# Universal Messaging Concepts

Version 10.7

October 2020

This document applies to Software AG Universal Messaging 10.7 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.

Copyright © 2013-2023 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, USA, and/or its subsidiaries and/or its affiliates and/or their licensors.

The name Software AG and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA Inc. and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Detailed information on trademarks and patents owned by Software AG and/or its subsidiaries is located at https://softwareag.com/licenses/.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices, license terms, additional rights or restrictions, please refer to "License Texts, Copyright Notices and Disclaimers of Third Party Products". For certain specific third-party license restrictions, please refer to section E of the Legal Notices available under "License Terms and Conditions for Use of Software AG Products / Copyright and Trademark Notices of Software AG Products". These documents are part of the product documentation, located at https://softwareag.com/licenses/ and/or in the root installation directory of the licensed product(s).

**Document ID: NUM-CO-107-20230713**

# Table of Contents

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

**1 Architecture** ..... **7**
Architecture Overview ..... 8
Realms ..... 9
Zones ..... 10
umTransport API ..... 15
Communication Protocols and RNAMEs ..... 22
Support for Open Standards ..... 24

**2 Management** ..... **25**
Administration and Management ..... 26
JMX Administration and Management ..... 27
JMX User Guide ..... 34

**3 Deployment** ..... **39**
Deployment ..... 40
Bi-directional Client and Server Compatibility ..... 40
Server ..... 42
Client ..... 61
Language Deployment Tips ..... 69

**4 Security** ..... **71**
Security Overview ..... 72
Authentication ..... 72
Access Control Lists ..... 99
Using SSL ..... 102
Setting up and using FIPS ..... 109

**5 Performance, Scalability and Resilience** ..... **115**
Overview of Performance, Scalabilty and Resilience ..... 116
Performance Tuning ..... 117
Multicast: An Overview ..... 126
Shared Memory (SHM) ..... 129
Realm Benchmarks ..... 129
Failover ..... 134
Connections Scalability ..... 135
Load Balancing ..... 135
Horizontal Scalability ..... 137
Event Fragmentation ..... 143

**6 Messaging Paradigms....145**
Overview....146
Publish / Subscribe (using Channels/Topics)....146
Publish / Subscribe (using Data Groups and Data Streams)....146
Message Queues....150
Details of Usage for Software Developers....150

**7 Durable Subscriptions....153**
Overview of Durable Subscriptions....154
Types of Durable Subscription....155
Using Durable Subscriptions with Multiple Clients....160
Using the Prefetch API....162
Durable Event Browsing and Purging....165

**8 Clustered Server Concepts....167**
Clusters: An Overview....168
Active/Active Clustering....172
Active/Passive Clustering....192
Active/Active Clustering with Sites....195

**9 MQTT: An Overview....205**

**10 AMQP....209**
Overview of AMQP....210
AMQP Guide....216

**11 Product Usage Metrics....269**

**12 Commonly Used Features....271**
Overview....272
Sessions....272
Channel Attributes....272
Storage Properties of Channels and Queues....277
Channel Publish Keys....279
Queue Attributes....281
Multi-File Disk Stores....282
Events....288
Native Communication Protocols....301
Comet Communication Protocols....304
Google Protocol Buffers....306
Using the Shared Memory Protocol....308
Rollback....309
Out-of-Memory Protection....309
Pause Publishing....311
Priority Messaging....313

# About this Documentation


- Online Information and Support ..... 6
- Data Protection ..... 6

# Online Information and Support

### Software AG Documentation Website

You can find documentation on the Software AG Documentation website at http://documentation.softwareag.com.

### Software AG Empower Product Support Website

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

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

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

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

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

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

### Software AG TECHcommunity

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

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

# Data Protection

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

# 1 Architecture


- Architecture Overview ..... 8
- Realms ..... 9
- Zones ..... 10
- umTransport API ..... 15
- Communication Protocols and RNAMEs ..... 22
- Support for Open Standards ..... 24

# 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 *umTransport* 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 has 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 and Queues functionality are 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++ running on Windows, Solaris and Linux are all supported. Mobile devices and Web technologies 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 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, ACLs, queues etc.) and query management data (throughput, cluster state, numbers of connections etc.) directly from Universal Messaging realm servers.

### umTransport Communication

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 (currently available for Java and C++). See the section "umTransport API" on page 15 for a description of the umTransport model.

## 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 or Message Queues.

Universal Messaging provides the ability to create clusters of realms that share resources (see "Messaging Paradigms" on page 145) within the namespace. Cluster objects can be created, deleted and accessed programmatically or through the Universal Messaging Enterprise Manager.

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 and resilience.

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 45) where realms may be in different physical location, but accessible through one physical namespace.

# Zones

## Overview of Zones

Zones provide a logical grouping of one or more Realms which maintain active connections to each other. Realms can be a member of zero or one zone, but a realm cannot be a member of more than one zone . Realms within the same zone will forward published channel messages to other members of the same zone, if there is necessary interest on corresponding nodes.

**Note:**
The forwarding of messages between realms in a zone applies only to messages on channels, not to messages on queues. Messages on queues are not forwarded between realms in a zone.

Clusters can also be members of zones. In this case, the cluster as a whole is treated as a single member of a zone. Members of a zone should be well-connected, i.e., always connected to all other zone members, however this is not enforced.

Zones are uniquely identifiable, and two disparate zones with the same name can co-exist.

### Interest Propagation

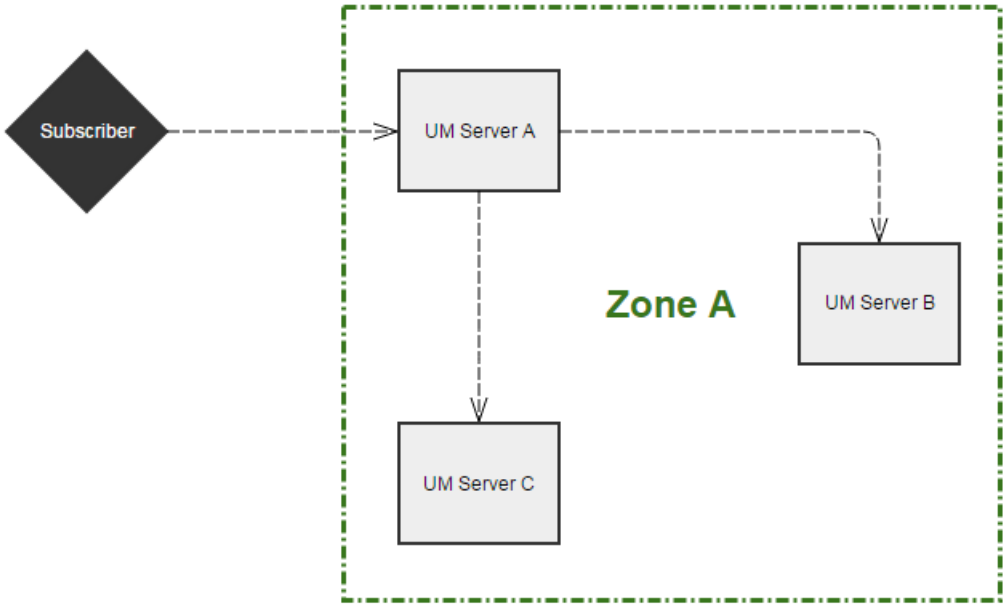
Using zones, Universal Messaging can deliver channel messages between realms and clusters when interest from one realm or cluster has been registered on another realm or cluster in the same zone. Here the term "interest" means that the realm wishes to receive a copy of messages that are sent originally to a channel of the same name on another realm within the same zone. This "interest" only arises on a realm when a client subscribes to a channel on the realm; if there is no such client on the realm, there is no "interested" client, so no interest propagation occurs.

A realm will forward its own interest in a particular channel to all realms directly connected to it within the same zone. When a realm receives a notification of interest from another realm within the same zone it will add an entry within its interest manager and begin to directly forward messages to this realm.

The mechanism within any given zone is as follows. The description assumes that there are several realms in the zone; we'll refer to the realm you are working on as the source realm and all the other realms as remote realms:

- When a client subscribes to a channel on the source realm, the source realm indicates to the remote realms that it wishes to receive any messages sent to the channel on the remote realms. In other words, the source realm propagates interest for this channel to the remote realms.
- Once interest from the source is registered on the remote realms, messages published to the channel on any of the remote realms in the zone will be propagated to the source realm.

### Example: Propagating Interest within a Zone

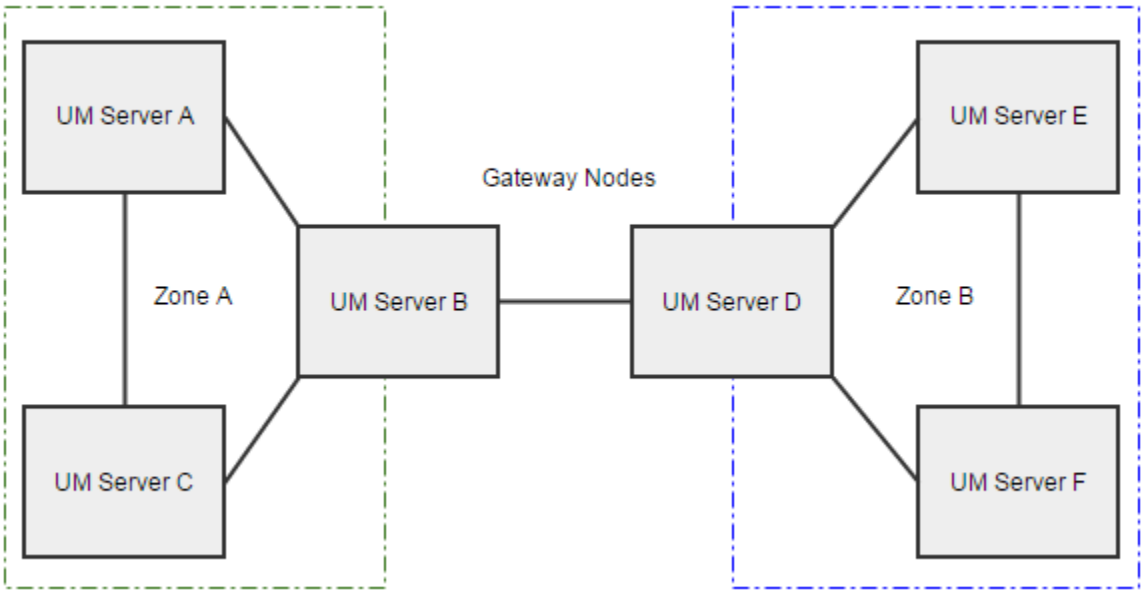



In the example above we see that a client has subscribed to a channel on realm A. The Interest Manager for realm A forwards a notification to all other realms within the same zone (realm B and realm C). Realm B and realm C now know that they should begin to forward messages they receive for this channel to realm A.

As both realm B and realm C are members of the same zone they do not propagate this interest any further. Therefore realm B knows that realm C has no local subscriber and will not forward messages to realm C. Similarly, realm C will not forward messages to realm B.

### Propagating Interest between Zones

You can propagate interest between zones using by manually creating static joins, for example:




**Note:**
Currently, a realm cannot be a member of more than one zone, so you cannot define a realm that is common to two zones. Therefore interest propagation between two zones using a common realm is not supported.

## Administration of Zones

You can perform the following functions using both the Administration API and the Enterprise Manager:

- Create a new zone
- Add a realm to a zone
- Remove a realm from a zone
- List the zone which a realm currently belongs to

### Administration of Zones using the Administrative API

The Administration API provides classes for performing the required administrative functions. These classes are summarized in the following outline example:

```
public class nRealmNode {
    ....
    public nZoneManager getZoneManager();
    ....
}
public class nZoneManager {
    ...
    public static nZone createZone(String zoneName);
```

```
    public boolean joinZone(nZone zone);
    public boolean leaveZone(nZone zone);
    public nZone getZone();
    ...
}
```

### Administration of Zones using the Enterprise Manager

The Enterprise Manager provides menu items for performing the required administrative functions.

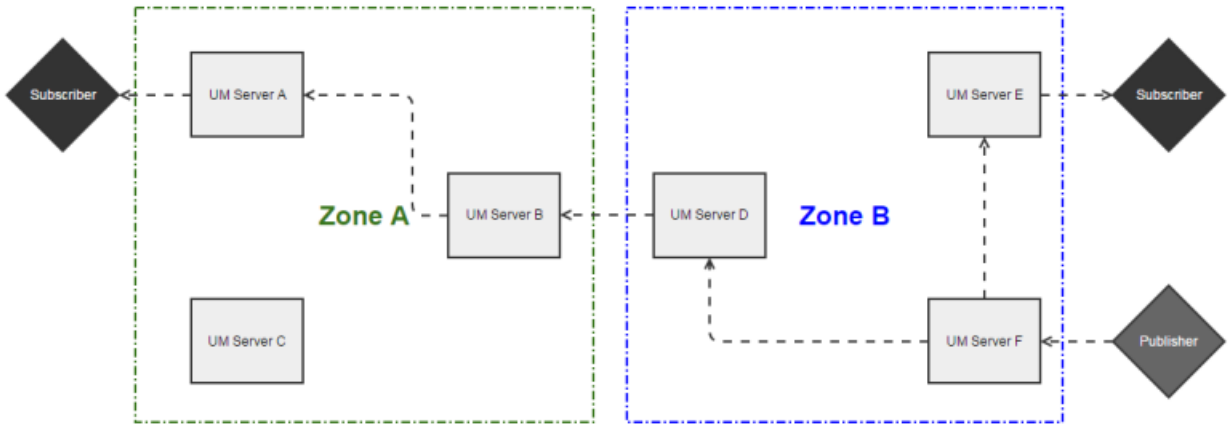
When you select a realm in the Enterprise Manager, you can perform the following functions within the context of the realm:

- Specify that the realm is a member of the given zone. If the zone does not exist already, it is created as part of this process. Currently, a realm cannot be a member of more than one zone.
- Remove the realm from a zone. If there are no more realms in the zone, the zone continues to exist, but its functionality is deactivated.
- Check which zone the realm belongs to.

See the documentation of the Enterprise Manager for details.

## Examples of Message Propagation

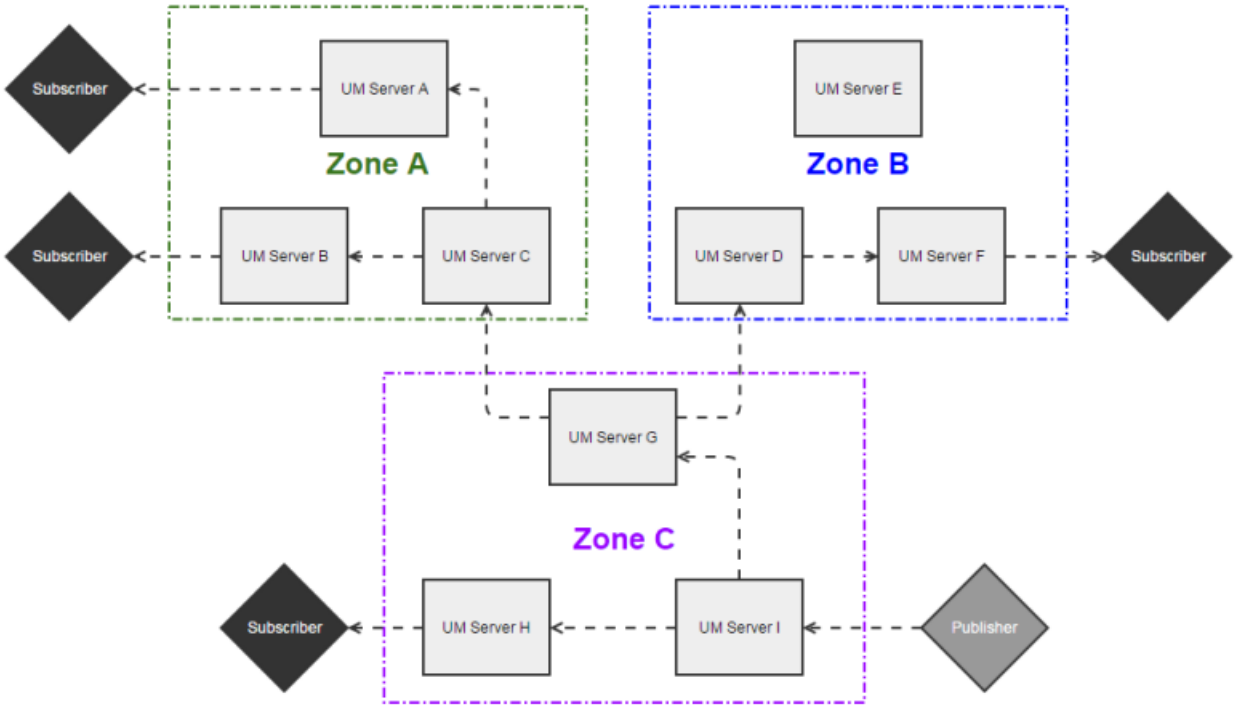
### Example 1




1. A message on a given channel is published to Realm F in Zone B.
2. Servers D and E in the same zone have registered interest in the channel, so Server F forwards the message to these servers.
3. Realm E has a local subscriber, so server E delivers the message to the local subscriber. Realm E has no static joins so will not forward the message any further.
4. Realm D has a static join to Realm B. It will forward this message over the static join to Realm B.

5. Realm B receives the message through a static join and therefore forwards the message to all members of its own zone that have registered interest.
6. Realm A has registered interest as it has a local subscriber. It therefore receives this message and delivers it to the local subscriber. There are no other realms with interest so it does not forward the message further.

## Example 2



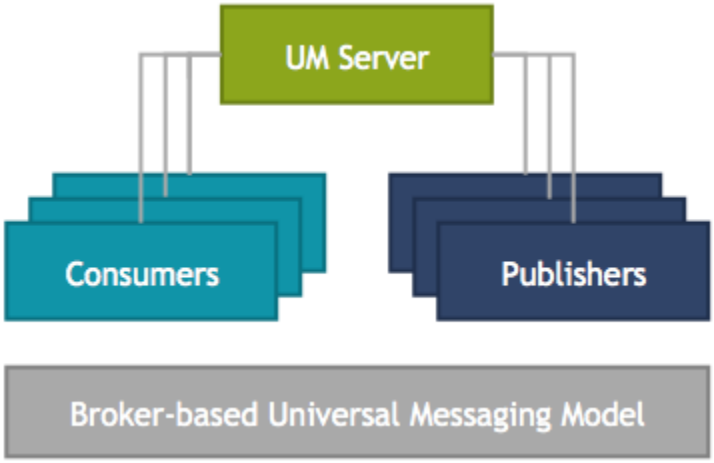

1. A message is published to a channel on Realm I in Zone C.
2. Realm I forwards the message to all realms in the same zone where interest has been registered (Realm H, Realm G), as it is the initial receiver of the message.
3. Realm H forwards the received message to its subscriber. Server H has no statically configured joins so does not forward the message any further.
4. Realm G receives the message and forwards it over the static joins to Realm C in Zone A and Realm D in Zone B.
5. Realm C receives the message and forwards it to all members in Zone A that have interest.
6. Realm A and Realm B deliver the message to their subscribers.
7. Realm D receives the message and forwards it to all members of Zone B that have interest. Therefore it sends the message to Realm F. Realm E has no interest and therefore it does not forward the message to this server.
8. Realm F delivers the message to its subscriber.

# 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

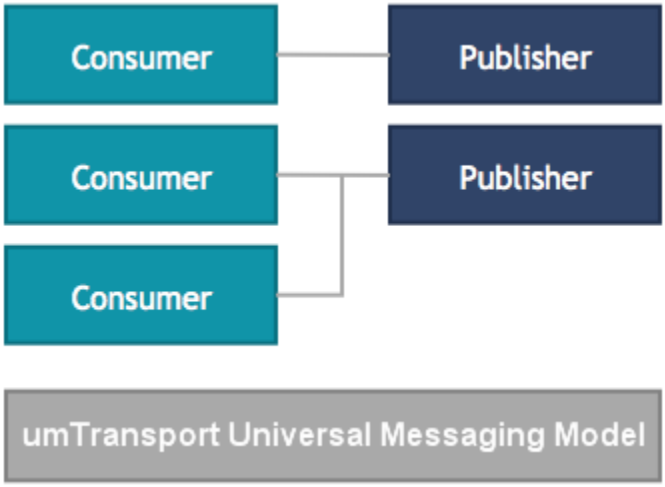
Historically, messaging architecture has predominantly been based on a 'broker in the middle' approach. This is often referred to as 'hub and spoke'. The broker acts as the communications hub, routing messages between logically decoupled peers:




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.modules.transport 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 does not support RDMA 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.modules.examples.transport.echo;
import com.softwareag.um.modules.transport.ClientContextBuilderFactory;
import com.softwareag.um.modules.transport.ClientTransportContext;
import com.softwareag.um.modules.transport.SynchronousTransport;
import com.softwareag.um.modules.transport.TransportFactory;
import com.softwareag.um.modules.examples.transport.SimpleMessage;
import com.softwareag.um.modules.examples.transport.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<SimpleMessage>(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.modules.examples.transport.echo;
import com.softwareag.um.modules.transport.ServerContextBuilderFactory;
import com.softwareag.um.modules.transport.ServerTransportContext;
import com.softwareag.um.modules.transport.SynchronousServerTransport;
import com.softwareag.um.modules.transport.SynchronousTransport;
import com.softwareag.um.modules.transport.TransportFactory;
import com.softwareag.um.modules.examples.transport.SimpleMessage;
import com.softwareag.um.modules.examples.transport.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();
  }
  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<SimpleMessage>(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
    std::shared_ptr<SynchronousTransport> transportShared(std::move(transport));
    SynchronousClient<SimpleMessage> client(std::move(transportShared));
    //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{
        SimpleMessage sm(input);
        client.write(sm);
      }
    }
    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)
 :
stopped(false){
  //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
      std::shared_ptr<SynchronousTransport> syncTrShared(
           std::move(m_transport->accept()));
      SynchronousClient<SimpleMessage> client(syncTrShared);
      //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 301) and Comet Communication Protocols (see "Comet Communication Protocols" on page 304).

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 | | ✓ | ✓ |

| | 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 301) RNAME string looks like this (using a comma-separated list):

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

or (using a semicolon-separated list):

```
<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 on.
- `<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 the traversal of the RNAME values in the order given, while using a semicolon-separated list indicates the random traversal of the RNAME values.

If a list of RNAMEs is used and the Universal Messaging session becomes disconnected and cannot reconnect, the API will traverse 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 168) and failover (see "Failover" on page 134).

## 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 the following extension available to them:

- *nhpsc*

  This mode of https extracts any configured proxy from within the JVM settings and issues a PROXY CONNECT command via this 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.

## 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.

# 2 Management


- Administration and Management ..... 26
- JMX Administration and Management ..... 27
- JMX User Guide ..... 34

# 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 26
- "Monitoring of Events" on page 26
- "Creation of Universal Messaging Resources, ACLs and Clusters" on page 26
- "Management of Configuration Parameters" on page 26
- "Seamless Integration with Third Party Enterprise Systems Management Tools" on page 27


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. See the code example "Monitor the Remote Realm Log and Audit File" for an illustration of usage.

### 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. See the code example "Monitor Realms for Client Connections Coming and Going" for an illustration of usage.

### Resource Creation

Resources (see "Messaging Paradigms" on page 145) 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

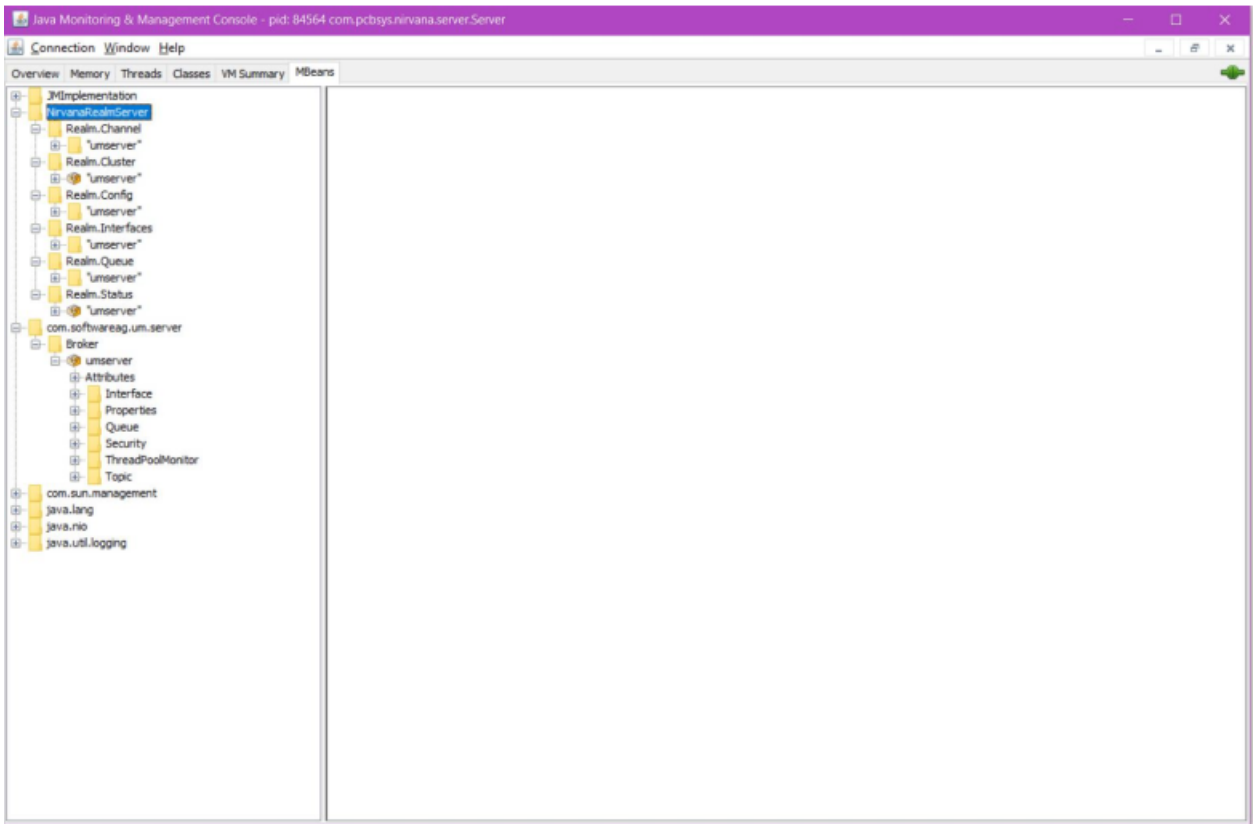
files enabling the very fast bootstrapping of new environments. The Enterprise Manager documentation has an XML sample.

### 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 27) 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:




In order to connect to Universal Messaging over the network using JConsole or another tool that supports JMX, you must first enable the realm configuration property `EnableJMX`. When you enable this property, you activate the view **com.softwareag.um.server** shown in the screenshot above. The node **Broker** shown directly under **com.softwareag.um.server** in the screenshot is the name of the type of JMX bean that generates the JMX metrics for the view.

For legacy handling of JMX, you must first enable the realm configuration property `EnableLegacyJMX`. When you enable this property, you activate the view **NirvanaRealmServer** shown in the screenshot above.

`EnableJMX` and `EnableLegacyJMX` can be enabled in the same Universal Messaging session, and in this case the JConsole display will show both **com.softwareag.um.server** and **NirvanaRealmServer**.

**Note:**
The **NirvanaRealmServer** view is deprecated in Universal Messaging v10.7 and will be removed in a later version of the product.

For information on setting these properties, see the section *Realm Configuration* of the Enterprise Manager documentation.

**Note:**

By default, the JMX RMI interface is not open. If you want to open it, use the realm configuration property `JMXRMIPort`.

See the section refer to the section *Realm Configuration* in the *Administration Guide* for details.

Start JMC (Java Mission Control), or whatever tool is your favorite, and connect with a JMX server URL of the following form:

```
service:jmx:rmi:///jndi/rmi://localhost:<JMXRMIPort>/server
```

For example, if you have set `JMXRMIPort` to "9010", then the URL is:

