it is never checked by the server. Instead, the client sends a request for the server's

certificate as usual but instead of the server encrypting data sent to the client with the client's public key, a session is created. To create a session, the client generates a random number and sends this number to the server encrypted with the server's public key. Only the server can decrypt the random number. So only the server and the client know this number. The random number is used along with the server's public key to create the session.

6 MQTT: An Overview

MQTT (MQ Telemetry Transport), is a publish/subscribe, simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth by IBM / Eurotech on 1999. The simplicity and low overhead of the protocol make it ideal for the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium. The protocol is openly published with a royalty-free license and a variety of client libraries have been developed especially on popular embedded hardware platforms such as arduino/netduino, mbed and Nanode.

In addition to Universal Messaging's own protocol, NSP interfaces are capable of also accepting MQTT connections over TCP sockets, while NSPS interfaces can accept MQTT connections over SSL/TLS for client implementations that support it.

Connecting

In order to connect to a Universal Messaging server using MQTT, you application needs to use a tcp://host:port URL (NSP Interfaces) or ssl://host:port URL (NSPS Interfaces). MQTT connections are treated the same way as any other connections by the Universal Messaging realm. The MQTT client ID value is combined with the optional username (or anonymous if no username is specified) in order to define a nirvana subject as follows:

Client ID_Username@hostname

protocol	user	host	connection	language
nsp	krital	localhost	127.0.0.1:56777	Java 1.6.0_29
mqtt	mqtt_utility_anonymous	localhost	127.0.0.1:56768	MQTT 3

Figure 1: Connection List with an MQTT connection

This way you can define realm and channel / queue ACLs as you would for any Universal Messaging connection. For example using the IBM WMQTT sample application with a client identifier of MQTT_Utility and no username/password to connect to tcp://localhost:1883 will result in a mqtt_utility_anonymous@localhost Universal Messaging subject.

Publishing

MQTT applications can publish events to channels or queues that have already been created in the Universal Messaging realm. These are regular Universal Messaging channels / queues that can utilise different reliability types (transient, simple, reliable, persistent, mixed) to enforce a persistence policy on all events. Events published via MQTT only contain a byte[] payload and are tagged MQTT. They are fully interoperable with any Universal Messaging subscriber on any client platform supported and can be snooped using the Universal Messaging Enterprise Manager:

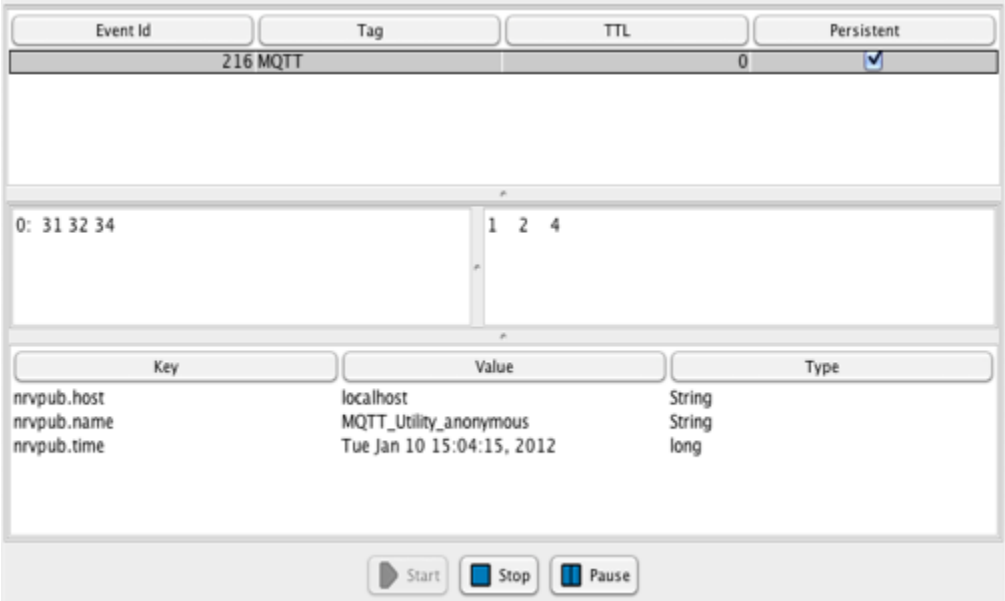


Figure 2: Snooping an MQTT message

Subscribing

MQTT applications can subscribe to channels or queues that have already been created in the Universal Messaging realm. These are regular Universal Messaging channels / queues of any reliability type offered and can contain messages published by either MQTT or Universal Messaging applications. Please note that if subscribed to a queue, the pop operation is destructive so each subscriber will get a single event in round robin while channels offer regular pub/sub behavior.

Quality of Service

Universal Messaging supports all three QOS levels defined by the MQTT standard. This is driven by the MQTT client connection and describes the effort the server and client will make to ensure that a message is received as follows:

1. QOS 0: The Universal Messaging realm will deliver the message once with no confirmation
2. QOS 1: The Universal Messaging realm will deliver the message at least once, with confirmation required.
3. QOS 2: The Universal Messaging realm will deliver the message exactly once, using a 4 phase approach as defined by the standard.