```
service:jmx:rmi:///jndi/rmi://localhost:9010/server
```

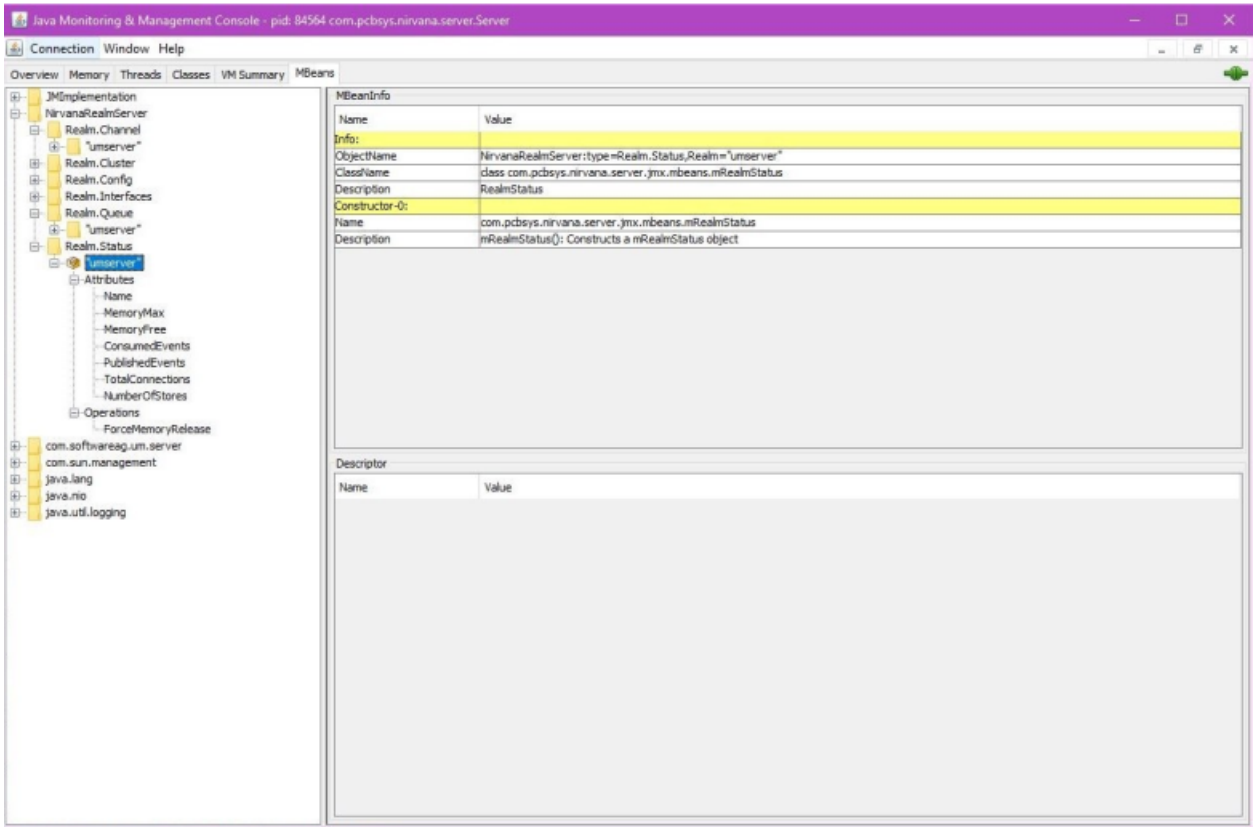
If you wish to enable authentication or SSL, please consult the JMX documentation or contact our support team.

Universal Messaging offers the following JMX beans:

- "Universal Messaging Realm Status JMX bean" on page 28
- "Universal Messaging Realm Cluster JMX bean" on page 29
- "Universal Messaging Realm Configuration JMX bean" on page 30
- "Universal Messaging Realm Interfaces JMX bean" on page 31
- "Universal Messaging Channel JMX bean" on page 32
- "Universal Messaging Queue JMX bean" on page 33

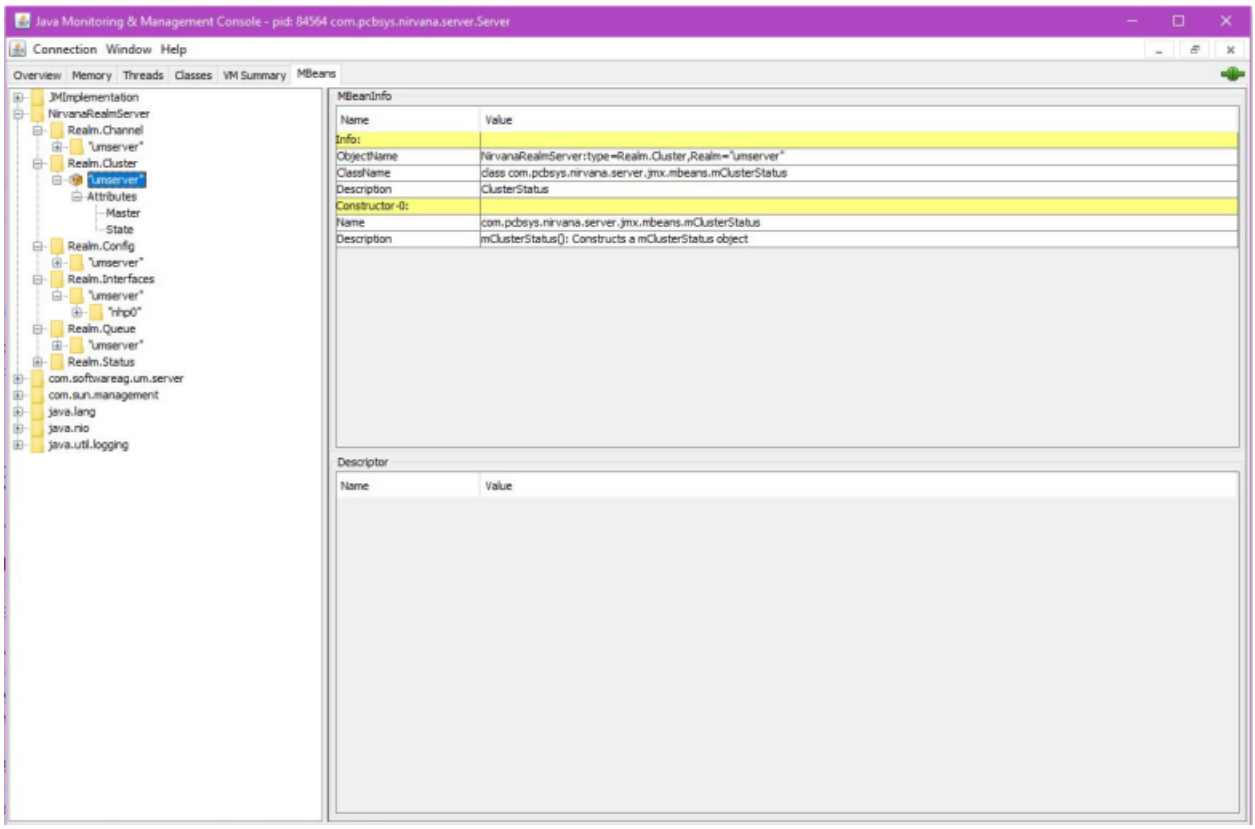
## Universal Messaging Realm Status JMX bean

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:



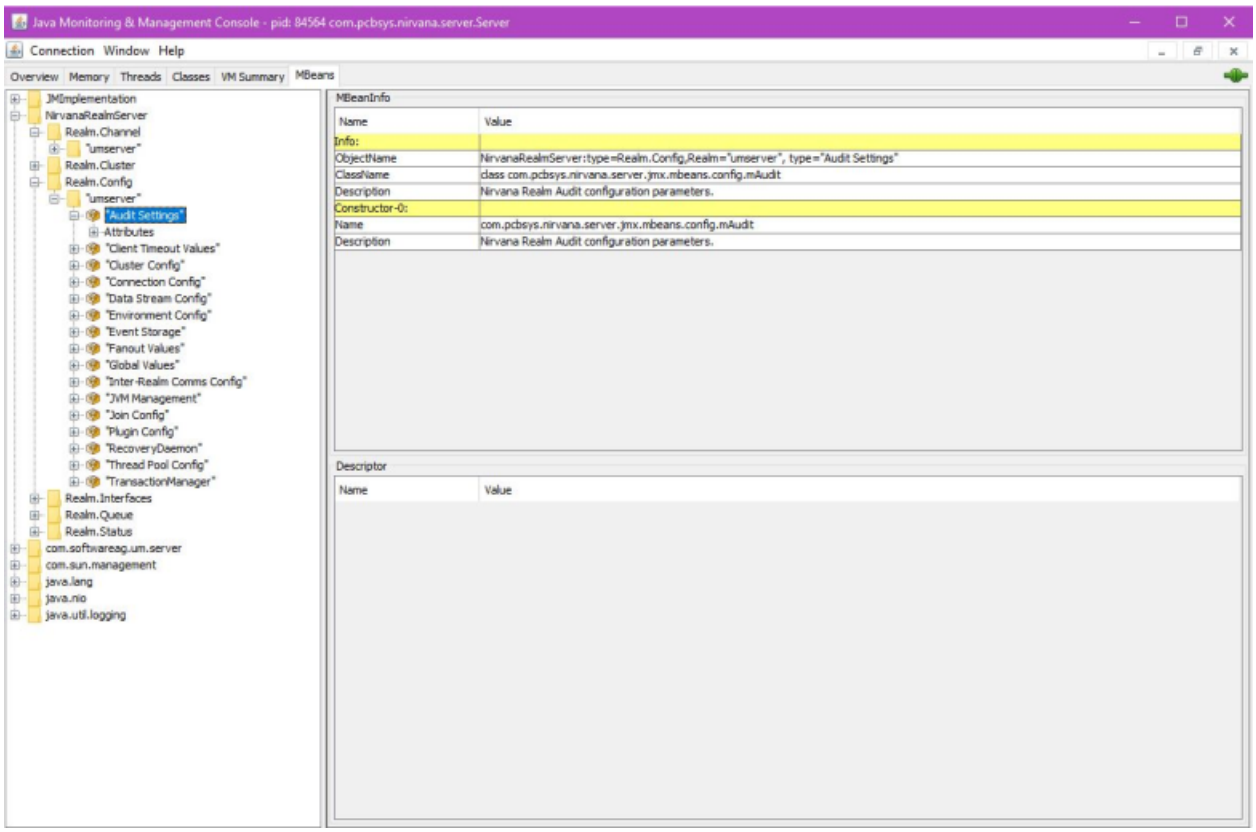

## Universal Messaging Realm Cluster JMX bean

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:



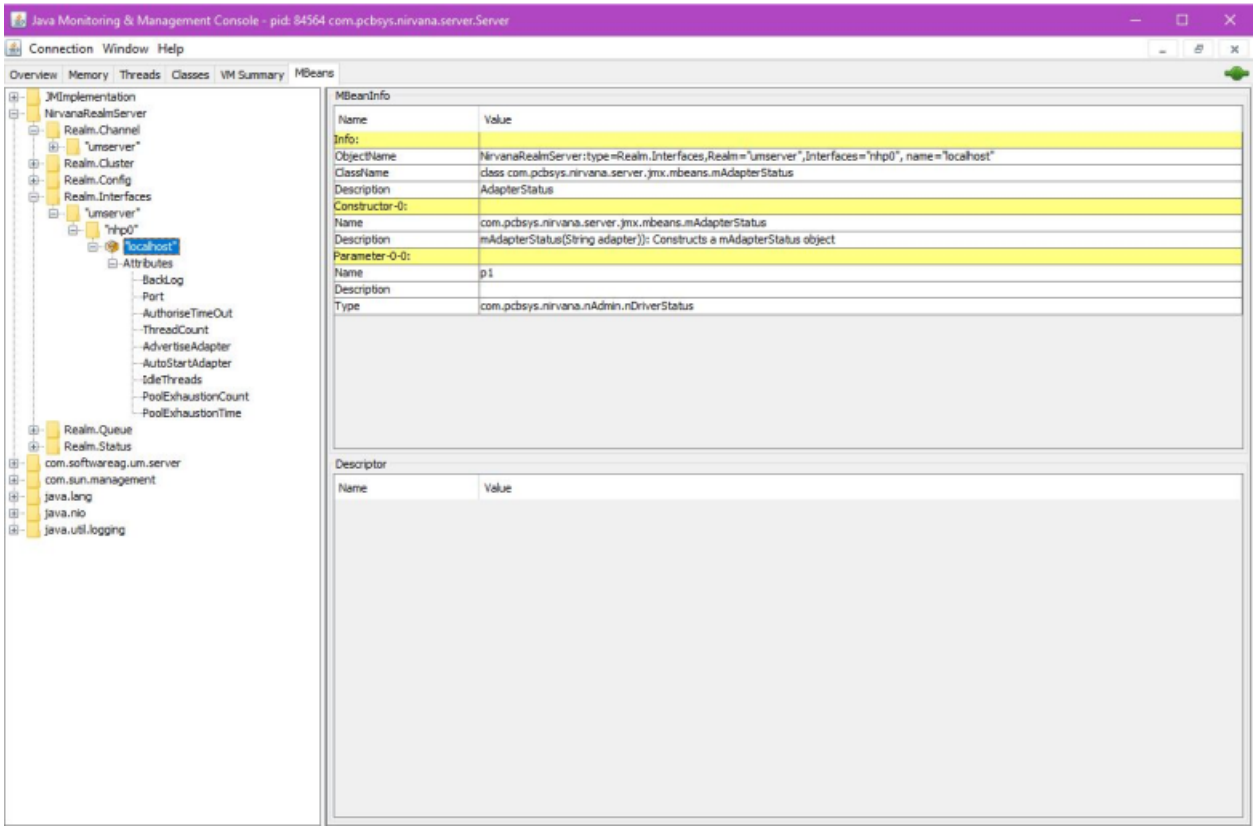

## Universal Messaging Realm Configuration JMX bean

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:



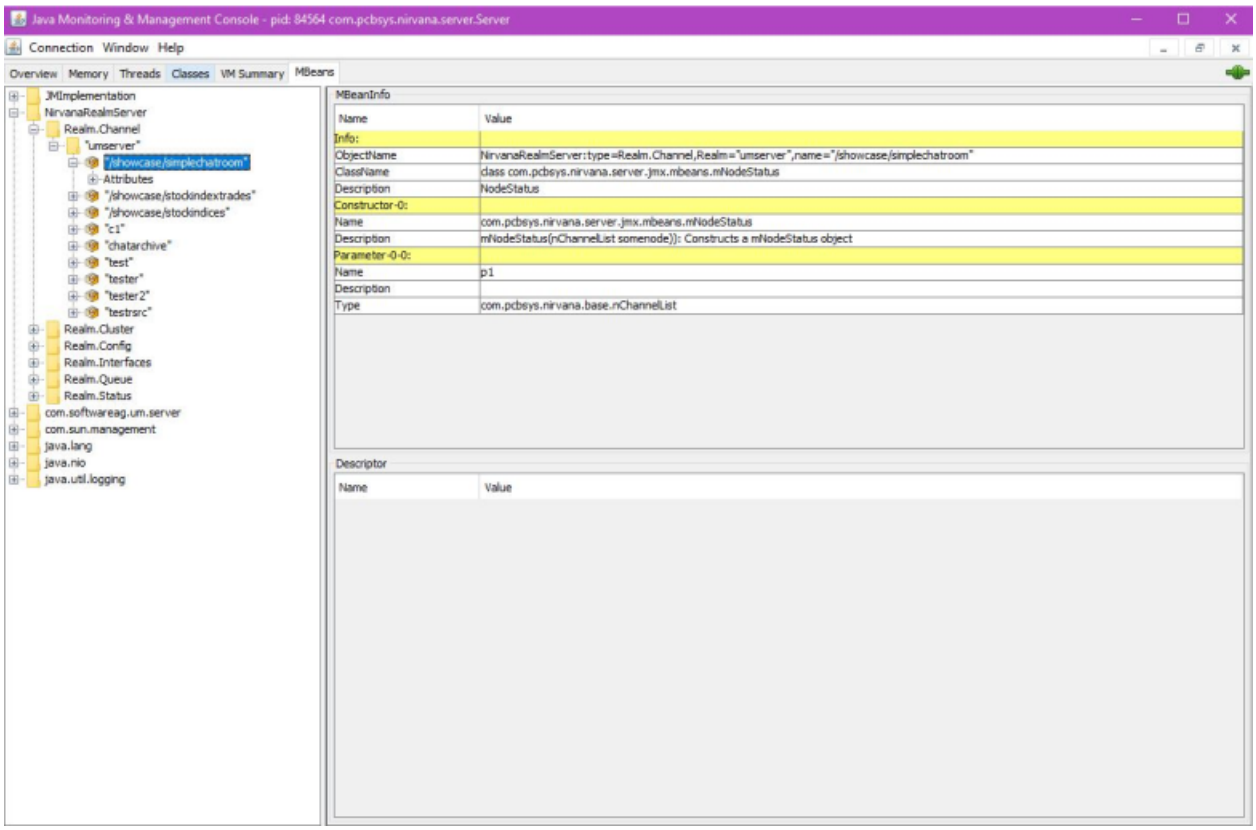

## Universal Messaging Realm Interfaces JMX bean

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:



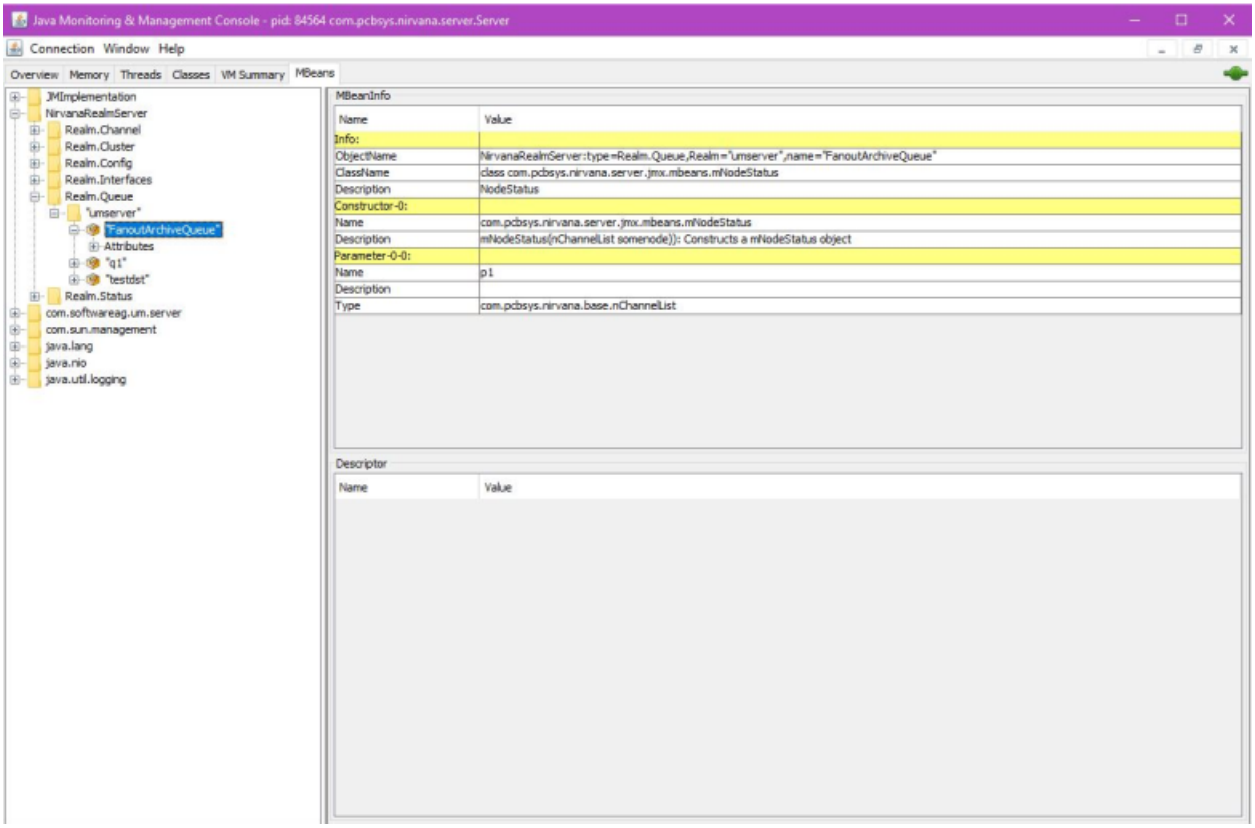

## Universal Messaging Channel JMX bean

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:




## 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:




# JMX User Guide

## Enabling JMX on the server

You can enable JMX on the server by using either the Enterprise Manager or the Admin API.

### Using the Enterprise Manager

1. Select **Realm > Config > JVM Management**
2. Set the property `EnableJMX` to "True".

### Using the Admin API

Use code such as the following:

```
nRealmNode node = new nRealmNode(new nSessionAttributes("<UM adapter>"));
nConfigGroup config = node.getConfigGroup("JVM Management");
Enumeration changedItems = config.getItems();
while (changedItems.hasMoreElements()) {
  nConfigEntry entry = (nConfigEntry) changedItems.nextElement();
  if (entry.getName().equals("EnableJMX")) {
    entry.setValue("true");
  }
}
node.commitConfig(config);
```

### JMX RMI Connections

If you want to connect to the JMX Bean Server remotely you can do it through the RMI Connection. By default this connection is not enabled. In order to enable the connection, do this:

1. In the Enterprise Manager, select **Realm > Config > JVM Management**
2. Set the property `JMXRMIPort` to a valid, not bound port number.

Changes will take effect immediately and no server restart is required. If the port is not available for some reason the realm server will go through the next 10 consecutive ports and will book the first free port. In order to disable the RMI, set the property to '0'.

**Note:**
The port is not secured in any way, so using this method to connect to the Bean Server is not recommended for a production environment.

### Using a generic JMX client such as JConsole

You can connect to the PID. The process name should be `com.pcbsys.nirvana.server.ServerService` or similar. The beans are arranged in a tree like structure, similar to the display of assets shown in the Enterprise Manager.

### Switching to Flat Namspeaces for Topics and Queues

By default, JMX namespaces for topics and queues are created using a folder tree structure, for example, `Folder1=q1,Folder2=q2,destinationName=queuename`, which might cause issues for some monitoring solutions.

For this reason, you can switch to flat namespaces for JMX topics and queues by using the `FlatStoreJMXBeanNamespace` realm configuration property, as follows:

1. In Enterprise Manager, select a realm and go to **Config > JVM Management**.
2. Set the **FlatStoreJMXBeanNamespace** property to **true**.
3. Disable the **EnableJMX** property to stop the JMX server.
4. Set **EnableJMX** to **true** again.

After you you enable the functionality, a JMX queue namespace, for example, will have the format `destinationName=/q1/q2/q3`.

### Universal Messaging - Metrics

The following section gives details about the metrics that Universal Messaging provides.

A complete list with Universal Messaging metrics is available in the JMX Exporter configuration file `jmx_exporter.yaml` mentioned above.

| **Server Statistics** | |
|---|---|
| | |
| HeapMemory | Shows heap memory usage in megabytes for the realm server instance. |
| MemoryFree | Amount of memory used by the realm |
| DirectMemory | Realm direct memory usage |
| MemoryMax | Amount of memory available for the realm |
| DiskUsage | Disk usage |
| CPU | CPU loads from UM perspective |
| CurrentConnections | Number of current connections for the realm |
| BytesIn | Realm's number of arrived bytes |
| BytesOut | Realm's number of consumed bytes |
| TotalPublishedEvents | Number of events published on the realm |
| TotalConsumedCount | Number of consumed events |
| | |
| **Cluster Metrics** | |
| | |
| ProcessQueue | The size of the queue waiting to be processed |
| ClientQueue | The size of the local client queue |
| CommsQueue | The size of the incoming cluster queue request |
| CurrentState | The current realm's cluster state |
| IsOnline | Is this realm member online to this node |
| | |
| **Thread Pools Statistics** | |
| | |
| Idle Threads | Realm server Thread Pool Monitor Idle threads. This attribute shows the number of idle threads. |
| Queued Tasks | Realm server Thread Pool Monitor Queued tasks. This attribute shows the number of queued tasks. |
| Allocated Threads | Realm server Thread Pool Monitor allocated threads. This attribute shows the number of allocated threads. |

| | |
|---|---|
| Stalled Tasks | Realm server Thread Pool Monitor stalled tasks. This attribute shows the number of stalled tasks. |
| Tasks Added | Realm server Thread Pool Monitor added tasks. This attribute shows the number of added tasks. |
| | |
| **Event Handling** | |
| | |
| TotalPublished | Number of total published events per channel, specific attribute for channel/queue |
| TotalConsumed | Number of consumed events per channel, specific attribute for channel/queue |
| NoOfEvents | Number of channel/queue events |
| MemoryUsage | The current memory used by the store |
| UsedSpace | The current space used by the store |
| CurrentConnections | The current number of connections to this node |
| TotalEvents | Durable metric, durable total number of events |
| Pending | Durable metric |
| | |
| **Number of Connections (Subscribers)** | |
| | |
| Acknowledged | Durable metric, durable connection number of acknowledged events |
| RolledBack | Durable Metric, durable connection number of rolled back events |
| Pending | Durable metric, durable connection number of pending events |
| | |
| **Queue data** | |
| | |
| TotalPublished | Number of total published events per queue, specific attribute for channel/queue |
| TotalConsumed | Number of consumed events per queue, specific attribute for channel/queue |
| NoOfEvents | Number of channel/queue events |

| | |
|---|---|
| MemoryUsage | The current memory used by the store |
| UsedSpace | The current space used by the store |
| CurrentConnections | The current number of connections to this node |

# 3 Deployment


- Deployment ..... 40
- Bi-directional Client and Server Compatibility ..... 40
- Server ..... 42
- Client ..... 61
- Language Deployment Tips ..... 69

# 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.

# Bi-directional Client and Server Compatibility

Universal Messaging enables both higher-release Java clients to connect to lower-release realm servers, and lower-release clients to connect to higher-release servers. The bi-directional client-server compatibility offers more flexible deployment strategies for both upgrades and daily operations, regardless of whether you are running higher-release clients, higher-release servers, or a mix. The functionality allows less release and upgrade planning between core components, resulting in faster deployments and less complexity in those deployments.

The Universal Messaging Java client version 10.5 and higher, and components built upon it including JMS and Resource Adapter, can connect to both lower-release and higher-release Universal Messaging servers. Java clients version 10.3 and lower can connect only to the same version or higher-release servers.

To use the bidirectional client-server functionality with Universal Messaging 10.3, 10.5, and 10.7, respectively, you must have the following Realm Server, Java Client, Shared Bundles, and Common Libraries fixes installed:

- Universal Messaging 10.3 Fix 26
- Universal Messaging 10.5 Fix 12
- Universal Messaging 10.7 Fix 4

Be aware that not all fixes might be applicable to your installation. Software AG Update Manager displays only the fixes required by your installation.

The following table shows the backwards compatibility, that is, higher-release clients connecting to lower-release servers, between the different supported Universal Messaging client and server versions.

| **Client** | **Server** | | | |
|---|---|---|---|---|
| | **10.3 Fix 26** | **10.5 Fix 12** | **10.7 Fix 4** | **10.11** |
| **10.3** | ✔ Supported Client Release Functionality is 10.3 | ✔ Supported Client Release Functionality is 10.3 | ✔ Supported Client Release Functionality is 10.3 | ✔ Supported Client Release Functionality is 10.3 |
| **10.5** | ✔ Supported Client Release Functionality is 10.3 | ✔ Supported Client Release Functionality is 10.5 | ✔ Supported Client Release Functionality is 10.5 | ✔ Supported Client Release Functionality is 10.5 |
| **10.7** | ✔ Supported Client Release Functionality is 10.3 | ✔ Supported Client Release Functionality is 10.5 | ✔ Supported Client Release Functionality is 10.7 | ✔ Supported Client Release Functionality is 10.7 |
| **10.11** | ✔ | ✔ | ✔ | ✔ |

| Client | Server | | | |
|---|---|---|---|---|
| | Supported Client Release Functionality is 10.3 | Supported Client Release Functionality is 10.5 | Supported Client Release Functionality is 10.7 | Supported Client Release Functionality is 10.11 |

**Important:**
Be aware that new client APIs added in a higher release that are not supported by the server component return a `com.pcbsys.nirvana.UnsupportedServerOperationException`, which extends `java.lang.UnsupportedOperationException`. For more details, see the Universal Messaging Java Client API documentation.

In addition, for information about how the bi-directional client and server compatibility impacts horizontal scalability environments, see "Horizontal Scalability Behavior in a Mixed Landscape" on page 142.

# Server

## 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 using Channel Joins

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

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

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

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

Joins can also be created in several configurations:

- A join may be created between a channel on a realm and a channel or queue on another realm federated with the source realm (see "Federation Of Servers" on page 45 for related information about setting up federation between two realms);
- A join can be created from a channel on a clustered realm to a channel or queue within the same cluster. A non-clusterwide channel can be joined to a cluster-wide channel or queue, but not vice versa;
- A channel can be joined from a channel on one cluster to a channel on another cluster by using an inter-cluster join (see "Inter-Cluster Joins" on page 44).
- Universal Messaging supports joins where the destination (incoming) is a queue. Universal Messaging does not support joins where the source of the join is a queue.

### 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 hop count 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 hop count of 5 then the event will be found on channel0 (the source channel), channels 1 to 5 (the intermediate channels) and channel6 (the endpoint).

### Event Filter

Joins can be created using filters , so that only specific events published to the source channel that match certain search criteria will be routed to the destination channel.

### 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 hop count 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.

### Archival Joins

An archival join is a specific type of join made between a channel and a queue, where events will not be checked for duplication. Events that are published to the source channel will appear in the destination queue. This may result in duplicate events in the queue if the queue has multiple sources.

A typical scenario for using this feature is if you have multiple routes that messages can take to reach the same destination, and you want to retain all duplicates at the destination. For example:

- if an event is duplicated at the source.
- if an event is copied from channel A -> B -> C and from channel A -> D -> C.
- if you define multiple paths over different network protocols between the same source and destination.

Without this feature, the duplicate detection would ensure that only one copy of the message is delivered to the destination queue.

### Inter-Cluster Joins

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

See the section "Inter-Cluster Connections" on page 188 for related information.

### Related Links

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

## Data Routing using Interest Propagation

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

There is an alternative mechanism, namely *Interest Propagation*. This mechanism expands upon the functionality provided by joins by providing the ability to forward events only when there

are subscribers to a channel of the same name on the remote realm or cluster. Forwarding only events which have an active subscription reduces the number of events and bandwidth used on these links.

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

### Managing Remote Interest using the Administrative API

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

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

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

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

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

### Sample Usage

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

## 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 or queues.

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 Enterprise Manager 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 "Using HTTP/HTTPS" on page 61) 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 *Reliable*.

If you have a Reliable 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 specified in the `Server_Common.conf` configuration file. This file is located in the `<InstallDir>`/UniversalMessaging/

`server/<InstanceName>/bin` directory of your installation, where *<InstanceName>* is the name of the Universal Messaging realm.

The -D options are specified in the configuration file in the following format:

```
wrapper.java.additional.<n>=-D<parameter>=<value>
```

where <parameter> is the name of the parameter whose value you wish to set, <value> is the value that you want to assign to the parameter, and <n> is any positive integer. If you have several -D options in the file, ensure that <n> values are unique. If you supply several -D options with the same <n> value, only the last of the specified options with this value of <n> will be effective and the earlier ones with this value of <n> will be ignored.

Example:

```
wrapper.java.additional.25=-DMaxFileSize=4000000000
wrapper.java.additional.26=-DLOGLEVEL=3
```

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

### Description of the Server Parameters

This section describes the parameters that are used when the server starts, what they are used for, and what their typical values are.

**Note:**

In some cases, the initial values described here can be changed dynamically, i.e. while the server is running, by using realm configuration properties. For example, the initial value set by the server startup parameter LogFileDepth can be superseded for a running server by the realm configuration property RolledLogFileDepth.

See the section *Realm Configuration* in the *Administration Guide* for further information.

| Name | Required | Default | Description |
|---|---|---|---|
| ADAPTER | N | | Specifies an interface to use, for example nsp://0.0.0.0:9000/ |
| ADAPTER_x | N | | Specifies an interface to use, for example nsp://0.0.0.0:9000/ where x = 0 -> 9 |
| CACHE_SIZE | N | 10 000 | Specifies how many events to cache in memory for a store with multi-file disk storage (where the "events per spindle" storage property is set above 0), thereby avoiding I/O operations on disk. |

| Name | Required | Default | Description |
|---|---|---|---|
| | | | You might want to increase or decrease this value depending on your publish and consume rates, available heap size and individual event size. |
| CAKEYSTORE | N | | Shorthand for javax.net.ssl.trustStore |
| CAKEYSTOREPASSWD | N | | Shorthand for javax.net.ssl.trustStorePassword |
| CHANNELUMASK | N | | Specifies the default channel protection mask |
| CKEYSTORE | N | | Shorthand for javax.net.ssl.keyStore |
| CKEYSTOREPASSWD | N | | Shorthand for javax.net.ssl.keyStorePassword |
| DATADIR | Y | | The path that Universal Messaging uses to store its internal status and configuration information about the realm. See the section below for related information. |
| DISK_USAGE_SCAN_ENABLE | N | (no default) | **Note:** This parameter is deprecated in Universal Messaging version 10.7 and will be removed in a subsequent release.<br><br>The parameter determines whether to check at regular intervals if there is sufficient free disk space for the Universal Messaging server to continue normal processing. This parameter can be set to `true` or `false`, or can be left undefined. If it is set to `true` or `false`, it overrides the setting of the realm configuration property `DiskScanEnable`.<br><br>If `DISK_USAGE_SCAN_ENABLE` is set to true, the percentage of available free space must be greater than the value given by the realm |

| Name | Required | Default | Description |
|---|---|---|---|
| | | | configuration parameter `DiskUsageFreeThreshold` in order for processing to continue normally. If the amount of free disk space drops below this percentage value, the server logs an error message. The server will initiate a clean shutdown with an appropriate error message if the disk space available is less than 500 MB.<br><br>If `DISK_USAGE_SCAN_ENABLE` is set to false, no check of available free space is done; in this case, if the server runs out of disk space, an unorderly shutdown may result.<br><br>If `DISK_USAGE_SCAN_ENABLE` is not set, i.e. is set to neither true nor false, the regular checks for free disk space can still be carried out if the realm configuration parameter `DiskScanEnable` is set to true.<br><br>See the description of realm configuration properties in the section *Realm Configuration* in the *Administration Guide*. |
| javax.net.debug | N | | Useful to debug SSL issues. |
| 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 |

| Name | Required | Default | Description |
|---|---|---|---|
| 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 |
| LOG_FRAMEWORK | N | | Specifies a third part logging framework to use. Can be set to "LOGBACK" or "LOG4J2". The default is "fLogger". |
| LogFileDepth | N | | The number of log files to keep on disk when using log rolling. The oldest log files will be deleted when new log files are created. |
| LOGLEVEL | N | 5 | Specifies the current log level to use |
| LOGFILE | N | System.out | Specifies a log file to write the log entries to |
| LOGSIZE | N | 100000 | The maximum size (in bytes) of the log file before the log file is rolled |
| MaxFileSize | N | 1000000000 (1GB) | Specifies the maximum size (in bytes) of a channel/queue file on disk, before automatic maintenance (auto-maintenance) is performed to re-order the file to remove purged events. The default is 1GB, although auto-maintenance is usually performed well before this limit is reached. |
| MaxMemory | N | Uses the -Xms (minimum heap size) value | Specifies a target value for the maximum memory usage. As this value is approached, the realm server will attempt to free caches in order to release memory. |
| MaxMemSize | N | | Specifies the default value of the "MaintenanceMemoryThreshold" realm configuration option. |

| Name | Required | Default | Description |
|---|---|---|---|
| | | | See the section *Realm Configuration* in the *Administration Guide* for related details. |
| mode | N | | If set to IPAQ forces a small memory mode for the server |
| Nirvana.auth.exempt | N | | Specifies the path to a text file that contains a list of users exempt from authentication. For more information about defining exempt users and configuring the property, see "Server-side Authentication" on page 90. |
| REALM | Y | | Specifies the name of the Realm Server. When creating a server instance, the server instance name is set to the realm name by default. After you create a server instance you cannot change the instance name or the realm name.<br><br>For more information about creating instances, see *Universal Messaging Instance Manager* in the *Installation Guide*. |
| 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 `Server_Common.conf` file for the super user entries in the file to be added to the realm ACL. |
| SSLProtocols | N | TLSv1,TLSv1.1,TLSv1.2 | Specifies one or more (comma-separated) SSL protocols that the realm server is allowed to |

| Name | Required | Default | Description |
|---|---|---|---|
| | | | use. If an attempt is made to use any other protocol on the server, an error will be raised. |

## Notes on DATADIR

The server parameter DATADIR defines a location where Universal Messaging stores its internal status and configuration information for the realm. This location is set automatically when you create a realm. Normally you do not need to change this location.

DATADIR points to the location `server/<realmname>/data`. This location contains various files and subdirectories. If you define any plugins, Universal Messaging stores the related internal configuration information in the parallel location `server/<realmname>/plugins`.

**Note:**
The `data` and `plugins` locations contain various files and subdirectories that are essential for the smooth running of the product, and any attempt by the customer to modify these files directly may make the system unusable.

The value of the DATADIR parameter is set in the following files that are located in the realm-specific location `server/<realmname>/bin`:

- `CertificateGenerator.conf`
- `env.bat` on Windows, or `env.sh` on UNIX-based systems
- `nstopserver.bat` on Windows, or `nstopserver` on UNIX-based systems
- `Server_Common.conf`

**Note:**
On UNIX-based systems, the environment variable in the `nstopserver` file is named `data` instead of `DATADIR`.

There are some rare cases when you might want to change the value of DATADIR to point to a new location. Such cases are:

- You want to point DATADIR to a new location but you want the realm server to continue using all of its existing status and configuration data.
- You want to point DATADIR to a new location but you do not want the realm server to use any existing status or configuration data.

The steps required to implement these cases are described in the following sections.

**Case 1: Define a new DATADIR, retaining the existing realm status and configuration data**

1. If the realm server is running, stop the realm server.

2. In a temporary location, make a copy of the contents of the `server/<realmname>/data`. Also copy `server/<realmname>/plugins` if you have defined any plugins.
3. Change the value of DATADIR in the files listed above. Ensure there are no trailing spaces in the new value. In the `nstopserver` file on UNIX-based systems, change the value of the variable `data` instead of `DATADIR`.

   If you specify a value for DATADIR that does not end with "/data", the realm server will assume a value for DATADIR as if you had appended "/data" to the value.

   For example, if you define DATADIR to be "/folder1/folder2/data", then the realm server uses exactly this value as the new DATADIR. If however you define DATADIR to be "/dir1/dir2", then the realm server will behave as if you had specified "/dir1/dir2/data".
4. The following steps 5-7 describe how to create the structure for the new `data` directory and copy in the required files from the old `data` directory. Note that there is an alternative method, described at the end of this procedure.
5. Start the realm server. This automatically creates the structure for the `data` directory at the location defined by the new DATADIR.
6. Stop the realm server.
7. Overwrite the contents of the new `data` directory and all subdirectories with the temporary copy of the `data` directory and all subdirectories you stored earlier.
8. If you had defined plugins, copy in the `plugins` directory and contents from your temporary copy.
9. Start the realm server.

As an alternative to steps 5-7 above, you can create the new `data` directory manually using standard operating system commands while the realm server is stopped, and copy in the contents of the old `data` directory and all subdirectories.

**Case 2: Define a new DATADIR, without using any existing realm status or configuration data**

If you want to define a new value for DATADIR but you do not want to copy in the existing status and configuration data from the old DATADIR location, follow the instructions in Case 1 above but omit any steps that involve manually copying data to the new DATADIR location. This means that when the realm server starts for the first time after DATADIR has been changed, the realm server has no knowledge of any previously stored data.

**Template files**

Note that there are also template files in the location `server/templates`. These template files are used to automatically create the above mentioned realm-specific files when you create a realm. If you want to create new realms and ensure that they also use the new value of DATADIR, you need to change the contents of these template files before you create the new realms.

## 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" on page 145) 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). 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.

The Universal Messaging Client API supports both custom SSL properties that can be set per UM session, and JSSE and Universal Messaging client properties that can be set via system properties.

When using custom SSL properties that are set on the session level, you can set keystore/trustore and other SSL properties for each of your connections.

When using JSSE, only 1 keystore file/keystore password and 1 truststore file/truststore password can be used. In order to achieve your goal you will then have to create a combined keystore and / or a combined truststore depending on your configuration.

### Setting Custom SSL Properties on a Session

Setting SSL properties directly is done using the session attributes. Before initializing the session, ensure that your session attributes for each session are set with the correct keystore and truststore information. You can also set the alias of the certificate to use when client certificate authentication is enabled on the interface you are connecting to (see "Client SSL Configuration" on page 104 for more info). If you are using JMS then you will need to set the SSL properties directly on the connection factory or on the create connection overload method provided as part of the connection factory implementation (see "JMS Client SSL Configuration" on page 107 for more info).

### Setting SSL Properties Using JSSE

Client certificate authentication NOT required

**Important:**
The CKEYSTORE, CKEYSTOREPASSWD, CAKEYSTORE, and CAKEYSTOREPASSWD system properties are deprecated.

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

**Important:**
The CKEYSTORE, CKEYSTOREPASSWD, CAKEYSTORE, and CAKEYSTOREPASSWD system properties are deprecated.

In the case where client certificate authentication is 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 (deprecated) 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 want 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 the 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.

### Using the Universal Messaging Client System Properties for Secure Communication

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

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

```
-Dcom.softwareag.um.client.ssl.keystore_path=
```

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

where:

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

## Periodic Logging of Server Status

The Universal Messaging server writes status information to the log file at regular intervals. The default interval can be configured using the `StatusBroadcast` realm configuration property, and the default value is 5 seconds).

For information on realm configuration properties, see the section *Realm Configuration* in the *Enterprise Manager* part of the *Administration Guide*.

Here is a sample status log message:

```
ServerStatusLog> Memory=481, Direct=355, Mapped=0, OpenFileDesc=81,
EventMemory=0, Disk=81913, CPU=1.5, Scheduled=51, Queued=0,
Connections=2, BytesIn=3898, BytesOut=194852, Published=0, Consumed=0,
QueueSize=0, ClientsSize=0, CommQueueSize=0
```

The following table describes the status log metrics:

| Metric | Description | Unit |
|---|---|---|
| Memory | Free JVM memory | Megabyte |
| Direct | Free direct memory | Megabyte |
| Mapped | Used mapped memory | Megabyte |

| Metric | Description | Unit |
|---|---|---|
| OpenFileDesc | On Unix only. The number of file descriptors opened by the Universal Messaging server process | |
| EventMemory | JVM memory consumed by events | Megabyte |
| Disk | Free disk space in UM data directory | Megabyte |
| CPU | Recent CPU usage for the JVM process in percent. Valid values are from `0` to `100` percent. | % |
| Scheduled | Number of tasks in the internal scheduler | |
| Queued | Number of total tasks currently queued in all thread pools. A value of zero indicates that at this moment in time there were no tasks waiting for a thread. | |
| Connections | Active Universal Messaging connections | |
| BytesIn | Total bytes received for currently active Universal Messaging connections | byte |
| BytesOut | Total bytes sent for currently active Universal Messaging connections | byte |
| Published | Total number of published events | |
| Consumed | Total number of consumed events | |
| QueueSize | The count of requests/responses awaiting processing that have arrived from other nodes in the cluster (inbound from the cluster).<br><br>On the master node:<br><br>■ the requests are from slave nodes, which forward requests from their locally attached clients to the master node.<br><br>■ the responses are from slave nodes, to confirm that the slave nodes have processed client requests previously propagated by the master node.<br><br>On slave nodes:<br><br>■ the requests are from the master node, which propagates (a) requests from | |

| Metric | Description | Unit |
|---|---|---|
| | clients attached locally to the master, and (b) requests received by the master from clients attached to slave nodes. The propagation goes to all slave nodes in the cluster.<br>■ the responses are from the master node, to confirm that the master node has processed client requests that were originally sent by the current slave node.<br>A continuously growing size of this queue indicates that the server is not able to process the cluster messages at the rate they are arriving.<br>This value is only present if the node is part of a cluster. | |
| ClientsSize | The count of local client requests awaiting processing by the current node.<br>On the master node, requests from locally attached clients will be processed on the master node, then propagated to the slave nodes in the cluster.<br>On slave nodes, requests from locally attached clients will NOT be processed locally but instead will be forwarded to the master for processing.<br>A continuously growing value of this metric indicates that the server is not able to process and send the client requests to other cluster nodes at the rate these requests are received.<br>This value is only present if the node is part of a cluster. | |
| CommQueueSize | The count of all requests/responses waiting to be sent to the other nodes of the cluster.<br>A continuously growing value of this metric indicates that writing data to the other cluster nodes over the network is slower than the rate at which messages are processed and queued. This indicates a | |

| Metric | Description | Unit |
|---|---|---|
| | network problem at the data layer level rather than a processing problem on the node.<br><br>This value is only present if the node is part of a cluster. | |

For a diagram showing how `ClientsSize`, `QueueSize` and `CommQueueSize` interrelate, see the section "Message Passing" on page 186.

During server startup and each time the log file is rolled, a status log legend is printed in the log file:

```
ServerStatusLog> Activating Realm status log with format:
Memory - free memory (MB)
Direct - free direct memory (MB)
Mapped - Used mapped memory (MB)
OpenFileDesc - File descriptors opened by process (Unix only)
EventMemory - used event memory (MB)
Disk - free disk space (MB) in server data directory
CPU - cpu load average
Scheduled - scheduled tasks
Queued - total queued tasks in all thread pools
Connections - active connections count
BytesIn - total bytes received (for all active connections)
BytesOut - total bytes send (for all active connections)
Published - total count of published events
Consumed - total count of consumed events
QueueSize - cluster queue size
ClientsSize - cluster client request queue size
CommQueueSize - cluster communication queue size
```

Periodic logging of the server status can be disabled using the `EnableStatusLog` realm configuration property.

For information on realm configuration properties, see the section *Realm Configuration* in the *Enterprise Manager* part of the *Administration Guide*.

For information on log rolling, see the section *Universal Messaging Enterprise Manager : Logs Panel* in the *Enterprise Manager* part of the *Administration Guide*.

# Client

## 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. See the Enterprise Manager documentation for details.

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.

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 Google Chrome, Mozilla Firefox, Microsoft Internet Explorer and Microsoft Edge.

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) |
| nJMS.jar | Provides Universal Messaging Provider to support JMS functionality | nClient.jar, gf.javax.jms.jar |
| nAdminAPI.jar | Provides Universal Messaging Administration & Monitoring functionality | nClient.jar |
| nEnterpriseManager.jar | Contains the Enterprise Manager tool | nClient.jar, nAdminAPI.jar, nAdminXMLAPI.jar (Optional), gf.javax.jms.jar |
| nServer.jar | Contains the Universal Messaging Realm Server | (none) |
| nPlugin.jar | Contains the Universal Messaging Server plugins | nServer.jar |
| gf.javax.jms.jar | JMS reference jar | (none) |

The jar files are located in `<InstallDir>/`UniversalMessaging`/lib`.

### Client Jar files that are no longer available

The following table indicates Jar files that were delivered with previous product versions, but which are not delivered as Jar files any more. Their classes have now either been folded into other Jar files, or have been removed from the product altogether.

| JAR File | Description | Status |
|---|---|---|
| nJ2EE.jar | Provided Universal Messaging support for interacting with Application servers that support J2EE. | The Jar file is no longer available. The Universal Messaging product now contains a resource adapter for use with application servers. |

| JAR File | Description | Status |
|---|---|---|
| | | See the topic *Resource Adapter for JMS* in the Java section of the *Developer Guide* for related information. |
| nAdminXMLAPI.jar | Provided Universal Messaging Configuration XML Import / Export functionality. | The Jar file is no longer available, but the contents have been folded into nAdminAPI.jar. |
| nP2P.jar | Provided Universal Messaging Peer-to-Peer functionality. | The P2P functionality was removed from the product in version 9.12. The Jar file is no longer available, and the contents have not been folded into any other Jar file. |

## 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 104) 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 parameters that you can use to change the behavior of the client.

You specify the parameters in the command line of any Universal Messaging client application by prefacing the name of the parameter by `-D`. For example, to set the value of the parameter `LOGLEVEL` to `4` for a given client application, specify `-DLOGLEVEL=4` as a parameter in the command line of the client application.

The following list describes the parameters and their typical values. All parameters are optional, unless otherwise stated.

**CAKEYSTORE (deprecated)**
Shorthand for javax.net.ssl.trustStore.

**CAKEYSTOREPASSWD (deprecated)**
Shorthand for javax.net.ssl.trustStorePassword.

**CKEYSTORE (deprecated)**
Shorthand for javax.net.ssl.keyStore.

**CKEYSTOREPASSWD (deprecated)**
Shorthand for javax.net.ssl.keyStorePassword.

**com.softwareag.um.client.connection.reader.daemon**
Specifies whether the UM-Connection-Reader thread is an ordinary thread or a daemon thread. Universal Messaging client system property. Valid values are:

- `true` - The thread is daemon.
- `false` (default) - The thread is non-daemon.

If you do not specify this system property, the value is extracted from the session attributes. Implicitly, the session attributes have the value set to `false`. You can change the value using the Universal Messaging client API.

**com.softwareag.um.client.ssl.keystore_password**
The password for the client keystore. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.ssl.certificate_alias**
The alias of the certificate in the client keystore that is sent to the server if client certificate authentication is required. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.ssl.truststore_path**
The CA keystore file location. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.ssl.truststore_password**
The password for the CA keystore. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.ssl.enabled_ciphers**
Comma-separated list of ciphers from which the client is allowed to choose for secure communication. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.ssl.ssl_protocol**
The protocol that is used for secure communication. Universal Messaging client system property. For more information about using the property, see "Using the Universal Messaging Client System Properties for Secure Communication" on page 106.

**com.softwareag.um.client.follow_the_master**
Used when creating a session to realms in a cluster. Specifies whether the session always connects to the master realm. The default value is `false`.

**com.softwareag.um.client.useSharedThreadPool**
Enables usage of a shared thread pool for processing incoming nConsumeEvent events for each client session. Values are `true` or `false` (default). By default, each client session creates and uses a separate thread pool for processing received nConsumeEvent events. When this system property is set to `true`, all client sessions will use a shared thread pool. Using a shared thread pool for all client sessions may have a performance impact depending on the size of the common thread pool and the number of client sessions.

> **Note:**
> When each client session uses a separate thread pool, closing the session interrupts any event listener threads that are blocked during processing. When all client sessions share a thread pool, closing a session does not interrupt any event listener threads that are blocked during processing. Thus, the blocked event listener does not release a shared thread after closing the client session. This behavior impacts the processing of incoming events, because the shared pool will not have available threads. Software AG recommends to enable the shared pool for all client sessions only if the event listener of the client application ensures that the processing of the events is fast and graceful.

**com.softwareag.um.client.SharedThreadPoolMaxSize**
Specifies the maximum size of the shared thread pool for the client session. The default is `5` threads.

**com.softwareag.um.client.session_disable_reconnect**
Specifies whether the session attempts to reconnect to the realm server after disconnecting. If the property is set to `true`, the session does not make attempts to reconnect. The default value is `false`.

**com.softwareag.um.client.write_handler**
Specifies the Universal Messaging JVM-wide socket write handling. Valid values are:

- `standard` (default) - enables a peak write handler, which is the most optimized mode for both latency and throughput. The handler writes events directly to the socket layer until a certain number of socket writes per second occur, at which point the handler switches to queued write handing until the rate of messages decreases.
- `direct` - enables a write handler that writes events directly to the socket layer and never queues events. This mode can have an impact on latency if the throughput is higher than the capacity of the socket layer.
- `queue` - enables a write handler that never writes directly to the socket layer, but pushes all events into an in-memory queue. The events are then written to the socket layer by another thread in batches, which increases latency. However, if many events are being written, this mode prevents further spikes in latency.

**com.softwareag.um.client.network_io_buffer_size**
Specifies the receive and send buffer size in bytes when a socket is created. The larger the value of this property, the larger the buffers that the operating system needs to allocate. The default value is `1310720`.

**HPROXY**
Shorthand for http.proxyHost and http.proxyPort.

**http.proxyHost**
Sets the proxy host name to use.

**http.proxyPort**
Sets the proxy port to use.

**http.proxySet**
Set to `true` if the URL handler is to use a proxy.

**javax.net.debug**
Useful to debug SSL issues.

**jdk.tls.client.protocols**
Specifies one or more (comma-separated) SSL protocols that the client is allowed to use. If an attempt is made to use any other protocol on the client, an error will be raised. Refer also to the option `SSLClientProtocol`, which can also influence the SSL protocol to be used.

This option is only available on the Oracle® JVM.

**LOG_FRAMEWORK**
Specifies a third-party logging framework to use. Can be set to `LOGBACK` or `LOG4J2`. The default is `fLogger`.

**LOGFILE**
Used to specify a log file to write the log entries to. The default value is `System.out`.

**LOGLEVEL**
Specifies the current log level to use. The default value is `7` (OFF).

**LOGSIZE**
Specified in bytes before the log file is rolled. The default value is `100000`.

**SSLClientProtocol**
Sets the default SSL Protocol to use.

If the option `jdk.tls.client.protocols` is defined, `SSLClientProtocol` must be one of the protocols defined in `jdk.tls.client.protocols`.

If `SSLClientProtocol` is not supplied, then client SSL communication will attempt to use one of the protocols defined in `jdk.tls.client.protocols`. If `jdk.tls.client.protocols` is not defined, then an attempt will be made to use the JVM default protocol, or TLS when that is not possible.

**user.name**
Used to override the current username without coding it. The default value is "Signed on name".

## Multiplexing Sessions

**Note:**
Multiplexing sessions has been deprecated in Universal Messaging v10.7 and will be removed from the product in a future release.

Universal Messaging supports the multiplexing of sessions to a specific host in Java 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.

### Multiplex session authentication

If you create a multiplex session (i.e. a new session created by multiplexing the original session) without specifying a user name explicitly, the Universal Messaging client will use the user of the original session as the multiplex session user.

For SSL connections that are using client certificate authentication, the original session's subject uses the common name of the client certificate as the primary principal instead of any user name specified when authenticating the session. Therefore, the Universal Messaging server also uses the same client certificate principal for any multiplex sessions which do not explicitly specify a user name.

# Language Deployment Tips

## 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.

# 4 Security


- Security Overview ..... 72
- Authentication ..... 72
- Access Control Lists ..... 99
- Using SSL ..... 102
- Setting up and using FIPS ..... 109

# Security Overview

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.

# Authentication

## Authentication Overview

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 Authorization 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.

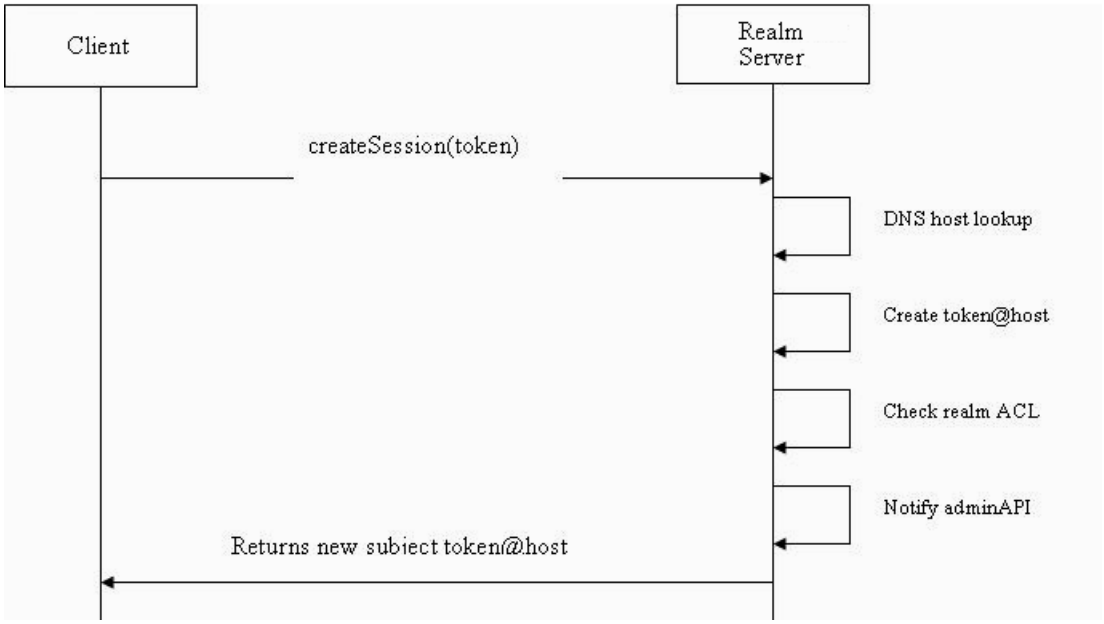
### Enterprise Application Authentication

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.

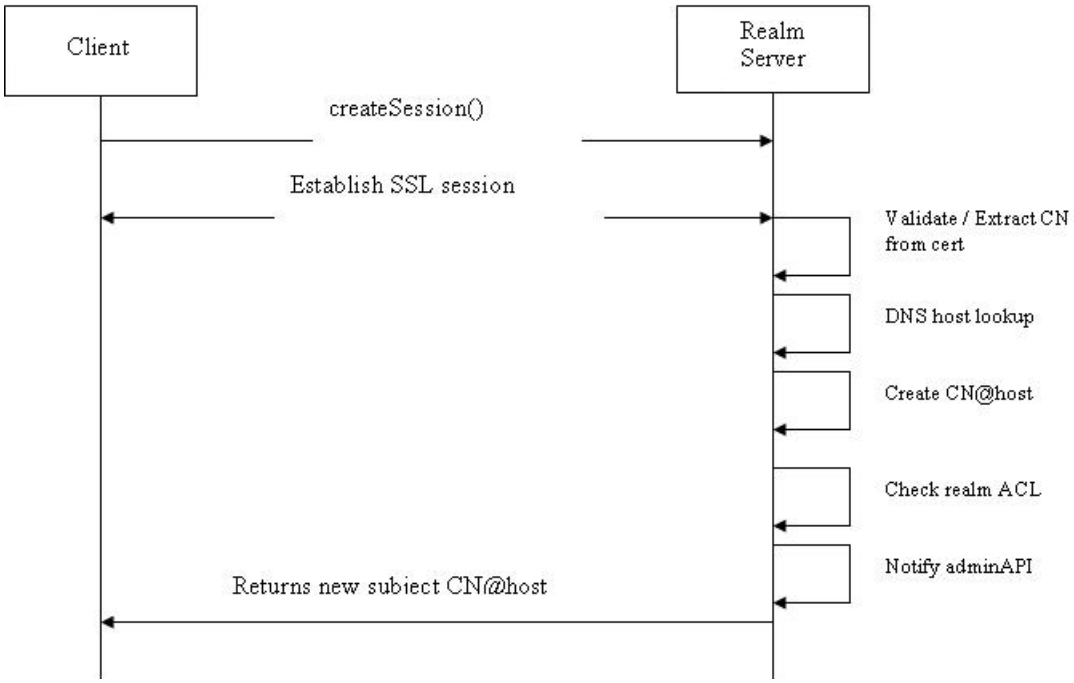
The client API can set the username component, however, the remote host is always set on the server. This stops clients from impersonating users from other hosts. The following diagram illustrates the basic authentication scheme's operation:




## SSL Authentication Scheme

The Universal Messaging Realm server can be configured to perform Client Certificate authorization 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 File plugin, the REST 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 initialize as shown below. This notifies the default authenticator that passwords are not encrypted so on the first load it will encrypt them, remove the initialize 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:

- “Name”: varchar
- “Password”: varchar
- “Rights”: int
- “Home”: 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 and admin APIs for Java (nsp/nsps/nhp/nhps, native and JMS) have overloaded variants that accept username/password credentials which are then supplied to the Universal Messaging server.

The Universal Messaging server enables those credentials to be authenticated against pluggable Directory backends (ranging from LDAP to flat files) or by means of JAAS-based methods (based on user-configurable pluggable modules).

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

The 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 Server_Common.conf configuration file.

**Note:**
Basic authentication, supplying a username and password, is only supported for Java and .NET clients.

### Client-side Authentication

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 for Java clients).

The client API is controlled by three main environment variables/system properties:

- *Nirvana.auth.client.jaaskey*

  **Note:**
  The client-side authentication with JAAS has been deprecated in Universal Messaging v10.7 and will be removed from the product in a future release.

  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.

  The login configuration file's pathname is specified by the usual JAAS system property, `java.security.auth.login.config`.

  The Universal Messaging 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 Universal Messaging administrators, 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, CRAM-MD5 and DIGEST-MD5.

The mechanism defaults to PLAIN if this system property is not set, and the usual SASL trade-offs apply. PLAIN transmits the user password in plain text, so it is advisable to only use it over an SSL connection. On the other hand, CRAM-MD5 and DIGEST-MD5 do not transmit the password in plain text so are more appropriate for general connections.

- *Nirvana.sasl.client.enablePrehash*

  This specifies whether to prehash the supplied password when using the CRAM-MD5 or DIGEST-MD5 mechanisms. It may be set to `true` or `false`. This should be set to `true` only when the server is using the fSAGInternalUserRepositoryAdapter to store client credentials, otherwise CRAM-MD5 and DIGEST-MD5 authentication will fail. If `Nirvana.sasl.client.enablePrehash` is not set, then the value defaults to `false` and prehashing is not enabled.

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.

**Note:**
Basic authentication, supplying a username and password, is only supported for Java and .NET clients.

## Enabling Reverse Hostname Resolution for localhost

You can use the `Nirvana.sasl.client.localhostResolve` property to control whether the reverse hostname resolution for localhost is enabled on the client. Valid values:

- `true` (default) - enables reverse hostname resolution.
- `false` - disables reverse hostname resolution. Set to `false` if you experience SASL-related client connection exceptions over the Universal Messaging Socket Protocol (nsp).

**Important:**
If you set this property to `false`, you must also set the `Nirvana.sasl.server.localhostResolve` property to `false`.

## Server-side Authentication

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 do not supply a username and password are rejected.

The following users are exempt from mandatory authentication:

- The super-user on localhost to prevent being locked out.
- A set of users in a .txt file specified in the optional JVM property `-DNirvana.auth.exempt=<path_to_file>`. You list the users in the exempt file one per line in ACL-style notation, for example, `username1@10.140.2.95`.

  The exempt file supports wildcard-character entries using the * (asterisk) symbol, such as `user@*` and `*@host`. However, `*@*` is not supported because it will allow any user to bypass authentication.

  **Note:**
  The username `*` is a valid username, but if you add such a user to the exempt list, the user will be read as a wildcard. In such cases, Software AG recommends against using the `*` username.

When a client does authenticate, the UM client-server protocol automatically signals the server whether they are 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 Software AG command-line utility `internaluserrepo.bat` (Windows platforms) or `internaluserrepo.sh` (UNIX-based platforms). This utility is located in `<InstallDir>/common/bin`.

This mechanism is the default user repository if the `Nirvana.directory.provider` property is not set.

You can use the following syntax to add a user “myuser1” with a password “mypwd1” to the internal user repository.