Will

Universal Messaging fully supports connections with Will settings, which indicate messages that need to be published automatically if the MQTT application disconnects unexpectedly.

Virtual Payload Types

Universal Messaging now provides MQTT users with the ability to define namespace roots where advanced namespace semantics can be applied. Specifically, the last part of the full topic namespace address can be used to indicate your preferred virtual payload type when publishing or subscribing to a topic. The currently supported types are:

```
-1: UnspecifiedType (default), binary, equivalent to no type indicator
 0: BaseMessageType, Unsupported
 1: MapMessageType, UTF-8 JSON on MQTT, JMS MapMessage on native UM
 2: BytesMessageType, binary, but identified as JMS BytesMessage
 3: ObjectMessageType, java serialized bytes identified as JMS ObjectMessage
 4: StreamMessageType, binary on MQTT, JMS StreamMessage on native UM
 5: TextMessageType, UTF-8 string on MQTT, JMS TextMessage on native UM
```

7 Commonly Used Features

Overview

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.

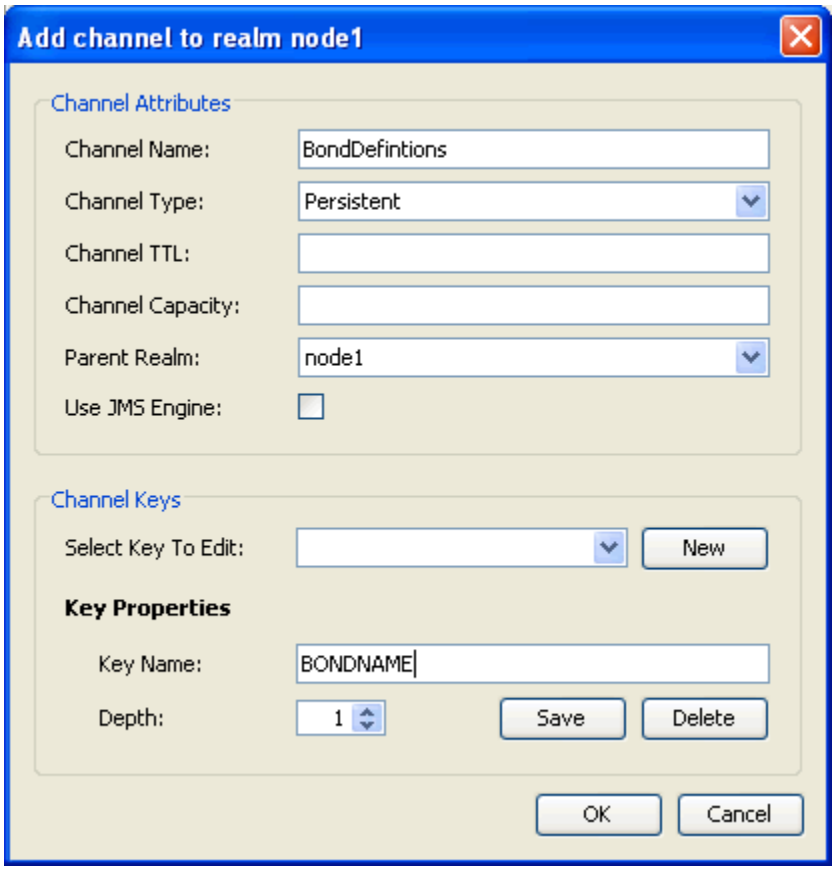
Channel Keys allow a channel or queue to automatically purge old events when new events of the same "type" are received. Two events are of the same "type" if the value in their dictionary (*nEnventAttributes*) for the key defined as the Channel Key are identical. The channel will store the specified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