```
internaluserrepo.bat -f <path_to_users.txt> -c -p mypwd1 myuser1
```

The location of the file containing the internal user repository is given by the system property `Nirvana.auth.sagrepo.path`, and would default to `./users.txt` (relative to the runtime directory of the Universal Messaging server), but the `Server_Common.conf` file shipped with Universal Messaging overrides this as `../users.txt`, locating it in the same `<InstallDir>`/UniversalMessaging/`server/<InstanceName>` directory as the `licence.xml` file.

The entry in `Server_Common.conf` looks like this:

```
wrapper.java.additional.18=-DNirvana.auth.sagrepo.path=../users.txt
```

The `Server_Common.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 Universal Messaging instance.

You should also set up the following configurations in the `Server_Common.conf` file, to ensure that basic authentication is activated and mandatory:

```
wrapper.java.additional.19=-DNirvana.auth.mandatory=Y
   // This makes basic authentication mandatory.
   // Calls not supplying the credentials in this case will be rejected.
wrapper.java.additional.20=-DNirvana.auth.enabled=Y
   // This enables the basic authentication feature,
   // but it is not mandatory to supply credentials.
```

If you add users to the internal user repository, you should also set up Universal Messaging server ACLs for these users, specifying e.g. myuser1@hostname, and give appropriate or full control.

**Note:**

- To activate any changes you make, you need to restart the Universal Messaging server.
- More details about the utility `internaluserrepo` and its options are available in the Software AG document *Command Central Help*.

### 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.

### Reverse Hostname Resolution for localhost

You can use the `Nirvana.sasl.server.localhostResolve` property to control whether the reverse hostname resolution for localhost is enabled on the server instance. Configure this property in the Server_Common.conf file. Valid values:

- `true` (default) - enables reverse hostname resolution.
- `false` - disables reverse hostname resolution. Set to `false` if you experience SASL-related client connection exceptions over the Universal Messaging Socket Protocol (nsp).

**Important:**
If you set this property to `false`, you must also set the `Nirvana.sasl.client.localhostResolve` property to `false`.

## Server JAAS Authentication with Software AG Security infrastructure component

Universal Messaging can use the Software AG Security Infrastructure component (SIN) to provide server JAAS authentication capabilities. The SIN component provides a variety of options for using different authentication back-ends and implementing flexible authentication scenarios.

Full details of the Software AG Security Infrastructure component are available in the Software AG documentation web site in the document *Software AG Infrastructure Administrator's Guide* . See the description of the predefined login modules in that guide for details of setting up authentication using JAAS login modules and a description of the parameters available for configuring the login modules.

## Enabling JAAS authentication

By default the Universal Messaging server is configured to use a JAAS context named `UM-Default`. This JAAS context is pre-configured to use the Software AG default internal user repository as an authentication back-end.

The JAAS context `UM-Default`, as well as JAAS contexts for LDAP, X.509 and others, are all defined in a file jaas.conf, that resides under the server's `bin` directory at `<InstallDir>/UniversalMessaging/server/<InstanceName>/bin`.

The server configuration file `Server_Common.conf` (again under the server's `bin` directory) defines the path to the JAAS configuration file as the value of the system property `java.security.auth.login.config`. The name of the JAAS context to be used for authentication is configured through the system property `Nirvana.auth.server.jaaskey`.

**Note:**
When the property `Nirvana.auth.server.jaaskey` is present, the server will use JAAS authentication, overriding any configured legacy directory-based authenticators.

The default values that Server_Common.conf defines for its related Universal Messaging server instance are:

```
-DNirvana.auth.enabled=N
-DNirvana.auth.mandatory=N
-DNirvana.auth.server.jaaskey=UM-Default
-Djava.security.auth.login.config=jaas.conf
```

**Important:**
In order to enable the server authentication, you need to switch the flag `Nirvana.auth.enabled` to `Y`. Additionally, in order to make authentication mandatory, you need to switch the flag `Nirvana.auth.mandatory` to `Y`. The changes will take effect at the next restart of the Universal Messaging server.

## SIN JAAS login modules

The supported SIN JAAS login modules are:

1. **Internal user repository login module**

   The SIN internal user repository module (`com.softwareag.security.jaas.login.internal.InternalLoginModule`) provides a mechanism to authenticate a user by a username and a password against a user database file. Every Software AG installation would deliver a default user database file called users.txt under the `<InstallDir>/common/conf` directory. A newly created Universal Messaging realm server will come readily configured for authentication using the default user database file through the `UM-default` JAAS context listed in the jaas.conf file under the realm server `bin` directory.

2. **LDAP login module**

   The SIN LDAP login module (`com.softwareag.security.sin.is.ldap.lm.LDAPLoginModule`) provides a mechanism to authenticate a user by a username and a password against an LDAP

server. The login module offers various options for using an LDAP or an Active Directory server in different authentication scenarios, some of which we will cover in the JAAS authentication templates section below.

3. **X.509 certificate login module**

   The SIN X.509 certificate login module (`com.softwareag.security.jaas.login.modules.X509CertificateLoginModule`) provides a mechanism to authenticate a user with a client X.509 certificate chain. This module allows users to access a Universal Messaging server that requires authentication over a SSL/TLS enabled interface with a client certificate.

## SIN authentication templates for JAAS

Universal Messaging installation delivers a JAAS configuration file jaas.conf that comes with a pre-configured `UM-Default` JAAS context and templates for several other authentication scenarios. The `UM-Default` is selected by default due to the setting of the property `Nirvana.auth.server.jaaskey` in the Server_Common.conf file, as described above.

### 1. Internal user repository

The `UM-Default` JAAS context sets up authentication using the internal user repository that is stored in `users.txt`:

```
/*
 * Authentication with SAG internal user repository - by default the user database
file is located under
 * <INSTALL_DIR>/common/conf/users.txt
 * in a standard Software AG installation. Depending on how the product is started
one may need to
 * reconfigure the path to the database file (using an absolute path)
 */
UM-Default {
  com.softwareag.security.jaas.login.internal.InternalLoginModule sufficient
        template_section=INTERNAL
        internalRepository="../../../../common/conf/users.txt";
};
```

The file users.txt is the internal user repository to be used for authentication when clients attach to the Universal Messaging server. It is located in `<InstallDir>/common/conf`. If you have multiple Universal Messaging servers, you can use the same users.txt file to unify client authentication for all of the servers.

You might wish to vary the client authentication depending on the Universal Messaging server being used. For example, you might want to define username USER1 with password PASSWORD1 on the first server, but on the second server you want to define the same username USER1 with a different password PASSWORD2. To do this, you will need two users.txt files, one for each server, and change the pathname `"../../../../common/conf/"` in each server's jaas.conf file to the path where the server-specific users.txt file is located. You can also provide custom names instead of the default users.txt.

To update the username/password definitions in each users.txt file as required, use the `internaluserrepo.bat` or `internaluserrepo.sh` script in `<InstallDir>/common/bin`. For example,

to add a user with password "mypwd1" and username "myuser1", run this command on the command line:

```
internaluserrepo.bat -f <path_to_users.txt> -c -p mypwd1 myuser1
```

To get instructions on how to create and delete usernames/passwords, as well as how to display a list of existing usernames, run the script with the "-h" option:

```
internaluserrepo.bat -h
```

**2. LDAP Lookup**

The LDAP look up method involves logging on to the LDAP server either anonymously (if allowed) or by using a predefined principal, and performing a lookup for a user entry matching a certain attribute. Finally the SIN-specific LDAP login module will attempt to bind with the retrieved user's distinguished name and the supplied password. The following template configuration is provided where you should fill in the required properties:

```
/*
 * Authentication with an LDAP server by looking up users based on an attribute.
 * The login module will use the configured principal (parameter "prin") to bind
 * and perform a lookup.
 * The principal can be omitted if the LDAP server is configured to allow
 * anonymous bind and lookup.
 * Afterwards the login module will search for an entry under
 * the specified root DN with the attribute
 * configured through the "uidprop" matching the supplied username.
 * The "uidprop" parameter can be omitted, its default value is "cn".
 */
UM-LDAP-Lookup {
    com.softwareag.security.sin.is.ldap.lm.LDAPLoginModule sufficient
       url="<LDAP URL>"
       prin="<principal complete DN>"
       cred="<principal password>"
       userrootdn="<root DN for searching users>"
       uidprop="<user ID property name>";
};
```

**3. Active Directory direct bind**

With active directory the SIN-specific LDAP login module offers the possibility for a direct bind with the supplied username and password, using the following configuration template where the user should fill in the active directory URL.

```
/*
 * Authentication with an Active Directory server by binding directly with
 * the supplied credentials.
 * Active Directory allows for a direct bind with a principal in the
 * format "domainPrefix\user".
 */
UM-AD-DirectBind {
  com.softwareag.security.sin.is.ldap.lm.LDAPLoginModule sufficient
       url="<LDAP URL>"
       noPrinIsAnonymous=false
       useFQDNForAuth=true;
};
```

**4. LDAP direct bind**

It is also possible to attempt a direct bind against an LDAP server where the bind DN will be constructed by the SIN-specific JAAS login module using a prefix and a suffix, with the supplied username in-between. In this case the user is required to configure the URL of the LDAP server, as well as the prefix and the suffix to be used for the construction of the bind DN:

```
/*
 * Authentication with an LDAP server by building a complete DN and binding directly.
 * The bind DN will be composed by prepending the supplied username with the
 * value of the "dnprefix" parameter and appending the value
 * of the "dnsuffix" parameter.
 */
UM-LDAP-DirectBind-Affixes {
    com.softwareag.security.sin.is.ldap.lm.LDAPLoginModule sufficient
       url="<LDAP URL>"
       noPrinIsAnonymous=false
       useaf=true
       dnprefix="<Bind DN prefix>"
       dnsuffix="<Bind DN suffix>";
};
```

**5. Combination of any of the above**

JAAS is extremely flexible and allows for using a combination of different back-ends, such as, several LDAP servers and/or repositories. For example it is possible to configure LDAP authentication with an internal user repository as a fallback mechanism:

```
/*
 * Authentication with an LDAP server through a user lookup, with an
 * internal user repository as a fallback mechanism.
 * This configuration would first try to log in against the LDAP if that fails
 * would try authentication against the internal user repository.
 * Similarly one could use several LDAP servers or
 * combine more authentication backends.
 */
UM-Combined {
    com.softwareag.security.sin.is.ldap.lm.LDAPLoginModule sufficient
       url="<LDAP URL>"
       prin="<principal complete DN>"
       cred="<principal password>"
       userrootdn="<root DN for searching users>"
       uidprop="<user ID property name>";
    com.softwareag.security.jaas.login.internal.InternalLoginModule sufficient
        template_section=INTERNAL
        internalRepository="../../../../common/conf/users.txt";
};
```

**6. Combining X.509 client certificate with a username/password authentication**

Authentication is configured globally for a Universal Messaging realm server, which means that if a user connects over an SSL/TLS interface that requires a X.509 client certificate, the user would still have to undergo an authentication check in order to establish a viable session. This is where the SIN X.509 certificate login module comes in handy, as it would allow users to authenticate with a presented client certificate over an SSL/TLS interface, or alternatively, present a username and a password credentials if connecting over an interface that does not require a client certificate. This can be achieved using the following template of the X.509 client certificate login module in combination with either an internal user repository, or an LDAP login module (or both).

```
/*
 * Authentication with a X.509 certificate chain, and a username/password, as a
 * fallback mechanism. This configuration would first try to log in using an X.509
 * certificate chain (if present), and if that fails, it would attempt
 * authentication against an internal user repository. A certificate chain would
 * only be present if the user connected through a TLS/SSL enabled interface that
 * required a client certificate, and this login context will give precedence to
 * the X.509 certificate chain credential.
 *
 * The same scenario can be implemented with an LDAP server by replacing the
 * internal user repository login module with an LDAP login module configuration.
 *
 * This context would allow that in a server that has authentication enforced
 * globally, users can connect with a username and a password through insecure
 * interfaces (or TLS/SSL enabled interfaces using server certificates only), and
 * using an X.509 certificate over an TLS/SSL interface that requires a client
 * certificate.
*/
UM-X509-UsernamePassword {
    com.softwareag.security.jaas.login.modules.X509CertificateLoginModule sufficient
        truststore_url="<truststore file URL>"
        truststore_password="<truststore password>"
        create_user_principal=true;
  com.softwareag.security.jaas.login.internal.InternalLoginModule sufficient
        template_section=INTERNAL
        internalRepository="../../../../common/conf/users.txt";
};
```

To use any custom JAAS context, or any of the predefined templates, you would need to modify the value of the `Nirvana.auth.server.jaaskey` system property inside the Server_Common.conf file in the server's `bin` directory, pointing it to the context to be used. At the moment the context name is globally defined for the entire server.

**Note:**
Since the jaas.conf file is edited manually and requires certain knowledge of the default JAAS configuration file format, there is always the possibility of a syntactic error. Such errors during parsing of the file will be printed to the UMRealmService.log file which resides under the server's `bin` directory.

## 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 99).

## Access Control Lists (ACLs)

**Note:**
ACLs provide authorization for your application, but authentication is also required for robust security. To achieve robust security you should use either LDAP or SSL client-side certificate authentication in addition to ACLs. If authentication is a concern, refer to the LDAP description in the section "Using SASL " on page 88, or the SSL client certificate authentication description in the section "Using SSL" on page 102.

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 Channels - Can add/delete channels on this realm
- Access The Realm - Can currently 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 - Can perform cluster operations, such as create, delete or modify cluster information

- Manage DataGroups - Can add, delete and manage any Data Group on the realm
- Publish to DataGroups - Can publish to any Data Group on the realm
- Own DataGroups - Can take ownership of any Data Group 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 to 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 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

### 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.

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 |

If a user is matched by more than one wildcard ACL, the user receives the cumulative permissions of all of these wildcard ACLs. If a user is matched by a combination of wildcard ACLs and non-wildcard ACLs, the effect of the permissions is also cumulative.

The subject *@* is a special case. It 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 the realm configuration parameter `OverrideEveryoneUser` is set to false (this is the default), then *@* will override any more specific ACL entries which match a given user. If this parameter is set to true, then *@* will be overridden by any more specific ACL entries which match a given user. `OverrideEveryoneUser` is one of the parameters in the Global Values realm configuration group; see the relevant section of the Enterprise Manager documentation for further information.

# Using SSL

## 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 its 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.

## SSL Encryption

Universal Messaging 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 client's certificate CN attribute.

Universal Messaging's support for SSL provides 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 102).

## Server SSL Configuration

By default, the Universal Messaging server supports the following SSL/TLS protocols for secure inbound and outbound communication:

- TLS 1
- TLS 1.1
- TLS 1.2
- TLS 1.3

You can configure the SSLProtocols server parameter to define which TLS versions the server is allowed to use. If the client attempts to use a version that is not defined in SSLProtocols, the connection will fail. The value of the parameter is a comma-separated list of TLS protocols. The default value is `TLSv1,TLSv1.1,TLSv1.2`.

You configure the parameter in the Server_Common.conf file in the *Software AG_directory* \UniversalMessaging\server*instance_name*\bin directory. For example, to specify only TLS 1.2 and 1.3 as allowed protocols, type the following:

```
wrapper.java.additional.n=-DSSLProtocols=TLSv1.2,TLSv1.3
```

where *n* is a unique positive integer.

## Client SSL Configuration

This section describes how to use SSL/TLS in your Universal Messaging Java 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.

Once you have created an SSL enabled interface for your realm (see the description of how to do this in the section "Creating an SSL network interface" of the Enterprise Manager documentation), you need to ensure that your client application passes the required SSL properties either via custom setters on the session attributes or via system properties used by your JSSE-enabled JVM.

Universal Messaging supports client SSL certificates in .jks (Java keystore) and PKCS12 format.

The Universal Messaging download contains some sample Java keystore files that will be used in this example.

The first such keystore is the client keystore, called `client.jks`, which can be found in your installation directory, under the `client/<umserver>/bin` directory. The second is the client truststore called `nirvanacacerts.jks`, which is also located in the `client/<umserver>/bin` directory. On Windows, these files exist once you run the '**Server/<*umserver*>/Create Demo SSL Certificates**' shortcut.

If you would like to add your own client certificates please see the section "How to generate certificates" of the Enterprise Manager documentation.

### Using Custom SSL Properties

Using the sample keystores, you can use the following steps to set the SSL properties on a session.

1. Construct your session attributes object:

   ```
   nSessionAttributes sessionAttributes = new nSessionAttributes(rname);
   ```

2. Set your required SSL attributes using these methods:

   ```
   attribs.setKeystore(String keyStorePath, String keyStorePassword)
   attribs.setKeystore(String keyStorePath, String keyStorePassword,
       String certificateAlias);
       // Certificate alias specifies which certificate the client can use
       // if client validation is enabled on the interface being connected to
   attribs.setTruststore(String trustStorePath, String trustStorePassword);
   ```

   So your session attributes should look like this:

```
attribs.setKeystore(%INSTALLDIR%\client\<umserver>\bin\client.jks, password);
attribs.setTruststore(%INSTALLDIR%\client\<umserver>\bin\nirvanacacerts.jks,
 password);
```

In addition to this you are also able to set a list of enabled cipher strings the client can use and the SSL protocol:

```
attribs.setEnabledCiphers(String[] enabledCiphers);
attribs.setSSLProtocol("TLSv1.2,TLSv1.1");
```

3. Construct your session with the constructed session attributes:

```
nSession session = nSessionFactory.create(attribs);
session.init();
```

## Using JSSE SSL System Properties

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:

**Important:**
The CKEYSTORE, CKEYSTOREPASSWD, CAKEYSTORE, and CAKEYSTOREPASSWD system properties are deprecated.

```
-DCKEYSTORE=%INSTALLDIR%\client\<umserver>\bin\client.jks
-DCKEYSTOREPASSWD=password
-DCAKEYSTORE=%INSTALLDIR%\client\<umserver>\bin\nirvanacacerts.jks
-DCAKEYSTOREPASSWD=password
```

where:

CKEYSTORE (deprecated) is the client keystore location

CKEYSTOREPASSWD (deprecated) is the password for the client keystore

CAKEYSTORE (deprecated) is the CA keystore file location

CAKEYSTOREPASSWD (deprecated) 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, set the following system properties:

```
-Djavax.net.ssl.keyStore=%INSTALLDIR%\client\<umserver>\bin\client.jks
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=%INSTALLDIR%\client\<umserver>\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.

### Using the Universal Messaging Client System Properties for Secure Communication

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

Universal Messaging supports client SSL certificates in .jks (Java keystore) and PKCS12 format.

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

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

where:

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

```
-Dcom.softwareag.um.client.ssl.ssl_protocol=TLSv1.2
```

### 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.

## JMS Client SSL Configuration

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

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

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

### Custom SSL Properties

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

Setting the SSL Attributes on the JNDI Context

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

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

Setting the SSL Attributes on the Connection Factory

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

```
connectionFactory.setProperties(env);
```

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

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

```
connectionFactory.setSSLStores(String keyStorePath, String keyStorePass,
    String trustStorePath, String trustStorePass);
connectionFactory.setSSLStores(String keyStorePath, String keyStorePass,
    String certificateAlias, String trustStorePath, String trustStorePass);
connectionFactory.setSSLProtocol(String protocol);
connectionFactory.setSSLEnabledCiphers(String[] enabledCiphers);
Connection con = connectionFactory.createConnection();
```

Setting the SSL Attributes on the Connection

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

## JSSE SSL System Properties

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

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

where :

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

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

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

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

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

### Using Universal Messaging Client System Properties

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

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

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

where:

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

## Setting up and using FIPS

Federal Information Processing Standards (FIPS) provides standards for information processing for use within the Federal government.

Many government and financial organizations require their software to be FIPS 140-2 compliant, which follows the current standards and guidelines for cryptographic information processing.

In Universal Messaging, FIPS capabilities can be enabled through the use of Mozilla NSS. NSS is an open source security library that has received FIPS validation/certification for both its FIPS and cryptographic modes. Further information can be found at the Mozilla website.

**Note:**
Universal Messaging itself is not FIPS 140 certified.

The NSS library is not bundled in the Universal Messaging product. Details of how to download and build your own copy of NSS for your respective platform can be found at the Mozilla website.

Once you have built the library, the JVM will need to know where to load this library from via environment variable (for example, LD_LIBRARY_PATH, or equivalent). Only the Oracle® JVM is supported at this time.

To enable FIPS in Universal Messaging, the property `enableFIPS` must be set on either the client and/or server before either one is started. For example:

```
-DenableFIPS=<NSS mode>
```

where <*NSS mode*> can be one of the following:

- `nssfips`
- `nsscrypto`

The biggest difference between these modes is that `nssfips` uses an NSS database for all key management purposes, while `nsscrypto` uses just the cryptographic algorithms and functions of NSS, while maintaining native JKS support. And, at the time of this writing, `nssfips` does not yet support TLS 1.2.

Storing certificates into an NSS database requires you to use certain NSS tools; documenting these tools is beyond the scope of this document (see the Mozilla website for details). However, an example shell script is provided below to get you started:

```
#!/bin/csh
setenv DYLD_LIBRARY_PATH /Users/test/Downloads/nss-3.20/dist/Darwin13.4.0_64_OPT.OBJ/lib

setenv PATH /Users/test/test/nss-3.20/dist/Darwin13.4.0_64_OPT.OBJ/bin:$PATH
echo Make nssdb directory
mkdir ~/nssdb
modutil -create -dbdir ~/nssdb
echo Enable FIPS mode on nssdb
modutil -fips true -dbdir ~/nssdb
# Password12345!
echo Set password for nssdb
modutil -changepw "NSS FIPS 140-2 Certificate DB" -dbdir ~/nssdb
# DER format/PKCS#7
echo Export alias from CA JKS to CRT
# Export only public keys
keytool -export -alias "localhost,127.0.0.1" -keystore ./nirvanacacerts.jks \
        -file public.crt
echo Import CRT into nssdb
certutil -A -d ~/nssdb -t "TCu,,TCu" -n "Test CA 1" -i ./public.crt
echo Convert client JKS to PKCS12
keytool -importkeystore -srckeystore ./client.jks -destkeystore ./client.p12 \
        -deststoretype PKCS12
echo Convert server JKS to PKCS12
keytool -importkeystore -srckeystore ./server.jks -destkeystore ./server.p12 \
        -deststoretype PKCS12
echo Import client PKCS12 into nssdb
pk12util -i ./client.p12 -n client -d ~/nssdb
echo Import server PKCS12 into nssdb
pk12util -i ./server.p12 -n CA -d ~/nssdb
```

The above example assumes self signed certificates. Certificate aliases are NOT supported when NSS is run in FIPS mode, so only 1 certificate/trusted certificate pair should be imported into an NSS database. It is recommended that you create both a client and server NSS database to separate certificates from one another. Each NSS instance/library has a single NSS database available for key management (i.e., the relationship is 1:1).

Both modes use the Oracle JVM support for NSS; specifically, this means using the Oracle PKCS11 provider to access functionality within the NSS library. Again, no other JVM is currently supported. Only 1 NSS instance/library can be loaded per JVM via the PKCS11 provider. Universal Messaging automatically manages the loading of the PKCS11 provider and NSS library for you, so you do not have to do this yourself (assuming that the LD_LIBRARY_PATH is set properly for the NSS library).

NSS uses separate configuration files to determine the mode that it should run in (among other things). For example, to run NSS in FIPS mode, create the following configuration file as a text file:

```
name = <unique name>
nssLibraryDirectory = <path to location of NSS library>
nssSecmodDirectory = <path to location of NSS database>
nssModule = fips
```

Example:

```
name = CNSS
nssLibraryDirectory = /Users/test/Downloads/nss-3.20/dist/Darwin13.4.0_64_OPT.OBJ/lib

nssSecmodDirectory = /Users/test/nssdb
nssModule = fips
```

The key "name" is important, as it signifies the NSS name which uniquely identifies the NSS instance. The rest should be self explanatory. If you set up an NSS configuration file for FIPS mode, then `-DenableFIPS` should be set to `nssfips`.

To run NSS in crypto mode, create a configuration file as follows:

```
name = <unique name>
nssLibraryDirectory = <path to location of NSS library>
nssModule = crypto
attributes = compatibility
nssDbMode = noDb
```

Example:

```
name = CNSS
nssLibraryDirectory = /Users/test/Downloads/nss-3.20/dist/Darwin13.4.0_64_OPT.OBJ/lib

nssModule = crypto
attributes = compatibility
nssDbMode = noDb
```

If you set up an NSS configuration file for crypto mode, then `-DenableFIPS` should be set to `nsscrypto`.

Note that NSS configuration files can only have 1 set of configuration parameters in them; in other words, you cannot have multiple configurations within a single file.

Once the configuration file and name have been set up, you'll need to specify them either via the Universal Messaging APIs, or via the Enterprise Manager (if configuring a Universal Messaging server NHPS or NSPS interface). For the Universal Messaging APIs, see the appropriate Java documentation for:

```
com.pcbsys.nirvana.client.nSessionAttributes.getPKCS11NSSConfigFile
com.pcbsys.nirvana.client.nSessionAttributes.setPKCS11NSSConfigFile
com.pcbsys.nirvana.client.nSessionAttributes.getPKCS11NSSName
com.pcbsys.nirvana.client.nSessionAttributes.setPKCS11NSSName
com.pcbsys.nirvana.nAdminAPI.nSSLInterface.getPKCS11NSSConfigFile
com.pcbsys.nirvana.nAdminAPI.nSSLInterface.setPKCS11NSSConfigFile
com.pcbsys.nirvana.nAdminAPI.nSSLInterface.getPKCS11NSSName
com.pcbsys.nirvana.nAdminAPI.nSSLInterface.setPKCS11NSSName
com.pcbsys.nirvana.nAdminAPI.nHTTPSInterface.getPKCS11NSSConfigFile
com.pcbsys.nirvana.nAdminAPI.nHTTPSInterface.setPKCS11NSSConfigFile
com.pcbsys.nirvana.nAdminAPI.nHTTPSInterface.getPKCS11NSSName
com.pcbsys.nirvana.nAdminAPI.nHTTPSInterface.setPKCS11NSSName
com.pcbsys.foundation.drivers.configuration.fSSLConfig
com.pcbsys.foundation.drivers.configuration.fHTTPSConfig
```

While running NSS in FIPS mode, the password to the NSS database must be specified via nSessionAttributes.setKeystore (keystore/alias would be set to null or is unused), or via nSSLInterface/nHTTPSInterface.setKeyStorePassword. While running NSS in crypto mode, use the APIs as you normally would for JKS keystores/truststores. Set the PKCS11 NSS configuration file and name using the appropriate nSessionAttributes/nSSLInterface/nHTTPSInterface.setPKCS11NSS* methods.

For configuring an SSL/HTTPS interface in the Enterprise Manager, specify a key store password, PKCS11 NSS configuration file and name for NSS FIPS mode; all other fields can be left blank. Any combination of SSL and/or HTTPS interfaces configured while in NSS FIPS mode will share the same NSS database (since only 1 NSS instance/database is available per JVM). For NSS crypto, configure the interfaces as you normally would for JKS keystores/truststores.

To recap, refer to the following NSS checklist when enabling FIPS for UM:

1. Shut down any Universal Messaging servers and/or clients that you want to configure for FIPS.
2. Download the Mozilla NSS open source library, and build it for your respective platform (see the Mozilla website for details).
3. Determine whether you want to run in NSS FIPS or crypto mode.
4. If you'll be running in FIPS mode, create an NSS FIPS configuration file. If you'll be running in crypto mode, create an NSS crypto configuration file.
5. If you'll be running in FIPS mode, create an NSS database, and store the appropriate certificates in it (both client and server). If you'll be running in crypto mode, create/use normal JKS keystores/truststores.
6. Set up LD_LIBRARY_PATH (or equivalent, depending on your platform) to point to the location of the NSS binary distribution. This can be done in either a shell, command line, shell/batch script or configuration file.
7. Write any API programs that are FIPS enabled (if applicable).

8. For client and/or server, set `-DenableFIPS=nssfips` in a command line, shell/batch script or configuration file if you'll be running in FIPS mode. Set `-DenableFIPS=nsscrypto` if you'll be running in crypto mode (again, for client and/or server).
9. Restart the Universal Messaging server(s), and configure SSL/HTTPS interfaces with FIPS as needed. Restart the Universal Messaging client(s).

# 5 Performance, Scalability and Resilience


- Overview of Performance, Scalabilty and Resilience ..... 116
- Performance Tuning ..... 117
- Multicast: An Overview ..... 126
- Shared Memory (SHM) ..... 129
- Realm Benchmarks ..... 129
- Failover ..... 134
- Connections Scalability ..... 135
- Load Balancing ..... 135
- Horizontal Scalability ..... 137
- Event Fragmentation ..... 143

# Overview of 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 *connection scalability*, which is the ability to support large numbers of concurrent connections; this is 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.

## 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.

# Performance Tuning

## Overview of Tuning

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

Much of the information given in this section 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.

- 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.

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.

## The Realm Server

This page details 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 verbose logging levels can negatively affect the performance of your application. Logging creates extra objects which increases memory usage, and also promotes contention between threads which wish to print to the logger.

The verbosity of the logging can be changed by using the Enterprise Manager. The most verbose log level is 0 and produces the most output. The least verbose log level is 6, which produces very little output. Unless you are attempting to gather logging information related to a particular issue,

it is recommended to use a the log level no more verbose than 5 (Quiet). Particularly demanding applications may wish to reduce the verbosity to logging level 6.

For client side logging please see the relevant API documentation.

### 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 utilize 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

The most aggressive fanout mechanism Universal Messaging provides is called *spin locking*. By default, this 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 available, 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 `Server_Common.conf` file under the `<InstallDir>`/UniversalMessaging`/server/<InstanceName>/bin` directory of your installation, where <*InstanceName*> is the name of the Universal Messaging realm:

```
-DCORE_SPIN=true
```

There are further flags that can be applied to the `Server_Common.conf` file to customize the behavior 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.

## The Java Virtual Machine (JVM)

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 outlines a selection of these JVMs.

Azul Zulu is Software AG's primary test environment for Universal Messaging before the product is released. This JVM is included in the Universal Messaging distribution kit for the Windows, Linux and Solaris platforms. The other JVMs listed below have not been tested to work in all product configurations.

### Azul Zulu®

This JVM is suitable to fulfil most users needs for Universal Messaging.

### Oracle HotSpot™ JVM

This JVM is suitable to fulfil most users needs for Universal Messaging.

### Oracle JRockit JVM

This JVM was made free and publicly available in May 2011. It contains many of the assets from the Oracle HotSpot VM.

### Azul Zing® JVM

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 JVM which may help improve application performance. These settings can be applied to a Universal Messaging Realm Server by editing the `Server_Common.conf` file found under the `<InstallDir>`/UniversalMessaging/server/`<InstanceName>`/bin directory of your installation, where <*InstanceName*> is the name of the Universal Messaging realm.

## 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 `Server_Common.conf` 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, data groups, 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, data groups or queues. Heavy dynamic creation and removal of ACL entries, channels, data groups 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, data groups 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, data groups 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 `Server_Common.conf`:

```
-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.

## Tuning the Linux Operating System

This topic details important operating system and kernel settings that should be considered when optimizing the server. The focus of this topic is geared towards Red Hat Enterprise 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, want 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
```

## Configuring Mapped Memory Regions

`vm.max_map_count` is a Linux kernel parameter that defines the maximum count of mapped memory regions allowed in the system.

When you use Universal Messaging multi-file disk stores, you might need to modify `vm.max_map_count` to prevent out-of-memory issues and increased resource consumption in your system.

With multi-file disk stores, Universal Messaging uses memory mapped files to read and write data with a focus on optimized performance. However, memory mapped files require mapped areas and file descriptors as system resources, which you control using the `vm.max_map_count` and `ulimits` parameters.

Therefore, you might need to calculate the minimum requirements for file descriptors and mapped memory regions per store and adjust `vm.max_map_count` accordingly. For more information about calculating these requirements, see "How to Calculate File Descriptor and Mapped Memory Requirements per Store" on page 284.

## 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.

### Stopping 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 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 Enterprise 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.

## 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.

This section assumes the reader has some knowledge of IP Multicast.

## Comparison of Unicast and Multicast

Universal Messaging delivers 'ultra-low latency' to a large number of connected clients by including IP Multicast functionality for both the delivery of events to Data Group consumers as well as between inter-connected realms within a Universal Messaging cluster.

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

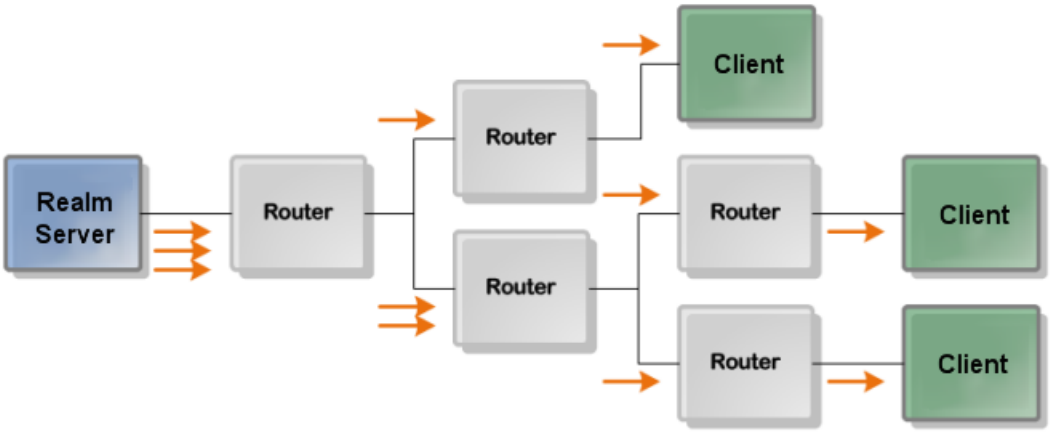



*Figure 1: Universal Messaging in Unicast Mode*

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

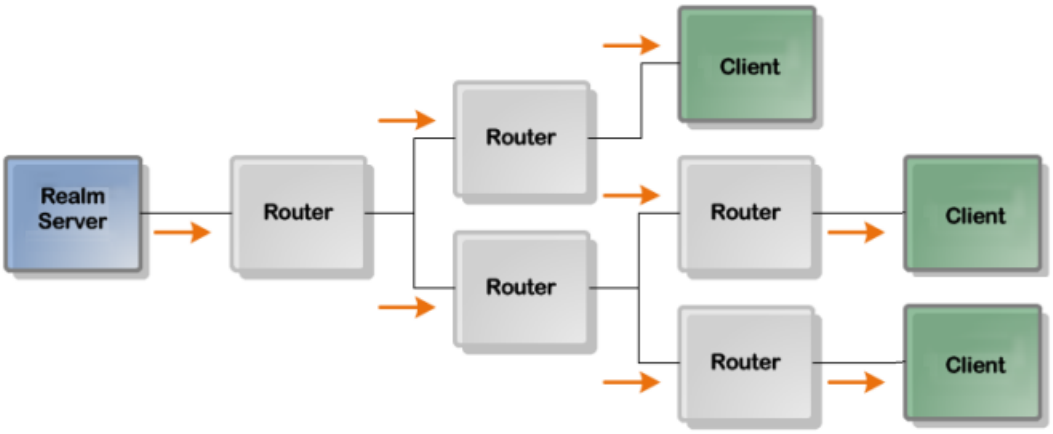



*Figure 2: Universal Messaging in Multicast Mode*

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 data groups. 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 data group 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 data stream 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 data stream.

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.

### Universal Messaging Multicast Architecture

Each Universal Messaging interface that you configure on a Universal Messaging Realm binds to one or all of the available physical Network adapters present on the host machine. In order to successfully configure Multicast on a Universal Messaging Realm you must ensure that you know the IP addresses of each of these network adapters (including virtual addresses if running on a virtual host), and which physical network adapter and its address is capable of supporting IP Multicast. This information allows the correct network adapter to be selected and bound to by the Multicast configuration. Once this information is known, you then need to ensure that the physical network infrastructure including switches and routers can support Multicast. Once validated, the next step is to select an available Multicast address which can be used.

Universal Messaging servers can use IP Multicast either to deliver Data Group events to its consumers or between Universal Messaging realms within a cluster. If you wish to enable IP Multicast delivery to Data Group consumers, you can create a Multicast configuration and select it for use with Data Groups. Once you have configured a Multicast adapter, when you create a Data Group with the enable Multicast flag set, the Universal Messaging realm will automatically begin delivering the events via Multicast when published to that Data Group. The client application requires no extra setup to begin receiving Multicast. When a client Data Stream is added to a Multicast enabled Data Group, the client will transparently receive the information it needs to begin consuming Multicast for that Data Group. The client will at first both consume the Unicast Data Group events, and if Multicast is possible, also consume the Multicast events. When both Unicast and Multicast are in sync after a period of time, the Universal Messaging Server will stop sending Unicast events for that Data Stream to the client. The Universal Messaging server will track whether each client is in fact able to process the Multicast packets and if any client does not successfully acknowledge safe receipt of the Multicast events, it will simply continue to consume the Unicast events.

With this model, the client is able to seamlessly interact with the Universal Messaging server and begin consuming Multicast events with no changes to the Client application required.

# 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 set up 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
- Shared memory drivers are currently not supported on the following configurations:
  - HP-UX systems
  - Solaris with SPARC architecture

# 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

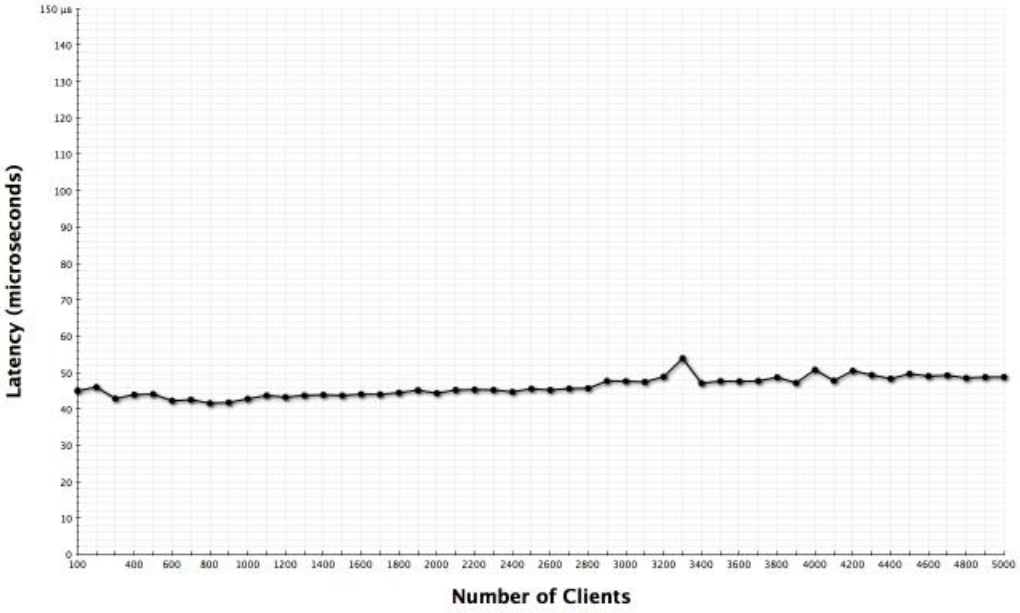
The low and medium rate tests are designed to model traffic which may be typical of non-time critical applications which may instead focus the number of concurrent connections a server can handle.

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 feature that 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.

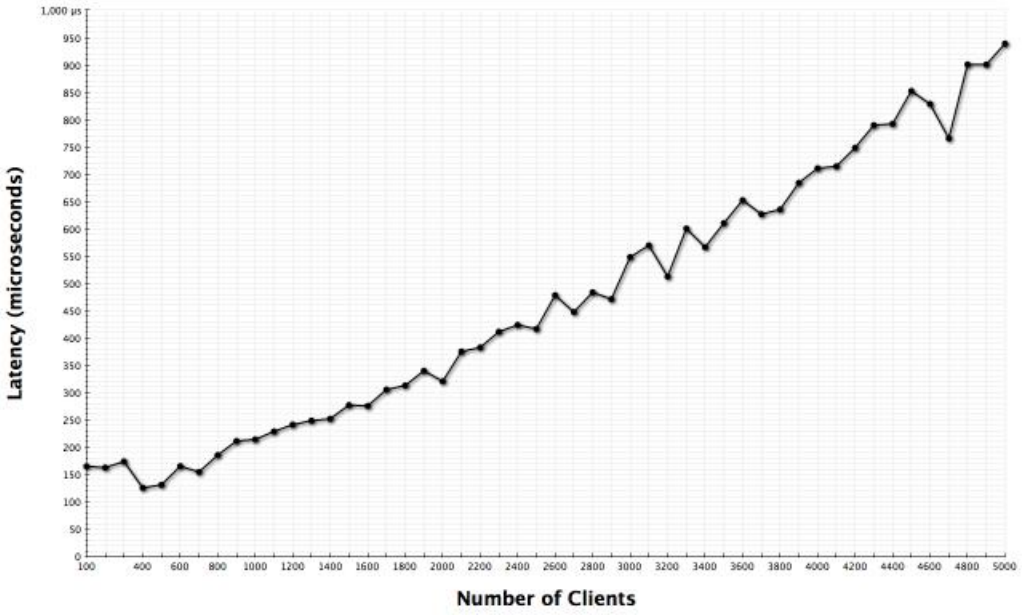
| | |
|---|---|
| DataGroups | 500 |
| Clients | 5,000 |
| Increment Rate | 100 every 60 seconds |
| Subscriptions per Client | 50 |
| Messages per second per Client | 50 |
| Peak Message Rate | 250,000 |




## 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.

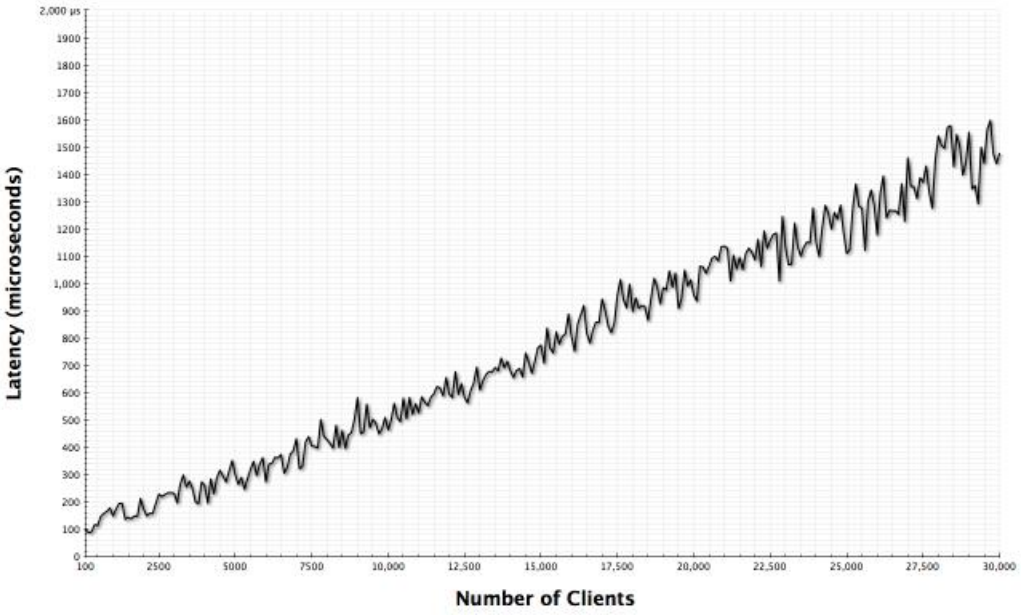
| | |
|---|---|
| DataGroups | 500 |
| Clients | 5,000 |
| Increment Rate | 100 every 60 seconds |
| Subscriptions per Client | 50 |
| Messages per second per Client | 50 |
| Peak Message Rate | 250,000 |




## 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.

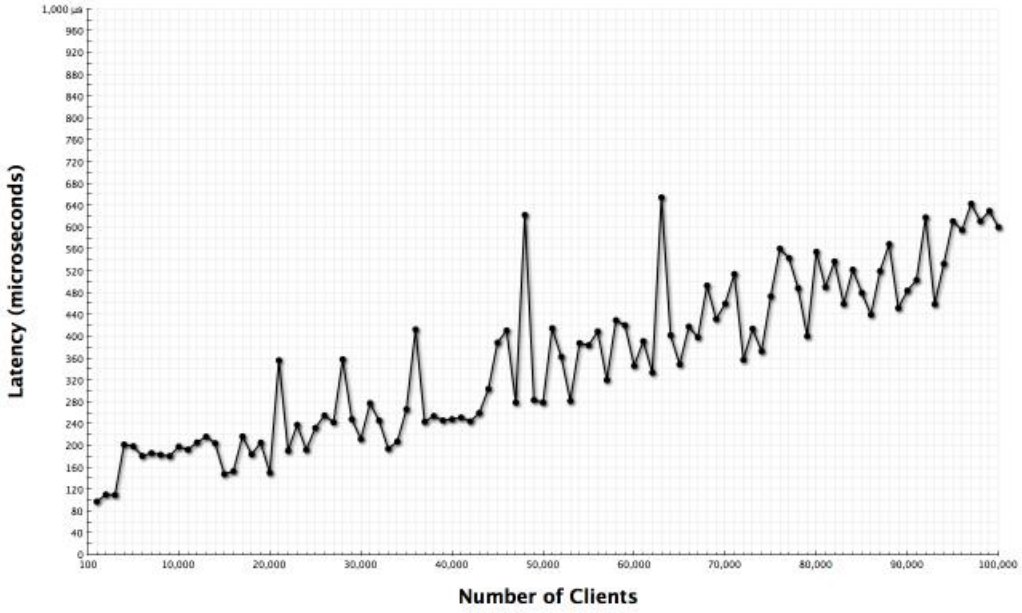
| | |
|---|---|
| DataGroups | 500 |
| Clients | 30,000 |
| Increment Rate | 100 every 60 seconds |
| Subscriptions per Client | 10 |
| Messages per second per Client | 10 |
| Peak Message Rate | 300,000 |




## 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 168) allows for multiple Realms to act as live replicas of each other. This has huge benefits in terms of 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 45) or by using a Universal Messaging Cluster (see "Clusters: An Overview" on page 168).

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.

## Load Balancing

Universal Messaging can work in environments where a third-party load balancer is in use. The main requirement is that the load balancer must support so-called "session stickiness". This means that when a load balancer assigns a client to a server in the cluster when initializing the client session, the load balancer will preferentially assign all subsequent communication from the client to the same server during the same client session.

From a technical point of view, load balancing for Universal Messaging can be used with either a single node or an active/active cluster. Before you choose to use a dedicated third-party load balancer to balance the load between Universal Messaging clients and servers, you should consider the following load balancing features already available in Universal Messaging:

- **Round-robin connections to Universal Messaging servers**

  If a load balancer is needed to force clients to establish connections to multiple Universal Messaging servers instead of just one server, then this feature can be used. Universal Messaging clients support a feature where connections to Universal Messaging servers will be done in a round-robin fashion. You can define the set of available servers using a semicolon(;) separated RNAME list. See the section "Communication Protocols and RNAMEs" on page 22 for details.

- **Horizontal Scalability**

  If a load balancer is needed to improve the event throughput by splitting the Universal Messaging server load to multiple servers, then the Horizontal Scalability feature of Universal Messaging should be considered. It allows a single client session to connect to multiple servers/clusters simultaneously. See the section "Horizontal Scalability" on page 137 for details.

- **Follow The Master: Applicable for active/active Universal Messaging cluster**

  If a load balancer is needed to improve the performance of an active/active Universal Messaging cluster by balancing the connections through all Universal Messaging cluster nodes, this feature should be considered.

  In an active/active cluster, best performance is achieved when client connections are established to the master node (refer to the section "The "Follow the Master" feature" on page 177). Thus, having all clients connected to the master node will normally give better performance than when connections are balanced through all cluster nodes equally.

## Limitations of using a Load Balancer with Active/Active Universal Messaging cluster

The Universal Messaging feature "follow-the-master" will not work in an optimal way if a load balancer is in use. Follow-the-master is an important Universal Messaging setting, as in most cases this is the most performant way to use an active/active cluster.

With a load balancer, when the client is redirected to connect to a slave node and follow-the-master is activated, the slave node will send the address of the master back to the client and will disconnect. Then the client will try to connect to the master, using the master's URL returned by the slave node, bypassing the load balancer. In that case, the client should be able to establish a direct connection to the master, otherwise the connect will always fail.

If the client doesn't have direct access to the cluster nodes and connection should always be done through the load balancer, then follow-the-master should not be set on clients.

To allow the follow-the-master behavior in this case, a possible solution would be to run a separate Universal Messaging Admin API client tool/script on the load balancer itself, whose sole purpose would be to get information from the Universal Messaging cluster about which node is the current master, then re-direct all the client connections to this node. This workaround can also be used by clients which don't support the "follow-the-master" feature.

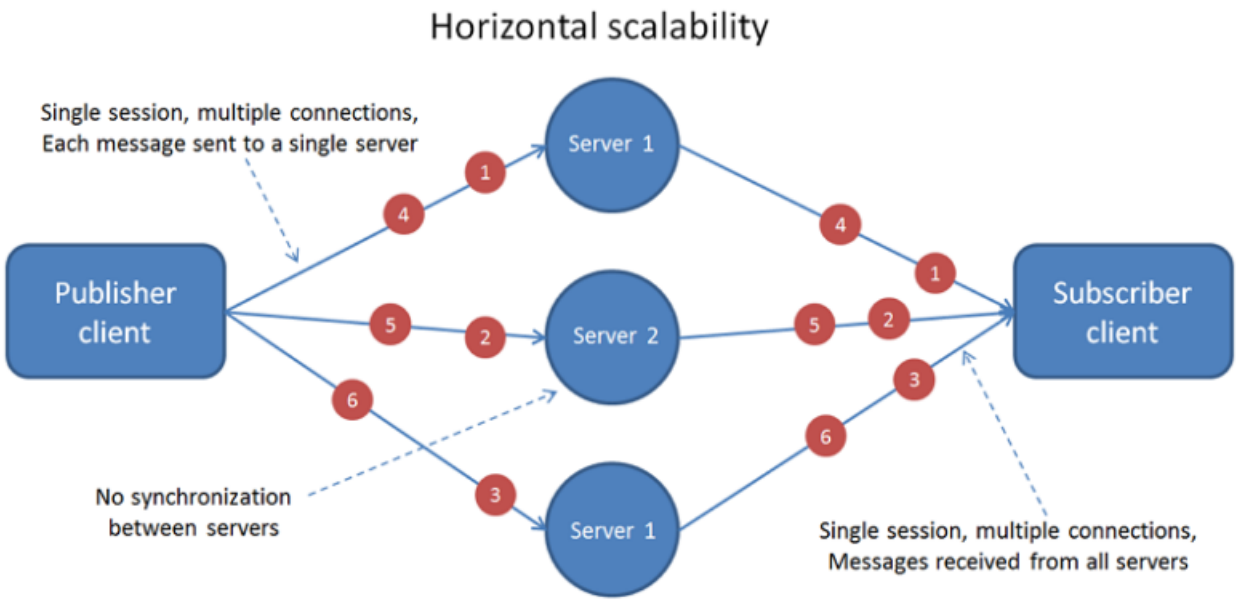
# Horizontal Scalability

## Overview

The *Horizontal Scalability* (HS) feature allows clients to seamlessly publish and consume events from multiple independent realms and clusters using a single connection. This feature is available for both the Universal Messaging native API for Java and the Universal Messaging API for JMS. It is enabled by using the horizontal scalability URL syntax.

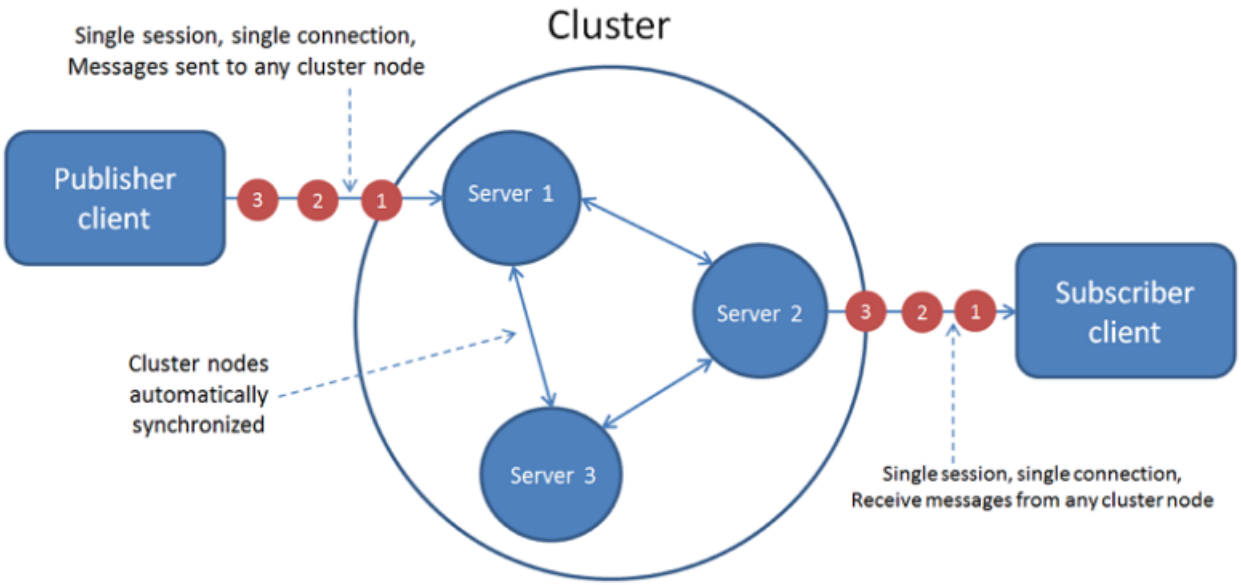
Horizontal scalability involves the use of several Universal Messaging servers that operate independently of each other. A publishing client using horizontal scalability can connect to multiple servers and publish messages to a channel or queue that is defined on each of these servers. The publishing client itself does not determine which server will receive which message; this is managed internally by Universal Messaging. A subscriber client that is using horizontal scalability, and that is subscribed to the channel or queue, will receive the messages from the servers. The servers are not synchronized, so the order in which the messages are received is not guaranteed.

### *Using Universal Messaging Horizontal Scalability*




This mode of processing is different from using a Universal Messaging cluster. In a cluster, all of the nodes are synchronized, so they all receive copies of the same messages that are sent by the publishing client. The subscribing client receives the messages from one of the cluster servers, in the order in which the messages were published.

### *Comparison with Using Universal Messaging Clustering*




HS is generally a client-side function, and a server that is used in a HS server set can also be used (at the same time) as a normal standalone realm or cluster. However, we don't recommend using the same channels/queues in both HS and non-HS modes.

## URL Syntax for Horizontal Scalability

In order to enable the HS feature, you provide a URL with HS-Syntax instead of the standard RNAME URL syntax (see the section "Client Connection" on page 176 for comparison).

The format of a horizontal scalability URL for both individual servers and clusters is bound by a set of parentheses - "(" and ")". The same syntax is used for both UM native sessions and JMS connections factories.

**Examples of Horizontal Scalability URL Syntax**

1. (UM1)(UM2)(UM3)(UM4) - Indicates 4 standalone realms, namely UM1, UM2, UM3 and UM4. In this case, 4 physical connections will be constructed and maintained underneath a single nSession and thus a JMS Connection.
2. (UM1,UM2)(UM3,UM4) - Indicates 2 clusters, one consisting of UM1 and UM2 and the other consisting of UM3 and UM4. In this case, 2 physical connections will be constructed and maintained.
3. (UM1)(UM2,UM3)(UM4) - Indicates one cluster consisting of UM2 and UM3, and two standalone realms, namely UM1 and UM4. In this case, 3 physical connections will be constructed and maintained.

For the URL syntax, the following rules apply:

1. Each set of parentheses must contain at least one valid connection URL.
2. There is no API imposed limit on the number of sets of parentheses in the URL. However there are resource limitations that must be considered.

3. Each set of parentheses indicates a unique physical connection, and the realm names within each set of parentheses will be supplied unchanged to the underlying implementation.

**Example for UM Native API**

```
// Connect to UM servers on hosts: host1, host2, host3 and host4.
// The server on host3 and host4 are configured in a cluster
nSessionAttributes sessionAttributes = new nSessionAttributes(
    "(nsp://host1:9000)(nsp://host2:9000)(nsp://host3:9000,nsp://host4:9000)");
nSession hsSession = nSessionFactory.create(sessionAttributes);
hsSession.init();
//Create a channel on all nodes in the HS set
nChannelAttributes nca = new nChannelAttributes("myChannel");
nca.setClusterWide(true);
nChannel channel = hsSession.createChannel(nca);
```

**Example for JMS API:**

```
// Create a connection factory which connects to the UM servers
// on hosts: host1, host2, host3 and host4.
// The server on host3 and host4 are configured in a cluster
//com.pcbsys.nirvana.nJMS.ConnectionFactoryImpl
ConnectionFactoryImpl conFac = new ConnectionFactoryImpl(
    "(nsp://host1:9000)(nsp://host2:9000)(nsp://host3:9000,nsp://host4:9000)");
Connection connection = conFac.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//Create a queue on all nodes in the HS set
session.createQueue("myQueue");
```

# Usage Notes for Horizontal Scalability

## JMS/JNDI Configuration

Although you would use an HS-specific RNAME when creating a JMS connection factory, you should use a regular RNAME URL when creating the JNDI initial context to bind and look up JNDI entries. For example: if the landscape consists of stand-alone Universal Messaging servers UM1 and UM2, and a two-node cluster with Universal Messaging server UM3 and UM4, you would have the following considerations:

- When creating the JNDI assets, the operations will need to be executed for each realm and/or cluster expected to be used by JMS in the HS environment.
- Whilst creating the JNDI assets, set the provider URL for the Initial Context per realm and/or cluster. For example, for the first initial context use a provider URL of "UM1", for the second initial context use a provider URL of "UM2", and for the cluster provide a URL of "UM3,UM4".
- When instantiating the JNDI context for use, set the provider URL as: "UM1, UM2, UM3, UM4". Thus if a realm is unavailable, you can move to the next available URL for the JNDI Context.
- When creating and binding JMS Connection Factories, instantiate them with an HS RNAME, e.g. "(UM1)(UM2)(UM3,UM4)". The JMS connections created from this connection factory will utilize the HS feature.