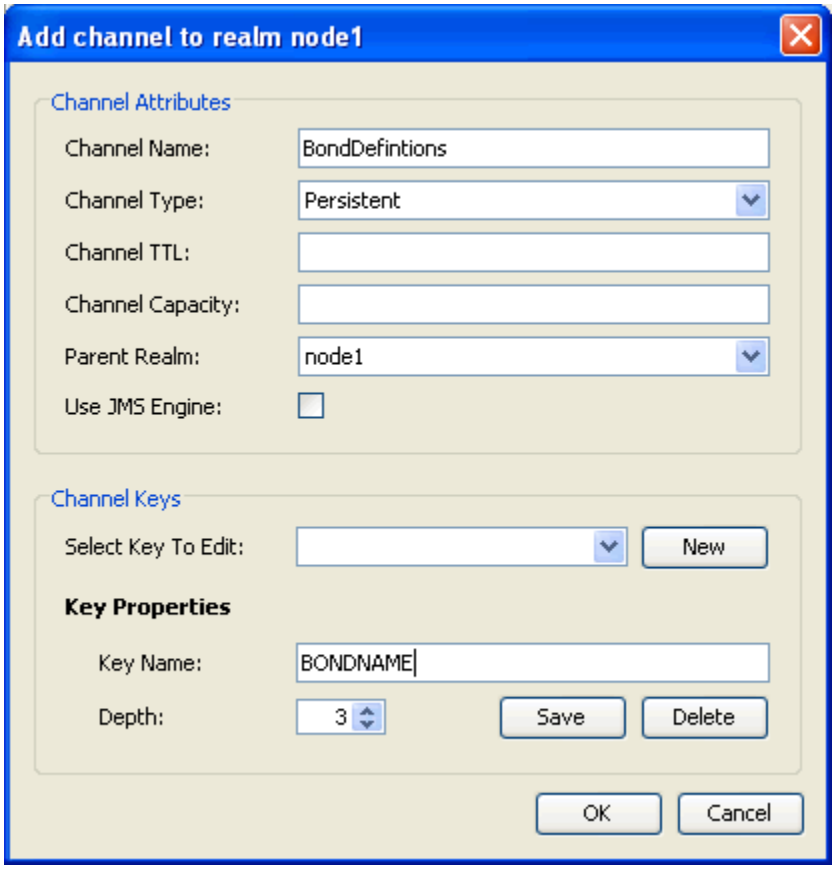


Note that in addition to creating a channel with the Name "BondDefinitions" and a type of "Reliable", the channel also has a Channel Key called BONDNAME with a depth of 1. The channel key defines the key in the nEventProperties which identifies events as being of the same type if their value for this 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

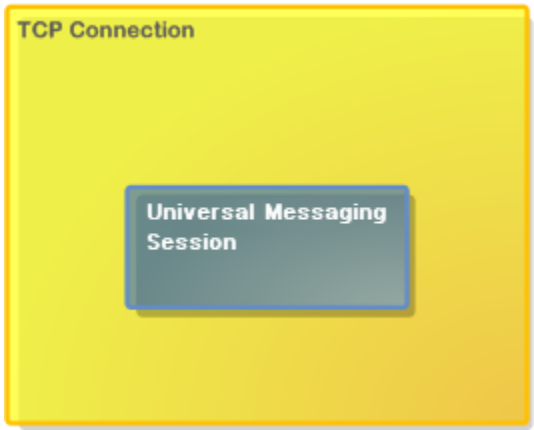
Universal Messaging supports four *Native Communication Protocols*. These TCP protocols are:

- 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)

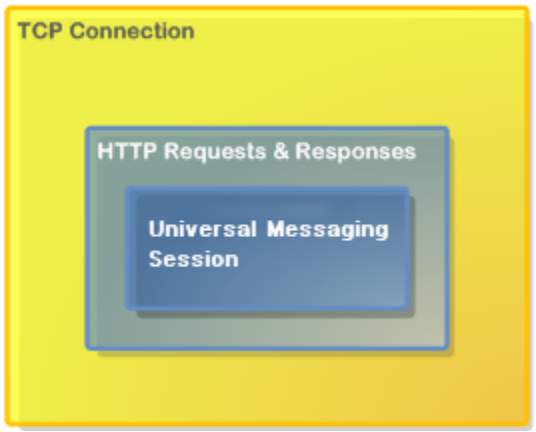
Universal Messaging HTTP Protocol (nhp)

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.

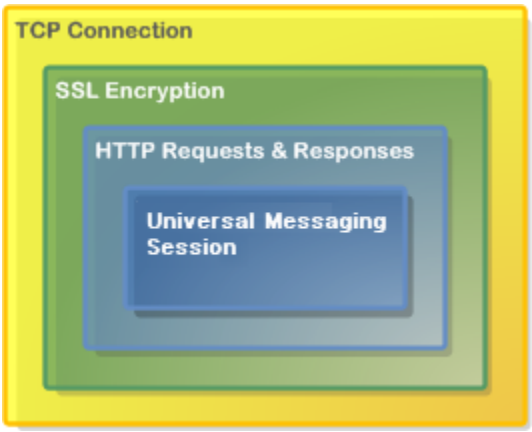
Communication Protocols

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.

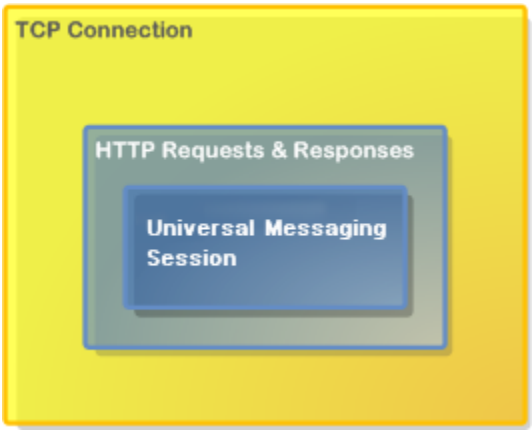


HTTPS Protocol (https)

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 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 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.

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).

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 163).

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 the 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