### Store configuration consistency

In order for the HS feature to function correctly, you should ensure the consistency of Universal Messaging channels and queues across the HS landscape. In other words, if you want to use a channel named "myChannel" in an HS manner, it should be present with the same channel setting across all Universal Messaging servers in the HS landscape. This can be achieved by either manually creating the channels/queues with identical configuration on all nodes, or by using an HS native nSession or JMS connection to create the channels/queues. **When creating stores using such a session/connection, channels/queues will not be created on nodes which are currently offline.**

### Unavailable servers

If a Universal Messaging Server in the HS configured nSession or JMS Connection is unavailable at the start of the session, or if a server becomes unavailable during the session for whatever reason, the session will automatically try to re-establish the physical connection to the server. If the unavailable server becomes available again, the HS session will try to resume publish/consume operations on that node as long as it contains the required stores - if the stores are not available, a log entry will be printed in the logs.

If the session is able to connect to at least one node in the HS Landscape, then the session will be in a connected and usable state.

If all the configured servers become unavailable, then the nSession or JMS Connection will be closed. To recover from this situation, you need to restart at least one of the servers, then restart the client sessions.

If a channel/queue creation or consumer operation is executed while any of the configured physical connections are not available, then this operation will not affect the offline nodes. Thus the channel/queue would need to be manually created and validated across the HS environment and the nSession or JMS Connection destroyed and restarted.

### Using Pause Publishing

When one of the servers in the HS session is configured for pause publishing, the clients that attempt transactional publishing to the server will receive `nPublishPausedException`. The rest of the servers will process the transactions and the events will be published successfully.

When clients use non-transactional publishing, events sent to the server with pause publishing enabled are lost. To ensure that the client is notified that the events are not published successfully, create an asynchronous exception listener for the HS session.

### Transactional Publishing When a Server is Unavailable

When a client is performing multi-threaded transactional publishing of messages using an HS session, and one of the Universal Messaging servers becomes unavailable during the transaction, some messages might remain unpublished. In addition, the client returns an `nSessionNotConnectedException`. The exception notifies you about the delivery state of the messages and suggests that you retry the transaction.

### Message ordering and duplicate detection

The HS feature does not provide any guarantees for the order of events being produced/published and consumed. The nSession or JMS Connection will pass incoming events to the consumers as soon as they are available and in a multi-server landscape this order is not guaranteed. The order of the events through a single server/cluster is still maintained.

The publishing is done in a round-robin fashion with each individual event being sent to the next available node. The same applies for transactions of events - all of the events in a single transaction will be sent to the same node, the events in the next transaction to the next node in the HS landscape and so on.

There is greater chance of duplicate messages if the client is not doing individual acknowledgement of messages within a HS environment. This is due to a side effect of transactional consuming, whereby an acknowledgement may return successful in the consumer API but a physical connection may be unavailable and the events that were outstanding will be redelivered.

HS has no ability to detect duplicate events, so if you were to produce the two streams of events with the same channel/queue name on two independent servers or clusters, then a client consuming using HS will receive duplicate events.

### Durable Subscribers

As stated above, message ordering is not guaranteed when producing messages due to the round-robin behavior of an HS producer. This holds especially true when both the producing and consuming of events is done through an HS enabled session. This means that while the individual durable instance for a specific server or cluster will continue to uphold any and all guarantees correctly, it is only at the point after it has been received into a server or cluster that it can maintain the order of events, and that is only for events it has received. HS consumer events will be interleaved with other realms and/or clusters events which will change the original HS producer's order of events. This is especially valid for the *Serial* durable subscribers, whereby an HS producer should not be used if the order of events is expected to be in the order of the producer/publisher submitting them. For example, in a simple two server HS environment UM1, UM2, we produce events 1, 2, 3, 4.The server UM1 receives events 1 and 3 and the server UM2 receives events 2 and 4. Upon consuming, we can expect to receive them in any of these combinations: 1-3-2-4, 1-2-3-4, 1-2-4-3, 2-4-1-3, 2-1-4-3, 2-1-3-4.

For related information on durable subscriptions, see the section "Durable Subscriptions" on page 153.

### Event Identifiers

In the native API, clients get event identifiers using the call `nConsumeEvent.getEventID()`. In the API for JMS they are hidden, though are used for event identification during processing. Within HS these identifiers are generated in the client API with the original event identifier tracked within the HS layer to facilitate event processing like acknowledgment and rollback. The important part to understand with generating the usable ID outside a server instance is that APIs that try to subscribe using event identifiers at a certain point within a channel/queue will be ignored and

will always start at the beginning. This may cause some duplication of events when not using durables or using the *exclusive durable* type in the native API.

### Server Ports/Interfaces

An HS session supports connecting to various connection interfaces, e.g. NHP/NHPS/NSP/NSPS, and you can use a combination of these in a single HS RNAME:

```
nSessionAttributes sessionAttributes = new nSessionAttributes(
"(nsps://host1:9000)(nsp://host2:9000)(nhp://host3:9000,nhp://host4:9000)");
nSession hsSession = nSessionFactory.create(sessionAttributes);
hsSession.init();
```

### Logging

HS related log entries will be produced in the client application's log with various log levels. The log entries for the HS layer are prefixed with the "HS>" string. Using the TRACE log level in an HS client will log each nConsumeEvent received from the servers and will substantially increase the amount of information logged.

## Horizontal Scalability Behavior in a Mixed Landscape

Because a higher-release Java client can connect to a lower-release server, in mixed server environments, the client might use a different release version for each server. Different versions can cause inconsistent behavior when operating against a horizontal scalability (HS) landscape. For example, some operations that are unsupported for some servers might not function properly.

In such cases, you must modify the HS URL of your application, restricting it to use the lowest server release version available within the HS landscape. To specify the lowest release version, you add the `hsReleaseVersion` parameter at the end of the HS URL.

The `hsReleaseVersion` parameter is an optional HS URL parameter that configures the Universal Messaging connection protocol release version to use when the client connects to an HS group of servers. The value of the parameter has a *`major.minor`* format, where the *`major`* and *`minor`* version components must be positive integers representing a valid Universal Messaging release version.

For example, you can have the following HS URL including `hsReleaseVersion`:

```
(nsp://localhost:9000)(nsp://localhost:9001)?hsReleaseVersion=10.5
```

In this example, the client will try to connect to both Universal Messaging servers requesting a connection protocol version of 10.5. If any of the servers does not support the configured version, connectivity to this server will not be established and it will not be available for user operations. However, while the horizontal scalability session is active, the session will continue to retry to connect to the target server in the background.

**Important:**
If you do not specify `hsReleaseVersion` in the HS URL, the session will validate that all servers have matching release versions (up to *`major`* and *`minor`* version components) and will fail to initialize if the versions of any of the servers are different. If some of the servers are not online during session initialization, but become available later after the session is initialized, and

negotiate a different Universal Messaging connection protocol version than the rest of the servers, the client will log an error message and close the session automatically.

For more information about bi-directional client and server compatibility, see "Bi-directional Client and Server Compatibility" on page 40.

## Restrictions

### nAdminAPI, Enterprise Manager and other tools built on top of the nAdminAPI

The administration API does not support HS URL syntax or the HS feature, so you must administer the servers individually. This adds some burden in creating resources.

### Datagroups

Datagroups are unsupported in an HS environment.

### JNDI Provider URL

Although the assets within JNDI support HS, the Initial Context does not.

### Language/API support

This feature supports only the Universal Messaging Client API for Java. The Universal Messaging Admin APIs for Java as well as all other language APIs (C++, .NET, JavaScript, etc.) are not supported.

# Event Fragmentation

Universal Messaging is capable of sending large messages. The maximum message size is given by the configuration parameter `MaxBufferSize`. For a description of this parameter, see the section Realm Configuration in the description of the Enterprise Manager in the *Universal Messaging Administration Guide*.

However, to get the best performance out of the system, it is important to consider how the data for such events is sent. In some cases, it might be better to compress or fragment the message rather than increasing `MaxBufferSize`.

If you want to send a large file, you could first compress the file before attaching it to an nConsumeEvent as a byte array. It takes time to compress data but as long as the data compresses well, you may find that the by reducing the network utilization, your system operates more efficiently.

Another option is to fragment the data. To fragment the data you need to convert to a byte array as before, but split the byte array and send multiple nConsumeEvent requests rather than one. By doing this, the events are handled completely separately by the server, so there are some things to consider; for example, this approach will not work if you are using a queue with multiple consumers.

# 6 Messaging Paradigms


- Overview ..... 146
- Publish / Subscribe (using Channels/Topics) ..... 146
- Publish / Subscribe (using Data Groups and Data Streams) ..... 146
- Message Queues ..... 150
- Details of Usage for Software Developers ..... 150

# Overview

Universal Messaging supports two broad messaging paradigms - Publish/Subscribe and Message Queues. 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 within 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 Data Groups and Data Streams)

## Overview of Data Groups and Data Streams

Universal Messaging *data groups* provide an alternative to channels (JMS topics) for the publish/subscribe messaging paradigm.

Universal Messaging data groups provide a very lightweight grouping structure that allows you to manage user subscriptions remotely and transparently. The process of managing subscribers can be carried out by publishers themselves or by some other process.

There are two resources that are used when interacting with data groups: *data streams* and *data groups*.

**Data Streams**

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

A data stream can be a member of one or more data groups.

A Universal Messaging client (typically, but not necessarily, a simple consumer) may initialize a data stream-enabled session, making it eligible for membership in one or more data groups. Such a client is considered a data stream.

**Data Groups**

A data group is essentially a group of consumers to which publishers can send events; more specifically, data group members are either individual consumers or other (nested) data groups. A consumer which is a member of a data group is known as a *data stream*.

Any event written to a data group will be propagated to all data streams that are members of that data group.

Data groups may also contain other data groups. Any event written to an upper level data group will be written to all contained data groups, and thus to all contained data streams.

Note that all data streams are automatically added to the realm server's Default data group. Writing an event to the Default data group, therefore, will ensure it is delivered to any client with a session configured to use a data stream.

Each data group is a resource that exists within the Universal Messaging realm server, or within a cluster of multiple realm servers. Creating a data group creates the physical object within the realm. Once created, references to the data group can be obtained using the Universal Messaging Client and Admin APIs. Data groups can also be monitored and managed using the Enterprise Manager.

Messages published to a data group will be sent to all members of the group. There can be multiple publishers associated with a single data group, and data group membership can be managed by any Universal Messaging client that has permissions to do so.

Data groups are designed to support large numbers of consumers whose subscriptions are typically fluid in nature. The addition or removal of consumers from data groups can be entirely transparent from the consumer perspective.

**Note:**
It is typical (though not necessary) for a process which publishes data to data groups to be independent of the process that manages data group structures and data group membership of end-user data streams. This allows the logic for these two very different responsibilities to be separated. Thus, data streams do not determine the data groups of which they are members; their membership is determined by clients with the appropriate permissions.

For information about setting up and using data groups and streams programmatically, see the Universal Messaging developer guide for your chosen programming language (*Developer Guide for Java*, etc.). Alternatively, refer to the *Enterprise Manager* for information on working with data groups and streams via a graphical user interface.

### Using in a Multicast environment

Data groups and data streams can be used in a *multicast* environment. Refer to the section "Multicast: An Overview" on page 126 for related information.

## Sample Structures for Data Groups and Data Streams

### Example 1: A sample data group structure

Imagine a Foreign Exchange organization that provides different tiers of currency price data to users on different desks at their various customers. Valued customer desks might be provided with gold level data, while less valued desks might receive silver or bronze tiered data:

- EURUSD_Gold
  - Customer_Desk_A
    - *User1*
    - *User2*

    - Customer_Desk_B
        - *User3*
        - *User4*
- EURUSD_Silver
    - Customer_Desk_C
        - *User5*
    - Customer_Desk_D
        - *User6*
- EURUSD_Bronze
- GBPUSD_Gold
    - Customer_Desk_A
        - *User1*
        - *User2*

A suitable data group structure to represent this arrangement might be to create eight data groups:

- EURUSD_Gold
- EURUSD_Silver
- EURUSD_Bronze
- GBPUSD_Gold
- Customer_Desk_A
- Customer_Desk_B
- Customer_Desk_C
- Customer_Desk_D

Then, structure these data groups as follows:

- EURUSD_Gold
    - Customer_Desk_A
    - Customer_Desk_B
- EURUSD_Silver
    - Customer_Desk_C
    - Customer_Desk_D

- EURUSD_Bronze
- GBPUSD_Gold
  - Customer_Desk_A

Price data would be published directly to the four data groups EURUSD_Gold, EURUSD_Silver, EURUSD_Bronze and GBPUSD_Gold (though, optimally, a publisher may well choose not to publish to the empty data group EURUSD_Bronze). By virtue of this structure, prices published to EURUSD_Gold, for example, will be delivered to both the data groups Customer_Desk_A and Customer_Desk_B.

End user clients with data stream-enabled sessions can be added to arbitrary data groups - and thus receive data from those data groups. So, for example, adding User1 and User2 as data stream members of the data group Customer_Desk_A, both users would, by virtue of the above structure, receive price data via their inherited membership of both EURUSD_Gold and GBPUSD_Gold.

### Example 2: Dynamic Changes in the sample data group structure

Let us now assume that our organization decides to upgrade the level of EURUSD price data sent to users on Customer_Desk_C from silver to gold.

Achieving this with data groups is extremely simple. All we need to do is remove the Customer_Desk_C data group from the EURUSD_Silver data group, and add it to the EURUSD_Gold data group instead, resulting in the following new structure:

- EURUSD_Gold
  - Customer_Desk_A
  - Customer_Desk_B
  - Customer_Desk_C
- EURUSD_Silver
  - Customer_Desk_D
- EURUSD_Bronze
- GBPUSD_Gold
  - Customer_Desk_A

### Example 3: Even More Dynamic Changes in the sample data group structure

Let's take our example a little further. Assume that the publisher of the gold EURUSD price data stops for some reason, and that the organization decides customers who were receiving this data should, for the time being, fall back to silver EURUSD level prices instead.

One way of doing this would be to move the three data groups Customer_Desk_A, Customer_Desk_B and Customer_Desk_C out of the EURUSD_Gold data group and into the EURUSD_Silver data group.

An even simpler way, however, would be to just move the EURUSD_Gold data group itself into the EURUSD_Silver data group, resulting in the following structure:

- EURUSD_Silver
  - EURUSD_Gold
    - Customer_Desk_A
    - Customer_Desk_B
    - Customer_Desk_C
  - Customer_Desk_D
- EURUSD_Bronze
- GBPUSD_Gold
  - Customer_Desk_A

This structure could be used while the gold level publisher of EURUSD prices remains inactive. Once the publisher of the gold level data restarts, the EURUSD_Gold data group can be removed from the EURUSD_Silver data group (since if it were not removed, the users expecting gold level EURUSD prices would now receive both gold and silver level prices).

## 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.

## 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.

### Comparison of Messaging Paradigms

| | Datagroups | Channels | Queues |
|---|---|---|---|
| Subscriptions Managed Remotely | ✓ | | |
| Nestable | ✓ | | |
| Clusterable | ✓ | ✓ | ✓ |
| Persistence (data) | | ✓ | ✓ |

| | Datagroups | Channels | Queues |
|---|---|---|---|
| Persistence (state) | ✓ | ✓ | ✓ |
| Message Replay | ✓ | ✓ | ✓ |
| Synchronous Consumer | | ✓ | ✓ |
| Asynchronous Consumer | ✓ | ✓ | ✓ |
| Synchronous Producer | ✓ | ✓ | ✓ |
| 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 | | ✓ | ✓ |
| User Access Controlled via ACLs | | ✓ | ✓ |
| Microsoft Reactive Extensions | ✓ | ✓ | ✓ |
| Accessible via JMS | | ✓ | ✓ |

# 7 Durable Subscriptions


- Overview of Durable Subscriptions ..... 154
- Types of Durable Subscription ..... 155
- Using Durable Subscriptions with Multiple Clients ..... 160
- Using the Prefetch API ..... 162
- Durable Event Browsing and Purging ..... 165

# Overview of Durable Subscriptions

Normally, events on a channel remain on the channel until they are automatically purged. Reasons for purging can be, for example, because the channel is getting full or because events have exceeded their preconfigured lifespan. In some cases, you may wish to retain events on the channel until they are explicitly acknowledged by a client application.

To cover this requirement, Universal Messaging offers a feature called *Durable Subscriptions*. A durable subscription provides the ability for the realm server to maintain state for events that are consumed by a specific client on a channel. Then, even if the client is stopped and restarted, the client will continue to consume available events from where it left off. Durable subscriptions can be used regardless of whether you have set up Universal Messaging to use the native fanout engine or the JMS-style fanout engine, and the behavior of the durable subscriptions is the same in both cases.

In this documentation, we also refer to durable subscriptions as simply *durables*. In previous product releases, we also referred to durable subscriptions as *named objects*.

There can be multiple durable subscriptions per channel, each identified by a unique name. They operate entirely independently from each other, and are unaware of each other's existence, although they access the same events on the channel.

A durable subscription continues to exist and receive events from the channel, even if there are no clients connected or subscribed. A durable subscription that contains no events also continues to exist. A durable subscription can only be deleted by explicit means, either by the client executing appropriate API code, or by an administrator using the appropriate feature in the Enterprise Manager, or Command Central, or on the command line.

*Each event remains on the channel until all durable subscriptions have consumed (and acknowledged) that event, and then the event is removed from the channel.*

One reason for using durable subscriptions is to facilitate recovery if a client is terminated. For example, if the consumer crashes or you turn off the computer and start up another one, the realm server will remember where that client finished.

Universal Messaging provides the ability to consume messages synchronously or asynchronously from durables.

There are different ways in which events consumed by clients can be acknowledged. By specifying 'auto acknowledge' when constructing either the synchronous or asynchronous client session, each event delivered to the client is automatically acknowledged as consumed. If 'auto acknowledge' is not set, then each event consumed has to be acknowledged explicitly by the client.

## Persistent durable subscriptions

Durable subscriptions are persistent. This means that if the Universal Messaging realm server is stopped and restarted, the processing of durable subscriptions by connected clients continues from where it stopped.

### Cluster-wide durable subscriptions

All durable subscriptions can be specified as "cluster wide". If the durable subscription is cluster wide and it exists on a cluster wide channel, the durable subscription will also be replicated across the cluster.

### Durable Subscriptions with multiple clients

There are several types of durable subscription, as described in the following section "Types of Durable Subscription" on page 155.

Durable subscriptions of type *Exclusive* allow only one active client. For such durable subscriptions, Universal Messaging maintains an internal record of events until the client has received the events. When the client acknowledges (i.e. informs the realm server) that it has received a certain event from the durable subscription, Universal Messaging deletes this event from the internal record and all other events in the internal record that have a lower event ID (i.e. were created at an earlier time).

There are some cases in which it may be desirable for two or more consumers to access a durable subscription at the same time. Universal Messaging offers such functionality with the durable types *Shared* and *Serial*. These durable subscriptions allow multiple consumers to access the same durable subscription. Universal Messaging delivers events to the consumers in round-robin fashion. So, for example, if there are 10 events in the durable subscription and two consumers who share the durable subscription, then Universal Messaging will deliver events 1, 3, 5, 7, 9 to the first consumer and events 2, 4, 6, 8, 10 to the second consumer.

A typical use case is load balancing, where multiple servers are available to process a stream of events.

### Rollback

If an event for an asynchronous durable subscription of type *Shared* or *Serial* has been rolled back, the event is immediately re-delivered. In addition, the event's redelivery count is incremented and the event's redelivery flag is set.

If an event for an asynchronous durable subscription of type *Exclusive* has been rolled back, the event is re-delivered after a re-subscription; this prevents duplicate events from being received. Note that in this case, the event's redelivery count and redelivery flag are left unchanged.

## Types of Durable Subscription

Universal Messaging offers the following types of durable subscription:

The available types are:

- Exclusive (also called *Named*)
- Shared
- Serial

Based on the type, the durable subscriptions are managed in a different way.

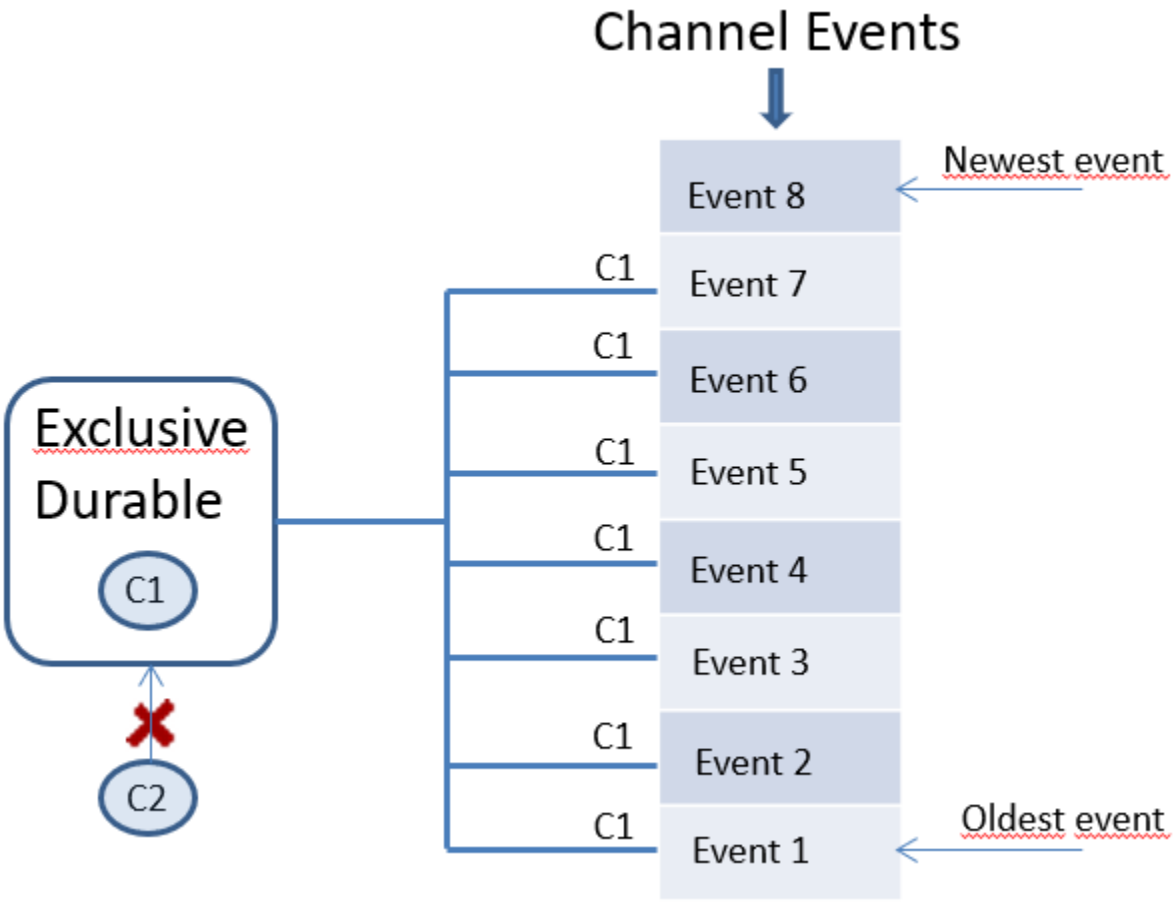
### Exclusive Durable Type

With the Exclusive durable type, only one client session is subscribed and connected to the durable subscription at a time. Other clients cannot subscribe while the current client session is subscribed.

Each read operation issued by a client returns a single event from the durable to the client. The client processes the event according to the client's application logic, then returns an acknowledgement to the durable that the event has been processed. Each subsequent read operation by the client returns the next event from the durable in chronological order.

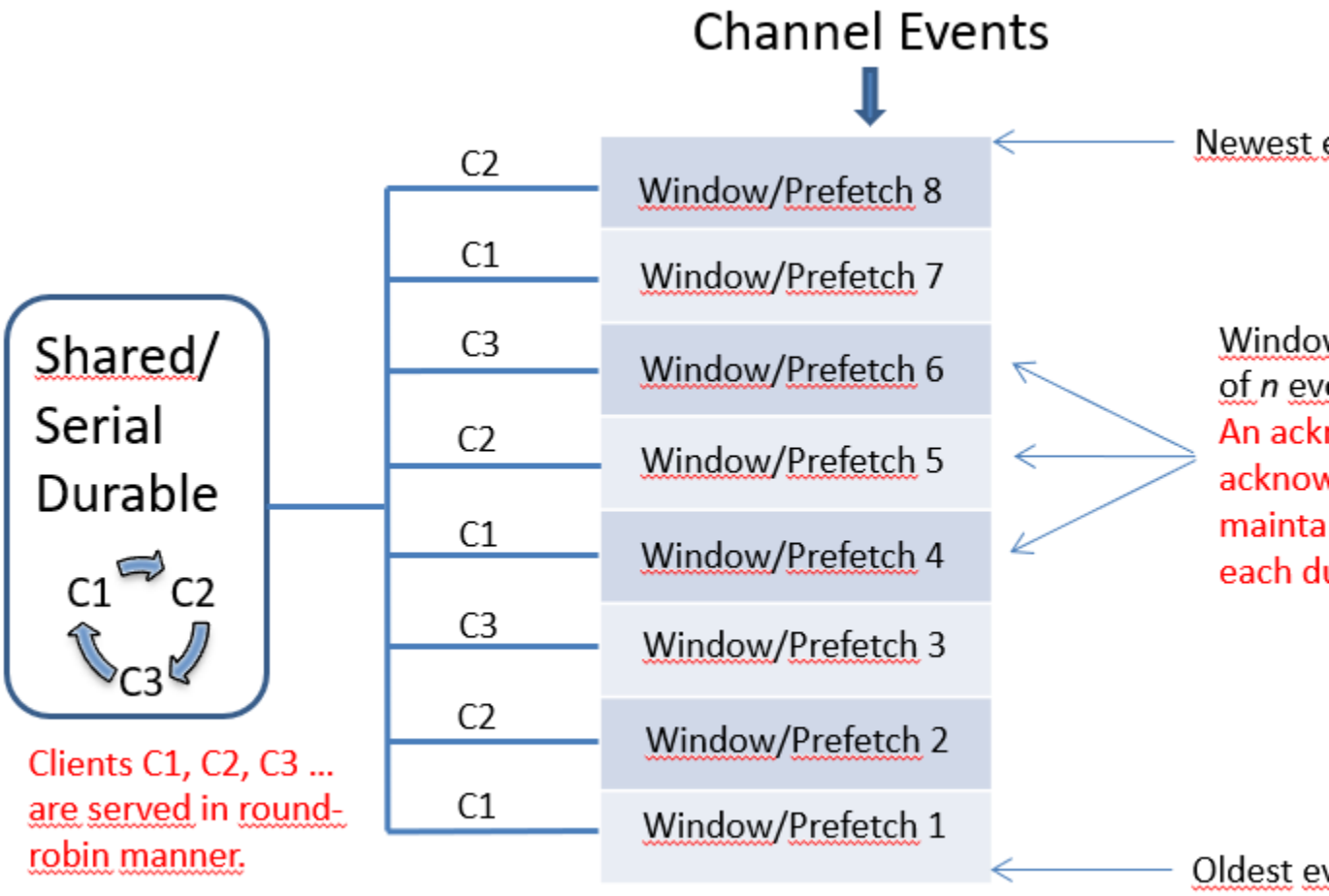
If the client connection is terminated, a subsequent client connection on the durable will restart at the next unacknowledged event on the durable.

The following diagram shows the behavior of the Exclusive durable. Client session C1 is subscribed and connected to the durable. No other client sessions can subscribe to the same durable while the current client session is active and subscribed.



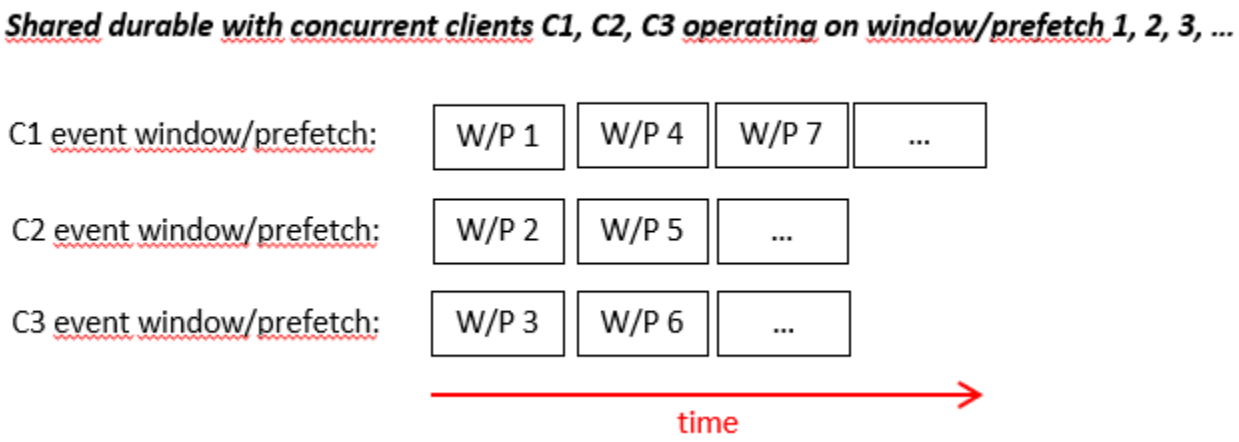
### Shared and Serial Durable Type

The following diagram illustrates the bahavior of the Shared and Serial durable types, which can have multiple active client sessions. The client sessions are served in a round-robin manner. Each client receives a window of events. The Shared type differs from the Serial type in how the clients process their assigned event windows.




### Shared Durable Type

With the Shared durable type, multiple clients can subscribe and receive events from the durable in round-robin fashion. You can configure the durable so that each client receives a window that consists of multiple events instead of a single event. As soon as one client has received its allotted event or window of events, the next client receives its events, and so on. All subscribed clients on the durable can process the events concurrently.

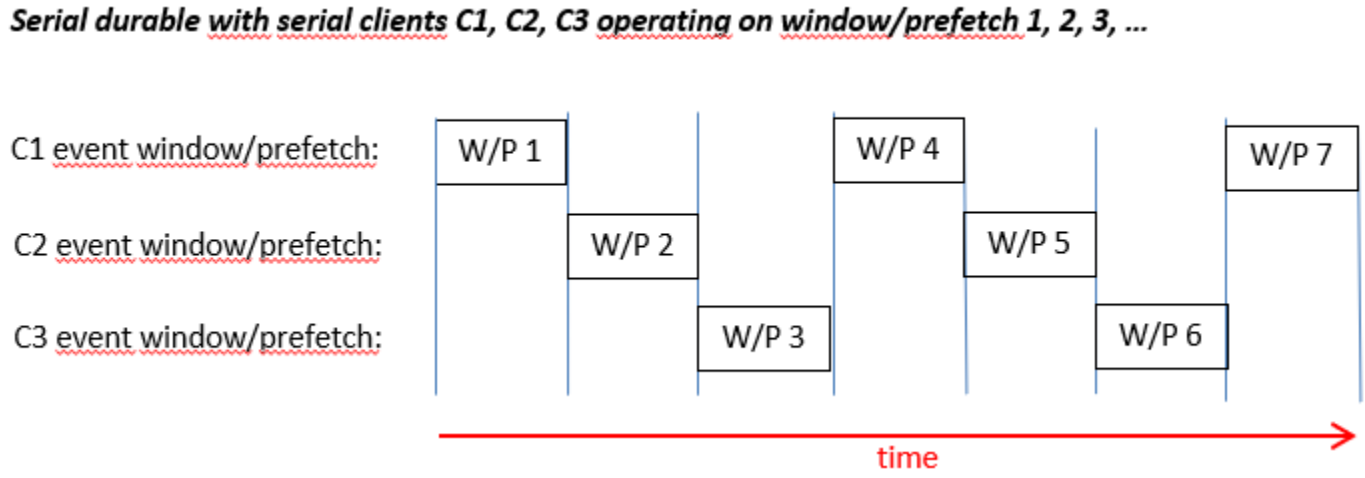



The chronological order in which events are processed is not guaranteed. For example: the C1 client receives a window of events, then the C2 client receives a window of events, but C2 completes its processing and acknowledges its events before C1 completes its processing.

### Serial Durable Type

With the Serial durable type, the events from the event windows are delivered to consumers in chronological order.

Multiple client sessions can subscribe and receive events from the durable, but the delivery method ensured that only one client will receive events at a time. The Serial type differs from the Shared type in that a client will only receive an event or a window of events when the previous client has acknowledged or rolled back its events. While any of the subscribed clients is processing its events, the other clients are inactive.




The window size of the Event delivery type and the number of client sessions will determine how Universal Messaging deals with the serial delivery of events.

- Window Size=1 with one client session

With a single client, Software AG recommends setting the window size to one to guarantee the serial delivery and processing of events. The server will not send more events to the client until the event that is already sent has been acknowledged or rolled back. Note that when "Window Size=1", the event delivery and processing is not done in parallel and the delivery throughput is lower.

Example:

The window size is set to 1. The client acknowledges events 1 to 4, but rolls back event 5. The server will wait for each event (1,2,3, and 4) to be processed before the server re-sends event 5. This ensures that events 1 to 5 are processed in the exact order, in which they were sent and received.

- Window Size > 1 with one client session

  When the size of the event window is larger than 1, the server performance improves, because the delivery and processing of the batches of events is done in parallel. However, the serial delivery of events is not guaranteed in the following cases:

  - When an event is rolled back, the client receives the next event from the window (which was sent during the rollback), instead of the event that was rolled back. In this case the order in which the events get delivered to the client does not match the sending order.

    Example:

    The window size is set to 6. The client acknowledges events 1 to 4, but rolls back event 5. If the server has already sent event 6, that event will be delivered next instead of the rolled-back event. Event 5 will get re-delivered after event 6.

  - The client receives a batch of events, but does not process the events in the order, in which they were received.

    Example:

    The window size is set to 6. The client receives events 1, 2, 3, and 4, but acknowledges the events in a different order, for example 4, 1, 3, 2.

- Window Size=1 with multiple client sessions

  The events are delivered to a client after they have been processed by the previous client. Only one client can have unprocessed events at a time. If none of the clients have outstanding events, the delivery can be scheduled for any of the available clients. The events are delivered among clients in a round-robin fashion.

- Window Size > 1 with multiple client sessions

  When the server attempts a round-robin delivery with an event window larger than 1, the round-robin delivery is not guaranteed. For example, if a client processes the events in parallel with the event delivery from the server, the client might receive more than one consequent batches of events.

  Example:

We have two clients, C1 and C2, and the window size is set to 6. The durable subscription has 6 events.

C1 receives all six events and acknowledges event 1. When a new event is published to the server, the server delivers the event to C1, because C1 has an empty seat for 1 event in its window and 5 unprocessed events. To ensure the serial processing of the events, the server will send events to C2 only after C1 processes all of its in-flight events.

# Using Durable Subscriptions with Multiple Clients

## Window/Prefetch Size of Shared and Serial Durable Subscriptions

The Universal Messaging client API offers the option of specifying a window size (for asynchronous clients) or prefetch size (for synchronous clients) for a shared durable subscription. The window/prefetch size can be set independently for both synchronous and asynchronous subscribers which use shared durable subscriptions. The client API contains two methods that accept the window/prefetch size as a parameter: one for creating an iterator for a channel and one for adding a new subscriber with nEventListener.

When specifying the window/prefetch size, the session is by default not auto-acknowledge. This means that when the window size is reached for an asynchronous client, the client stops receiving events. When an iterator is used with a defined prefetch size for a synchronous client, the server returns the requested number of events, and it is up to the synchronous client whether to commit or roll back or prefetch the next set of events.

If the API parameter for window size is not specified for an asynchronous subscriber, `nConstants.setMaxUnackedEvents(x)` will be used as the default client window size, if this value has been set before subscribing with listeners. If the durable subscription is not a shared durable subscription, an `nIllegalArgumentException` is thrown. If you specify a negative window size for this API parameter, the default value stored in the server configurations will be used instead. The default value is "ClientQueueWindow", which is available in the "Cluster Config" group.

**Example 1**

You can set the size of an asynchronous client window using a method from the public nChannel API: `channel.addSubscriber(nEventListener nel, nDurable name, String selector, int windowSize)`. A different window size can be configured for every asynchronous subscriber. Here is a simple code snippet for this:

```
nChannelAttributes channelAttr = new nChannelAttributes("channel");
nChannel channel = session.createChannel(channelAttr);
boolean isPersistent = true;
boolean isClusterWide = false;
int startEventId = 0;
nDurableAttributes attr =
    nDurableAttributes.create(nDurableAttributes.nDurableType.Shared,
    "shared-durable");
attr.setPersistent(isPersistent);
attr.setClustered(isClusterWide);
attr.setStartEID(startEventId);
nDurable sharedDurable = null;
// Need to check first in case shared durable already exists!
```

```
try {
     sharedDurable = channels.getDurableManager().get(attr.getName());
   } catch (nNameDoesNotExistException ex) {
     sharedDurable = channels.getDurableManager().add(attr);
}
nEventListener listener = new nEventListener() {
@Override
public void go(nConsumeEvent evt) {
System.out.println("Consumed event " + evt.getEventID());
}
};
int windowSize = 4;
channel.addSubscriber(listener, sharedDurable, null, windowSize);
```

**Example 2**

You can also set the prefetch size for a synchronous client when using a channel iterator. Here is an example:

```
nSession session = nSessionFactory.create(sessionAttr);
session.init();
nChannelAttributes channelAttr = new nChannelAttributes("channel");
nChannel channel = session.createChannel(channelAttr);
boolean isPersistent = true;
boolean isClusterWide = false;
int startEventId = 0;
nDurable sharedDurable = null;
// create the shared durable
nDurableAttributes attr =
    nDurableAttributes.create(nDurableAttributes.nDurableType.Shared,
    "shared-durable");
attr.setPersistent(isPersistent);
attr.setClustered(isClusterWide);
attr.setStartEID(startEventId);
// Need to check first in case shared durable is already exist!
try {
    sharedDurable = channels.getDurableManager().get(attr.getName());
    } catch (nNameDoesNotExistException ex) {
    sharedDurable = channels.getDurableManager().add(attr);
}
int prefetchSize = 4;
nChannelIterator iterator = channel.createIterator(sharedDurable, null);
List<nConsumeEvent> evts = iterator.getNext(prefetchSize, timeout)
```

## Checking the number of unprocessed events on a durable

You can use the API method `getDepth()` on `nDurableNode` to get the number of events outstanding for a durable.

For non-shared durables this method returns an *estimation* of the number of outstanding events. This estimation does not include unacknowledged, purged or filtered events.

**Example**

```
nRealmNode realmNode =
    new nRealmNode(new nSessionAttributes("nsp://localhost:9000"));
nTopicNode topicNode = (nTopicNode) realmNode.findNode(CHANNEL_NAME);
nDurableNode durableNode = topicNode.getDurable(DURABLE_NAME);
```

```
long eventCount = durableNode.getDepth();
```

The number of outstanding events waiting for a commit or a rollback can be found out by:

```
long unackedCount = durableNode.getTransactionDepth();
```

### Storage Considerations

There are some storage considerations to be aware of when using shared durable subscriptions as opposed to normal (i.e. non-shared) durable subscriptions.

When you define a shared durable subscription, Universal Messaging maintains a copy of all of the events of the durable subscription in an internal store. The copies remain in the internal store until the original events have been accessed by the consumer. The benefit for the consumer is that events are held until the consumer is ready to receive them.

If the consumer accesses the events using event filtering, some events are never selected, and therefore the copies of these events remain in the internal store. As mentioned earlier, when an event is consumed for a normal durable subscription, all earlier events are also removed; however, for a shared durable subscription, only the consumed event is removed from the store. This is by design, and can, in time, lead to Universal Messaging retaining large numbers of events that are never used. If you have made the shared durable subscription persistent, a copy of the internal store will be maintained on disk, and this disk copy will also contain copies of the events that are not used.

In order to delete messages from the internal store you can purge events from the channel that the durable subscription is created on. That purge will also flow through to the internal store.

The set of events that are copied to the internal store for a given shared durable subscription is limited according to the following rules:

- When the first consumer connects to the shared durable subscription and specifies an event filter, then only events that match the filter will be copied to the internal store.
- If a second consumer connects to the same shared durable subscription and also specifies an event filter, then from that time onwards, only events matching the second filter will be copied to the internal store. The first consumer receives an asynchronous exception due to the changed filter, but still uses its original filter to take events off the internal store.
- If the second consumer specifies no event filter, the filter of the first consumer remains in effect for copying events to the internal store.

## Using the Prefetch API

### Overview

For better user experience with Universal Messaging's synchronous messaging APIs, a new set of synchronous consumer prefetch APIs were added in v10.7. The old "non-prefetch" APIs had the following disadvantages:

- The maximum number of events which can be read from the server was limited by the Window Size setting. The window size defines the maximum number of events that have been sent to consumer but have not yet been acknowledged by the consumer. The value of the window size was not transparent for the consumer and it could not be changed dynamically. This meant that if event acknowledgment was slower than synchronous consumption, synchronous consumers would throw an exception "Need to Commit or rollback" when the window size was reached.
- Events were cached-client side. Often, events were delivered to consumers in batches, but only one event was returned by the synchronous APIs. This left some pending cached events client-side, and this situation was not transparent for the user.

The prefetch APIs were created to address the above issues:

- The synchronous APIs can now receive prefetch events which they requested and will not be limited by a window size. The client application is now in control of how many pending/unacknowledged events it can hold.
- The APIs now return a list of events which eliminates the caching client-side and makes event delivery transparent for the consumer.

The old deprecated APIs that did not use prefetch are limited in v10.7 to have a prefetch of '1'. This means they cannot receive events in batches as they did before.

Window size is still in use for asynchronous consumers where the consumer is not in control of how many events will be received, thus here the window size makes sense to throttle the number of events sent.

## APIs added

The list of prefetch APIs added in v10.7 is as follows:

**Java**

- **nChannelIterator:**

```
public List<nConsumeEvent> getNextEvents (int prefetchSize)
public List<nConsumeEvent> getNextEvents (int prefetchSize, long timeout)
```

- **nQueueSyncReader:**

```
public List<nConsumeEvent> popEvents (int prefetchSize)
public List<nConsumeEvent> popEvents (int prefetchSize, long timeout)
public List<nConsumeEvent> popEvents (int prefetchSize, long timeout, String
selector)
```

- **MessageConsumerImpl:**

```
public List<javax.jms.Message> receiveMessages (int prefetchSize)
public List<javax.jms.Message> receiveMessages (long timeOut, int prefetchSize)
```

**C#**

- **nChannelIterator:**

```
public List<nConsumeEvent> getNextEvents(int prefetchSize)
public List<nConsumeEvent> getNextEvents(int prefetchSize, long timeout)
```

- **nQueueSyncReader:**

```
public List<nConsumeEvent> popEvents(int prefetchSize)
public List<nConsumeEvent> popEvents(int prefetchSize, long timeout)
public List<nConsumeEvent> popEvents(int prefetchSize, long timeout, String selector)
```

**C++**

- **nChannelIterator:**

```
std::list<nConsumeEvent*>* getNextEvents(int prefetchSize)
std::list<nConsumeEvent*>* getNextEvents(int prefetchSize, long timeout)
```

- **nQueueSyncReader:**

```
std::list<nConsumeEvent*>* popEvents(int prefetchSize)
std::list<nConsumeEvent*>* popEvents(int prefetchSize, longlong timeout)
std::list<nConsumeEvent*>* popEvents(int prefetchSize, longlong timeout, std::string
 selector)
```

## Deprecated APIs

The APIs deprecated in v10.7 are:

- **nChannel:**

```
public nChannelIterator createIterator(nDurable name, String selector, int
windowSize)
public nChannelIterator createIterator(nDurable name, String selector, int
windowSize, boolean autoAck)
```

- **nChannelIterator:**

```
public nConsumeEvent getNext()
public nConsumeEvent getNext(long timeout)
```

- **nQueueSyncReader:**

```
public final nConsumeEvent pop()
public final nConsumeEvent pop(final long timeout)
public nConsumeEvent pop(final long timeout, final String selector)
```

## Horizontal Scalability (HS) notes

Within the HS use case it is possible that the returned set of events is bigger than the requested prefetch size. The size can be the size of a previous prefetch value.

The HS mechanism sends the requests to every server in the HS landscape, and delivers the fastest response to the client. Thus if a previous request was done with a bigger prefetch value, a subsequent receive/pop call can actually receive a set of events which was requested with the bigger prefetch value and thus will be returned at once to avoid caching client-side.

A prefetch call in HS can return only events from one server. Currently the returned event set will not mix events from different servers.

For example, if we have a 3-server HS landscape and an HS channel with 100 events on each server. An iterator calls `getNext` with a prefetch size of 5. We receive 3 responses with 5 events in each, and we deliver the first response to the client and the other 2 responses are kept waiting in the HS layer. If now a subsequent `getNext` is called with prefetch size of 2, we deliver the cached response which has 5 events.

### Performance Notes

- Using a bigger prefetch size can improve performance

  Using a bigger prefetch size can improve performance as events will be delivered in batches (if there are events piled up on the server), especially for small events. There is no upper limit or the prefetch size, but there is a limit of 1MB for each batch. Performance-wise there is no benefit of using batches bigger than 1MB.

- JMS API

  The prefetch functionality is not part of the JMS spec, and therefore JMS clients cannot take advantage of the prefetch functionality right away. By default, the JMS API uses a prefetch size of 1 to avoid caching client-side. Nevertheless, to allow users to consume events in batches to improve performance, the system configuration property `nirvana.syncPrefetchSize` can be set client-side. This integer value property defines how many events the JMS synchronous consumer should request from the server (prefetch logic is the same). While the consumer will still get a single event from the receive call, the rest of the prefetch will be cached, so the client has the control over the cache, and also the same (or better) performance.

  After 10.7 Fix 1 this prefetch size for JMS can be set per Connection Factory using the connection factory property `JMS_my-channels_SyncPrefetchSize`.

- Using the prefetch APIs moves the responsibility for consumer event throttling to the client application; the server will no longer honor any window size. This needs to be taken into an account when upgrading synchronous consumer applications to 10.7+ .

### Older clients, working with the 10.7+ server

While the new synchronous clients will not be using a window size, clients from before the introduction of the prefetch feature will still be doing so. The new server will honor the window size (and the 10.5 queue batching) for older clients. This was done to preserve the behavior of old client applications until they can be migrated.

## Durable Event Browsing and Purging

### Durable Event Browsing

You can browse the un-acknowledged events on all durable types.

To do this, you need the durable object associated with the channel you want to browse, which you can get as follows:

```
nDurable durable = myChannel.getDurableManager().get("your durable name");
nDurableViewer durableViewer = durable.createViewer();
```

The nDurableViewer object has two methods:

- next() - Returns the next un-acknowledged nConsumeEvent. If there are no further events to be consumed at this time, the method returns null.
- close() - Closes the created nDurableViewer.

### Durable Event Purging

You can purge the un-acknowledged events on a durable.

To purge events, you need the durable object again:

```
nDurable durable = myChannel.getDurableManager().get("your durable name");
```

The durable object offers the following methods:

- removeAll - Removes all events on the durable.
- remove(long start , long end) - Removes an event range on a durable.
- remove(String filter) - Removes events matching a filter on a durable.
- remove(long eid) - Removes a single event with corresponding Event ID from the durable.

Currently, the purge capability is supported for serial and Shared Durables (i.e. on the nSharedDurable implementation of the nDurable object). If any of the above listed remove methods is called on a different durable type, it will throw an nIllegalStateException.

# 8 Clustered Server Concepts


- Clusters: An Overview ..... 168
- Active/Active Clustering ..... 172
- Active/Passive Clustering ..... 192
- Active/Active Clustering with Sites ..... 195

## Clusters: An Overview

Universal Messaging provides guaranteed message delivery across public, private, local and wide-area infrastructures.

A Universal Messaging cluster consists of Universal Messaging realm servers working together to provide high availability and reliability. An individual Universal Messaging realm server in the cluster is also referred to as a cluster node.

The realm servers 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 on one realm, the state change is updated on all realms in the cluster. For example, if an event is popped from a clustered queue on one realm, it is popped from all realms within the cluster.

Creating a cluster of Universal Messaging realms ensures that applications either publishing / subscribing to channels, or pushing / popping events from queues, can connect to any of the realms and view the same state. If one of the realms in the cluster is unavailable, client applications can automatically reconnect to any of the other cluster realms and carry on from where they were. For the clients, a cluster appears to be a single Universal Messaging server.

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.

A Universal Messaging cluster protects your messaging system from the following failures, and provides support for business contingency and disaster recovery:

- Application and service failures
- System and hardware failures
- Site failures

Universal Messaging clusters can be created, configured and administered using either Universal Messaging Enterprise Manager, the Universal Messaging administration APIs, or Command Central.

**Note:**
Some of the industry-standard protocols that are supported by Universal Messaging do not include full support for clustered operation, e.g. clients may not automatically fail over to another server if the server they are connected to fails. For specific protocols please consult their documentation to understand their capabilities and limitations. Universal Messaging's native APIs and JMS provider implementation have full support for clustering.

Universal Messaging provides built-in support for *active/active* 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 Universal Messaging realms in *active/passive* server configurations that use Shared Storage (see section "Shared Storage Configurations" in “About Active/Passive Clustering” on page 192).

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.

**Tip:**
Since the underlying purpose of a cluster is to provide resilience and high availability, we advise against running all the servers in a cluster on a single physical or virtual machine in a production environment.

## 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
- Data Groups
- Access Control Lists
- Resource Attributes 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.

## Approaches for Clustering

Universal Messaging supports these clustering approaches:

| | |
|---|---|
| Active/Active cluster with three or more servers | This approach offers the following features:<br>■ Active/Active<br>■ Transparent Client Failover<br>■ Transparent Realm Failover<br>Universal Messaging clusters are our recommended solution for high availability and redundancy. State is replicated across all active realms.<br>With 51% of realms required to form a functioning cluster (see the section "Quorum" on page 180 for an explanation of this percentage figure), this |

| | |
|---|---|
| | is an ideal configuration for fully automatic failover across a minimum of three realms.<br><br>As a general rule of thumb, an active/active cluster can be considered in the following circumstances:<br><br>■ You have a requirement for availability that cannot be satisfied otherwise.<br><br>■ You understand and can implement the deployment recommendations for active/active.<br><br>For more information on active/active clustering, refer to the section "Active/Active Clustering" on page 172. |
| Active/Active cluster with sites | This approach offers the following features:<br><br>■ Active/Active<br><br>■ Transparent Client Failover<br><br>■ Semi-Transparent Realm Failover<br><br>Universal Messaging Clusters with Sites provide most of the benefits of Universal Messaging Clusters but with less hardware and occasional manual intervention.<br><br>This configuration is designed for two sites, such as Production and Disaster Recovery sites, containing an equal number of realms (for example, one realm on each site or two realms on each site). In such a configuration, if the communication between the sites is lost, neither site can achieve the quorum of 51% of reachable realms required for a functioning cluster. This situation can be resolved by defining one of the sites to be the so-called *prime site*. If the prime site contains exactly 50% of reachable realms in the cluster, the prime site is allowed to form a functioning cluster. Failover is automatic should the "non-prime" site fail, and requires manual intervention only if the prime site fails.<br><br>**Note:**<br>Switching the prime site MUST be a manual operation by an administrator who can confirm that the previous prime site is indeed down and not merely disconnected from the other sites. Attempts to automate this process raises the risk of "split brain" situations, in which loss of data is very likely.<br><br>For more information on active/active clustering with sites, refer to the section "Active/Active Clustering with Sites" on page 195. |
| Active/Passive cluster with shared storage | This approach offers the following features:<br><br>■ Active/Passive |

| | |
|---|---|
| | ■ Transparent Client Failover<br>■ Manual Realm Failover<br>As an alternative to native Universal Messaging Clusters, Shared Storage configurations (see section "Shared Storage Configurations" in "About Active/Passive Clustering" on page 192) can be deployed to provide disaster recovery options.<br>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.<br>In general, we recommend the use of Universal Messaging Clusters or Universal Messaging Clusters with Sites in preference to shared storage configurations.<br>An active/passive cluster should be considered in the following circumstances:<br>■ You have an availability requirement that is already met by an active/passive Broker setup.<br>■ You need higher performance than is possible with an active/active cluster.<br>For more information on active/passive clustering, refer to the section "Active/Passive Clustering" on page 192 . |

## Planning for Cluster Implementation and Deployment

A Universal Messaging cluster with three or more servers (active/active cluster) is the recommended approach for clustering. This approach supports high availability and resilience, reduces outage during failover, and uses standard (local) disks.

The table below will help you decide on the clustering approach for your Universal Messaging clustering solution.

| Clustering Approach | Automatic client and server failover? | Vendor-specific cluster software required? | Shared storage required? | Minimum number of servers required? |
|---|---|---|---|---|
| Active/Active cluster with three or more servers | Yes | No | No<br>Uses standard (local) disk | 3 |
| Active/Active cluster with sites | Automatic client failover. | No | No | 2 |

| Clustering Approach | Automatic client and server failover? | Vendor-specific cluster software required? | Shared storage required? | Minimum number of servers required? |
|---|---|---|---|---|
| **Note:** Administrator must manually set the "IsPrime" flag to the other site if the site with the "IsPrime" flag fails. | Semi-automatic server failover. | | Uses standard (local) disk | |
| Active/Passive cluster with shared storage | Yes | Yes | Yes | 2 |

The most significant advantages and disadvantages of the active/active and active/passive clustering approaches are summarized in the table below:

| | Active/active | Active/passive |
|---|---|---|
| Advantages | Highest level of availability.<br>Runs on commodity hardware with simple storage and no additional software. | Higher performance than A/A.<br>Can re-use existing Broker A/P clustering infrastructure. |
| Disadvantages | Cluster state replication will impact performance.<br>Requires different hardware than Broker clusters. | Lower availability than A/A.<br>Requires shared storage and potentially additional management software. |

# Active/Active Clustering

## About Active/Active Clustering

### Introduction

In an active/active cluster, multiple servers are active and working together to publish and subscribe messages. Universal Messaging clients automatically move from one server to another server in a cluster as required or when specific servers within the cluster become unavailable to the client for any reason. The state of all the client operations is maintained in the cluster to enable automatic failover.

To form an active/active cluster, more than 50% of the servers (a quorum) in the cluster must be active and intercommunicating. Quorum is the term used to describe the state of a fully formed cluster with an elected master.

Applications connected to a Universal Messaging cluster can:

- Publish and subscribe to channels
- Push and pop events from queues
- Connect to any Universal Messaging server instance and view the server state

If a cluster node is unavailable, client applications automatically reconnect to any of the other cluster nodes and continue to operate.

## What is active/active clustering?

A Universal Messaging active/active cluster has multiple Universal Messaging server nodes (realm servers) running simultaneously. All nodes in the cluster are able to accept connections from publishing and subscribing clients.

Every node in the cluster maintains a separate but identical copy of the entire cluster state, i.e. channels, queues, durable subscribers, in-flight messages, etc. The cluster nodes co-operate continuously to ensure that the replicated state is maintained correctly across the whole cluster. The cluster elects a single node to act as the "master" and co-ordinate the state replication process.

If a cluster node becomes unavailable, e.g. due to a software, hardware or network failure, all clients connected to that node will quickly fail over to another node with minimal interruption to messaging traffic. This typically takes no more than a few seconds and, provided the clients are correctly configured, this process is transparent to the applications and no messages will be lost. If the cluster master node becomes unavailable, the remaining nodes will elect a new master between them. When a failed or disconnected node is able to reconnect to the cluster, it will automatically resynchronize its state and continue operating.

An active/active cluster will continue to process messages as long as more than 50% of the nodes in the cluster are operating and able to communicate with one another. For example, this means that a three-node cluster will continue operating as long as at least two of its nodes are running and able to communicate with each other.

Active/active clustering is an advanced capability offered by Universal Messaging that is not available with the webMethods Broker.

**What are the benefits of using active/active clustering?**

Active/active clustering provides a high availability resiliency within the messaging software layer as part of a high availability strategy. This is achieved using cheap local disks attached to each node with the messages and events synchronized between the nodes by Universal Messaging. This approach removes the need for expensive network disks (NAS or SAN) as well as the requirement for additional software to manage any failover.

**What should you know before using active/active clustering?**

The following table provides an initial set of questions you should think about when deciding on whether active/active is suitable as part of your high availability strategy.

| | |
|---|---|
| What runtime infrastructure will be used - virtual machine or physical? | Ideally nodes should be on physical machines.<br><br>Virtual machines should have pinned resources to ensure appropriate infrastructure for the cluster to run. Each node virtual machine should be on a separate physical machine. |
| What storage will be used – local or SAN? | Local disk should be used. SAN can cause contention when used below a cluster as all realms are saving to the same disk. |
| How many nodes in the cluster? | Best practice is 3 nodes, this allows the cluster to keep working if a node drops. |
| Are the nodes in the same data center or are they on multiple sites? | If you have a deployment across multiple data centers, we suggest you consider using active/active with sites, but be aware of the restrictions and operational considerations with such a configuration. See the section "Active/Active Clustering with Sites" on page 195 for related information. |
| Is this a new deployment or a Broker migration? | While Universal Messaging generally performs better than Broker, a cluster reduces performance of Universal Messaging by approximately 40% . |
| What are the availability requirements? | Clusters cannot be updated on a rolling basis. There will be a few seconds if the master node drops for a re-election to happen. |

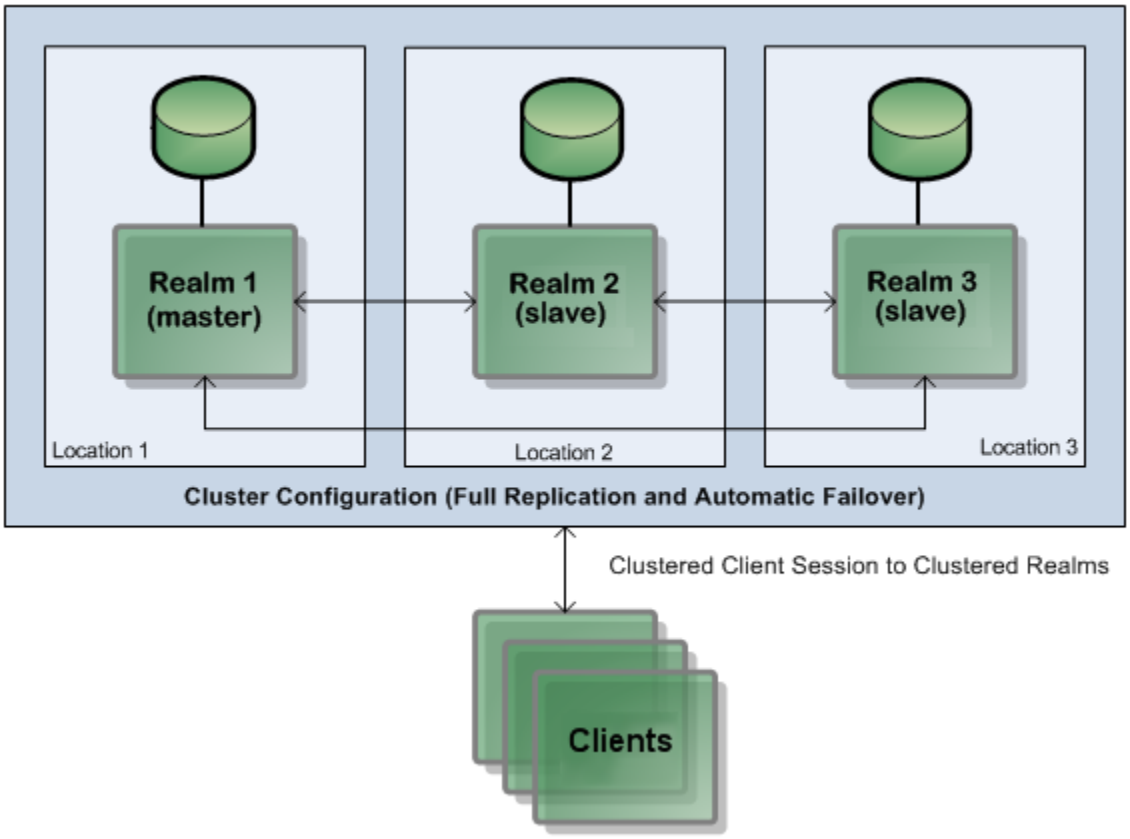
## How does the active/active cluster work?

In an active/active cluster, one of the cluster nodes must be designated as the master node. The master node is selected by the cluster nodes. Each cluster node submits a vote to choose the master node. If the master node exits or goes offline due to power or network failure, the remaining active cluster nodes elect a new master, provided more than 50% of the cluster nodes are available to form the cluster.

Cluster nodes replicate resources amongst themselves, and maintain the state of the resources across all cluster nodes. Operations such as configuration changes, transactions, and client connections go through the master node. The master node broadcasts the requests to the other cluster nodes to ensure that all the servers are in sync. If a cluster node disconnects and reconnects, all the states and data are recovered from the master node.

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

### *Typical three-realm active/active cluster distributed across three physical locations*




You can connect one cluster to another cluster through remote cluster connections. Remote cluster connections enable bi-directional joins between clusters, therefore joining the resources of both the clusters for publish and subscribe.

## How should an active/active cluster be deployed?

It is essential that each node in an active/active cluster is deployed on physically separate hardware, to reduce the risk of a single hardware failure affecting multiple cluster nodes. When virtualization is being used, this means that each node must run on a virtual machine that is pinned to a different physical host. In the case of blade servers, it is recommended that each cluster node runs on a blade in a different chassis.

Ideally, the storage used by each cluster node will be local to the physical hardware that the node is running on, even when the machine and storage are virtualized. Using networked storage will have a performance impact and introduces additional failure modes that the cluster cannot protect against. Simple network accessible storage (NAS) becomes a single point of failure that would prevent the entire cluster from functioning if it failed. A storage area network (SAN), while it might be more resilient to failure, unnecessarily replicates data that is already being replicated by the cluster and can have an extreme negative effect on cluster throughput.

In general, a three node cluster is recommended as the best trade-off between availability and deployment cost, when running the cluster in a single data center. See the description of using "sites" for an alternative approach when multiple geographically separated data centers are used.

An active/active cluster will require more hardware resources (CPU, RAM, I/O bandwidth and network bandwidth) than a single UM server to support the same throughput. These resources are required to physically run the additional servers in the cluster, transfer cluster state across the network, and store multiple copies of the complete cluster state. In high load situations it is recommended to deploy a separate network for intra-cluster traffic, to keep this separated from client (message publishing and subscribing) traffic.

When virtualization is used, it is highly recommended to allocate enough virtual resources to every node to handle the maximum expected load on the cluster, and to ensure these resources are not shared with any other virtual machine. This will help to prevent outages caused by a shortage of shared hardware resources during periods of high load.

Many virtual machine infrastructures have the ability to move virtual machines between physical hosts to automatically load balance (known as VMotion with VMWare). This capability is very attractive, but comes at a cost. While the virtual machine is being moved it will appear to be unavailable for a period of time, the length of which will depend on the network infrastructure and a number of other factors. If that period extends beyond a few seconds that can cause the UM cluster to regard the node as being unavailable and it will therefore be removed from the cluster. When the node then re-joins the cluster it will have to resynchronize with the other nodes, which can be expensive. Therefore if virtual machines are moved around routinely it can have a serious impact on cluster performance and availability.

Virtual machine (VM) live migration (VMware vMotion, Microsoft Hyper-V live migration) allows a perceived zero downtime migration of a running VM. This is a common capability, however it can complicate UM clustering as UM requires constant and timely responses due to having strong consistency guarantees. To deploy UM with infrastructure using live migration, it is imperative to deploy the infrastructure such that UM cluster communication is isolated from all storage, live migration and other infrastructure management traffic. It is highly recommended that it is on physically separate hardware.

The live migration scenario should be part of the load/stress testing to ensure that the cluster behaves as expected when migrations take place on the infrastructure it is deployed on.

## Working with an Active/Active 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.

An *active/active* Universal Messaging cluster achieves this by the implementation of some basic concepts described in the following sections.

### Client Connection

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 nSession object 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 realms transparently without the use of multiple RNAMEs.

### Client Failover Using Multiple RNAMEs

Using an array of RNAME URLs allows client applications to seamlessly fail over to different cluster realms 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 first RNAME (in this example, nsp://host1:9000) 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 the next RNAME in the list (in this example, nsp://host2:9000).

### The "Follow the Master" feature

The *follow the master* feature allows you to configure a client session so that it is always connected to the master realm in a cluster (the client will follow the master). This means that if a client is initially connected to the master realm, and then the master realm becomes unavailable, the client API will automatically reconnect the client application to the new master realm of the cluster.

Using the Universal Messaging client API for Java, this behavior can be configured through the method `setFollowTheMaster` of the `nSessionAttributes` class (Universal Messaging client API), or in the JMS ConnectionFactory. The default configuration is that the client will not follow the master.

You can also configure the client to follow the master by setting the system property `FollowTheMaster` on the client side when you start your client application. To do this, use "`-DFollowTheMaster=true`" on the command line. Alternatively, you can invoke `java.lang.System.setProperty("FollowTheMaster", "true")` in your client application code.

When a realm (let's call it realm A), that is currently not the master, receives a client connection request with *follow the master* enabled, realm A will request the current master (let's call it realm B) to send a list of interfaces that the client can use to connect to realm B. When realm B builds the list of interfaces that the client can connect to, it checks that each such interface has the attributes

`Advertise Interface` and `Allow Client Connections` enabled before adding it to the list being sent back to the client. The combination of these two attributes on an interface allows clients to access the realm via the interface.

If you have an interface that you want to use exclusively for realm-to-realm communication in the cluster (thereby disabling client-to-cluster communication on the interface), it is recommended to disable the `Allow Client Connections` attribute, as this will stop all connections except for realm-to-realm communication. This setup leads to added transparency about which client or realm is connected to which interface. You might also want to consider adding firewall rules that protect connections using non-SSL interfaces, so that only known clients (and the other cluster realms) can use such interfaces.

**Note:**
If a client has activated the "Follow the Master" behavior by use of any of the above mentioned techniques, the protocols (`nsp`, `nsps`, etc.) in the RNAME list used by the client for the *initial* connection will not necessarily be the same protocols that will be offered following a redirect to a newly elected master realm. For example, if the realms in the cluster all offer both `nsp` and `nsps` interfaces, it is possible for a client to be offered an `nsp` interface of a newly elected master realm even if the RNAME list the client used only contained `nsps` RNAMEs. This may not be what the client intended and in some cases may open a potential security risk.

**Note:**
If the realm configuration property `ClusterMode` (available in the `Cluster Config` property group) is specified as "Replication", the system property `FollowTheMaster` on the client side will be ignored. See the section "Cluster Modes" on page 178 below for details.

See the section *Basic Attributes for an Interface* in the *Administration Guide* for information about interface attributes. See also "Separating Client and Cluster Communication" on page 190 for related information.

## Cluster Modes

You can specify whether non-admin clients are allowed to connect to nodes other than the master node in a cluster using a cluster mode configuration that is set by the `ClusterMode` realm server property. This property can have the value "Replication" or "Active".

Admin clients can connect to both master and non-master nodes, irrespective of the `ClusterMode` configuration.

- Cluster mode: "Replication"

  If `ClusterMode` is set to "Replication", non-admin client connections can be made to **only** the master node in the cluster. In this mode, nodes that are not the master node will reject all non-admin connections.

  If a non-admin client is initially connected to the master realm, and then the master realm becomes unavailable, the client will be disconnected from the server. If you are using Universal Messaging client APIs to establish the connection, the client will automatically reconnect to the new master realm of the cluster.

In Replication mode, the RNAME list used by the client for connection purposes must contain the RNAMEs corresponding to all of the nodes in the cluster. The client will attempt to form a connection to the first RNAME in the list, and if the connection is refused because the RNAME is not the RNAME of the master node, the client re-connection logic will automatically try the next RNAME from the session's list. This automatic procedure continues until the correct RNAME for the master node is found, and the connection will take place.

The `FollowTheMaster` setting on the client side will be ignored if you have set `ClusterMode` to "Replication".

If Replication mode is enabled, you'll not be able to make use of local resources (e.g. local queues, channels, etc.) that exist in the server nodes that are part of the cluster. Client connections can do operations on the local resources in the master node, but the resources are not accessible when the master is changed to a different node.

**Note:**

If you are using C# or C++ client version 10.3 or 10.5, make sure that the client is upgraded with the latest client libraries before enabling Replication mode. Also, C#/C++ clients in v10.1 and earlier cannot make use of the feature.

Universal Messaging client APIs for C# and C++ in v10.3 and v10.5 will require an upgrade to the following fix levels to be able to work properly in the Replication cluster mode:

- v10.3: Fix 19
- v10.5: Fix 10

The connect/reconnect behavior of these clients can be broken if not upgraded.

Java clients in versions earlier than v10.7 can connect to a server where the Replication cluster mode is configured. However, we recommend upgrading to the fix levels mentioned above since they have some optimizations with regard to the feature.

All other protocol clients are compatible with this feature.

- Cluster mode: "Active"

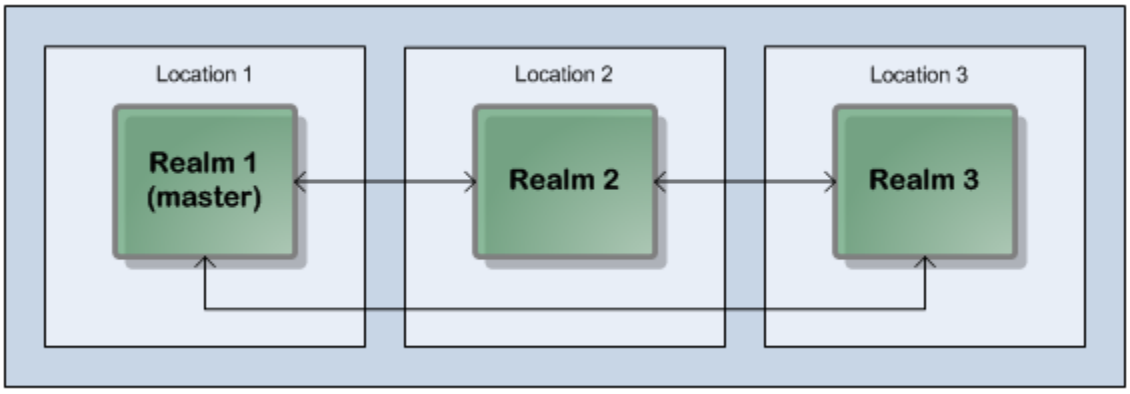
  If `ClusterMode` is set to "Active", which is the default value, client connections can be made to both master and non-master nodes in a cluster. The clients can also make use of the `FollowTheMaster` configuration, which allows specific clients to connect to only the master node in the cluster.

See also the summary of the `ClusterMode` property in the list of realm server properties at *Realm Configuration* in the *Administration Guide*.

## Masters and Slaves

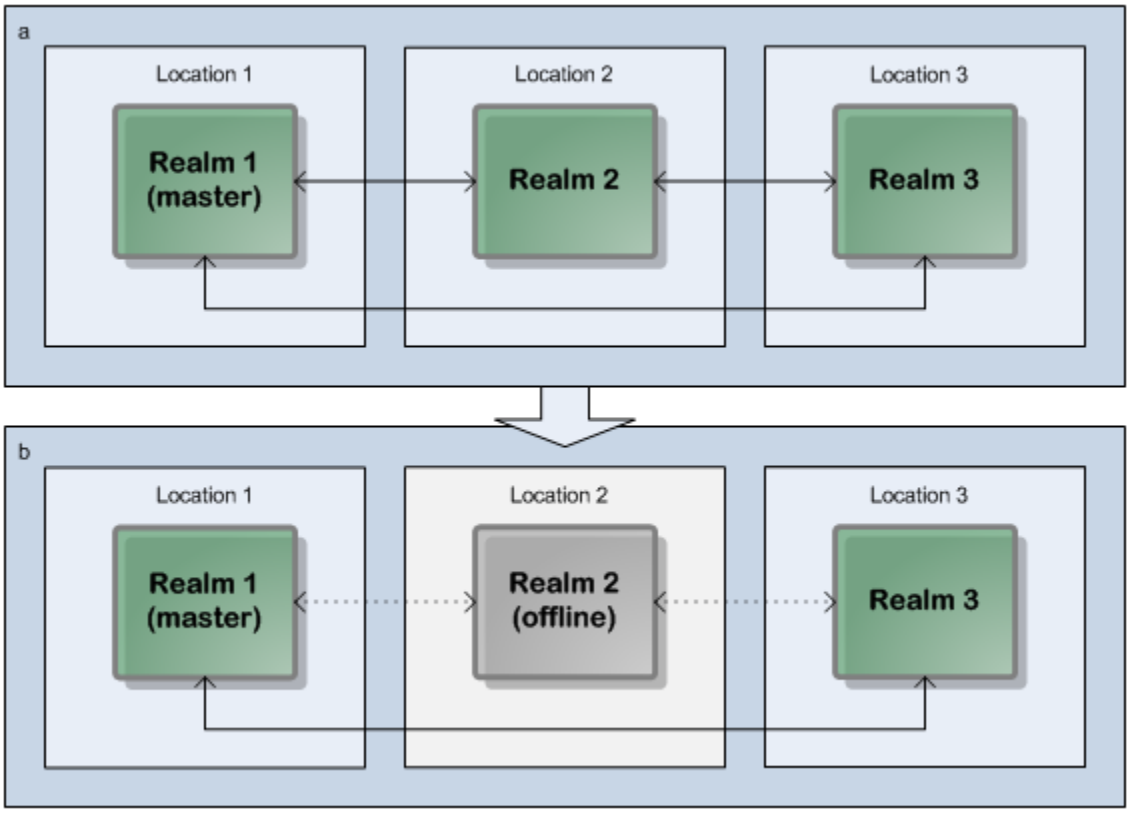
As explained in the Clustering Overview (see "Clusters: An Overview" on page 168), a *cluster* is a collection of Universal Messaging Realm Servers (realms).

*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" on page 145), each published event will be allocated a unique event ID by the master, which is then propagated to each slave.

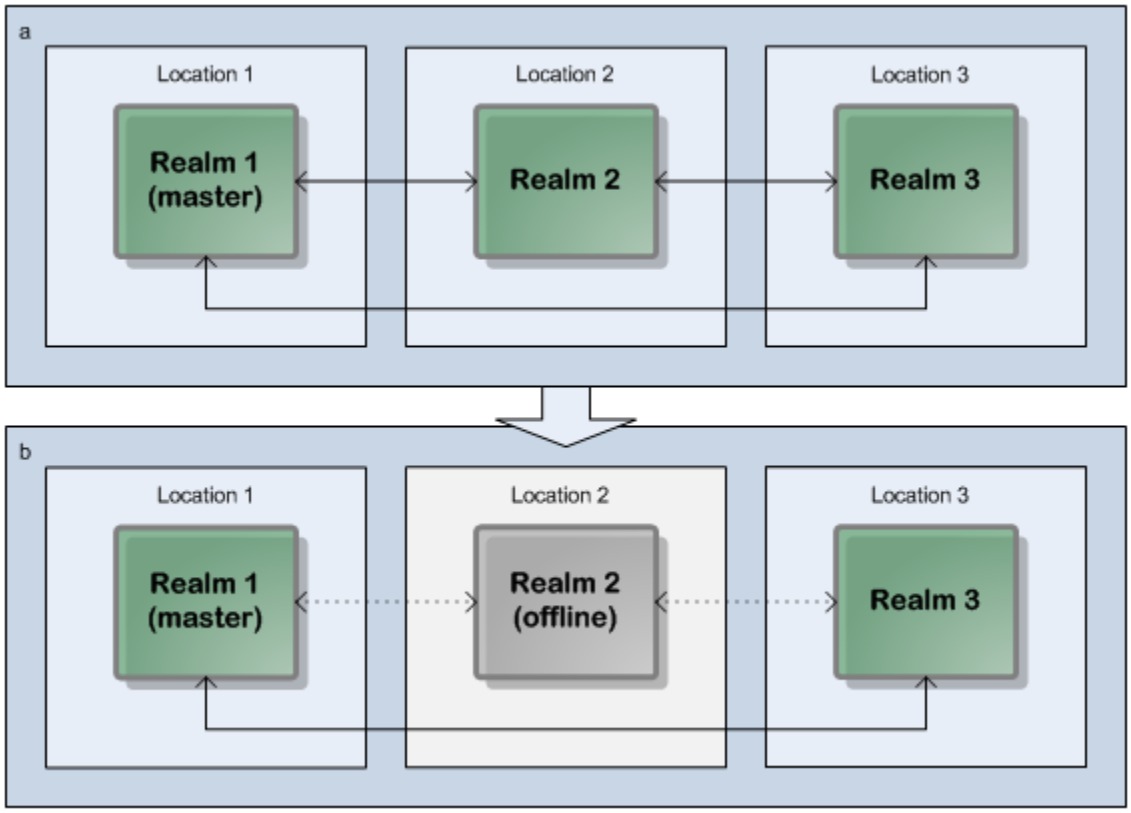
## Quorum

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.

There is an exception to the 51% rule if you use Clusters with Sites. This allows quorum to be achieved in certain circumstances with just 50% of the cluster nodes online and intercommunicating. See the section "About Active/Active Clustering with Sites" on page 195 for details.

### 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 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 182).

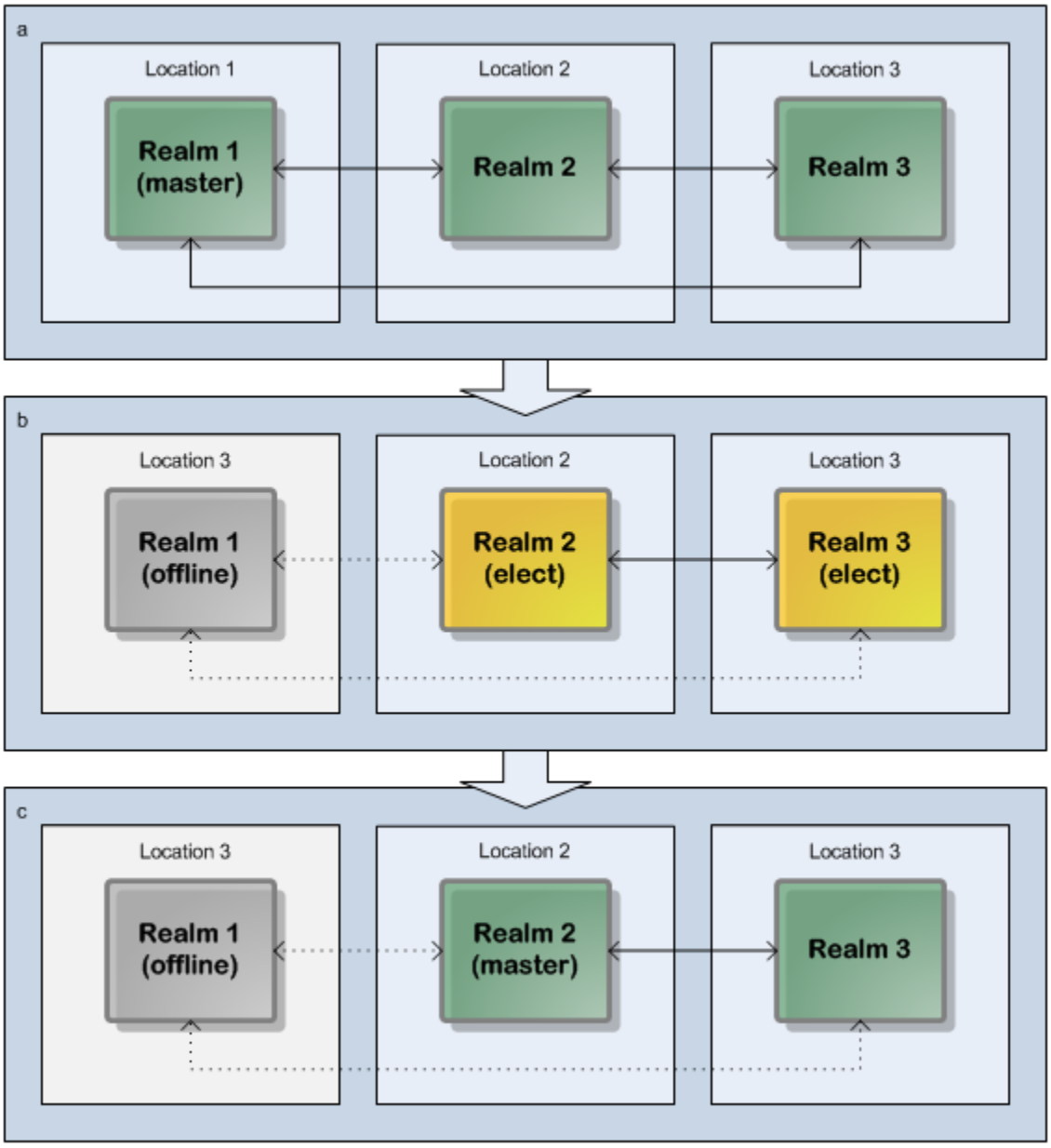
### Election of a new Master

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 (see "Quorum" on page 180, 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:

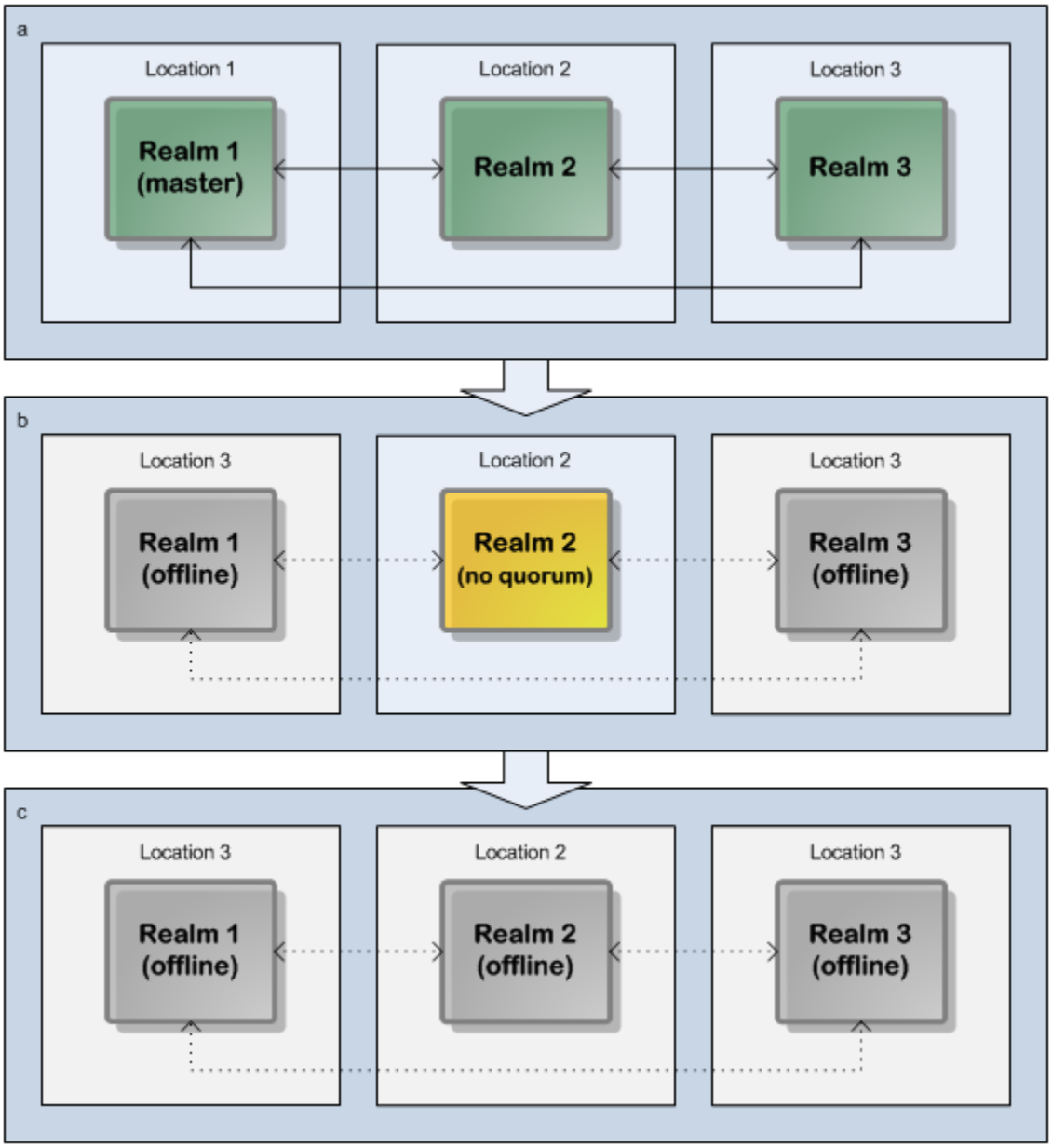



*Three-realm cluster over three locations: quorum permits election of a new Master and cluster continuation.*

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

## Examples: Insufficient Quorum for Master Election

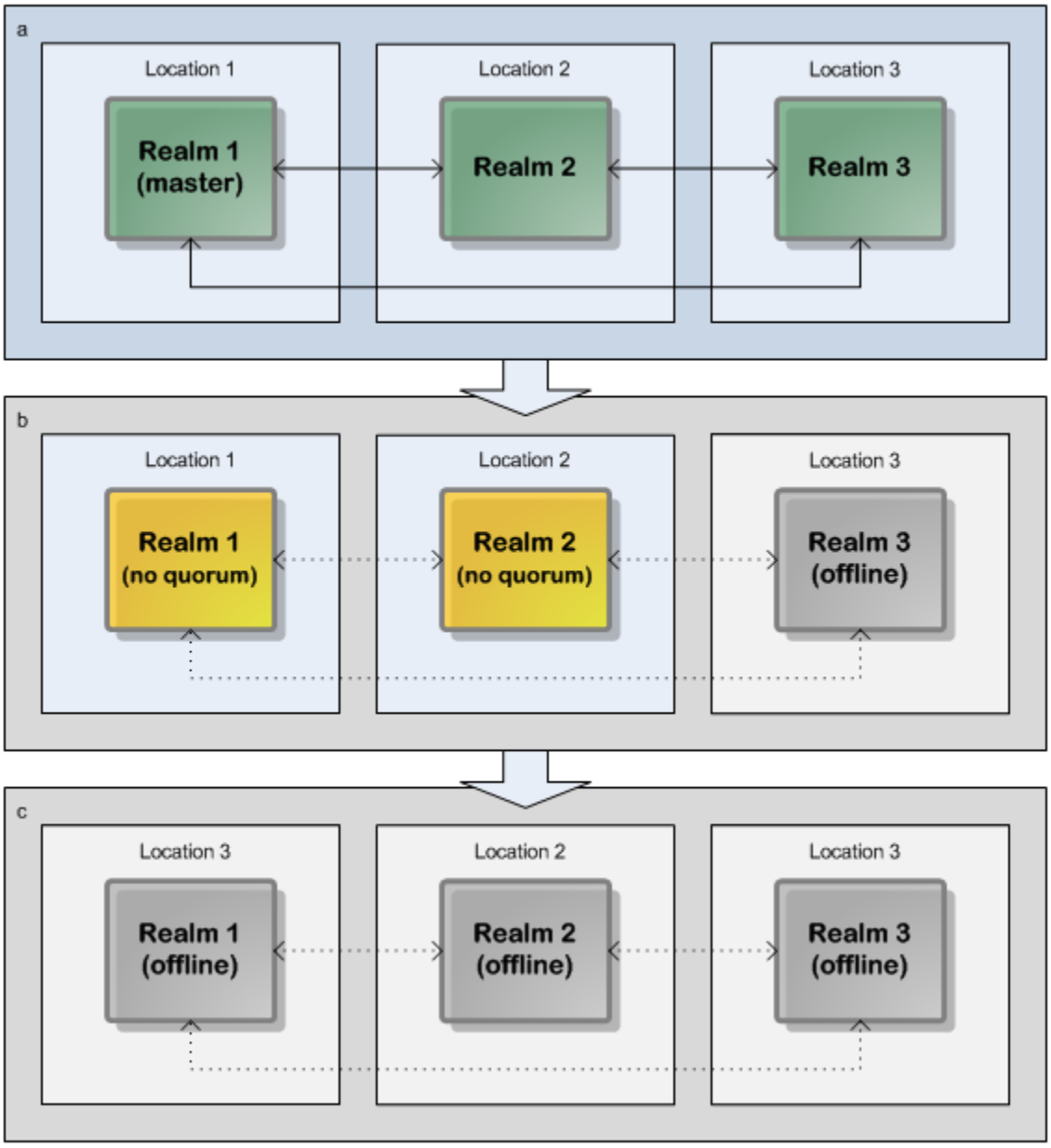
In this example, we again examine a three-realm cluster, distributed across three physical locations. In this case, both the master realm and one slave realm have failed, so the remaining realm represents only 33% of the cluster members (which does not satisfy the 51% quorum minimum requirement). As 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.

If you publish to the master and subscribe from both master and slave nodes, the master will simply pass the event on to 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.

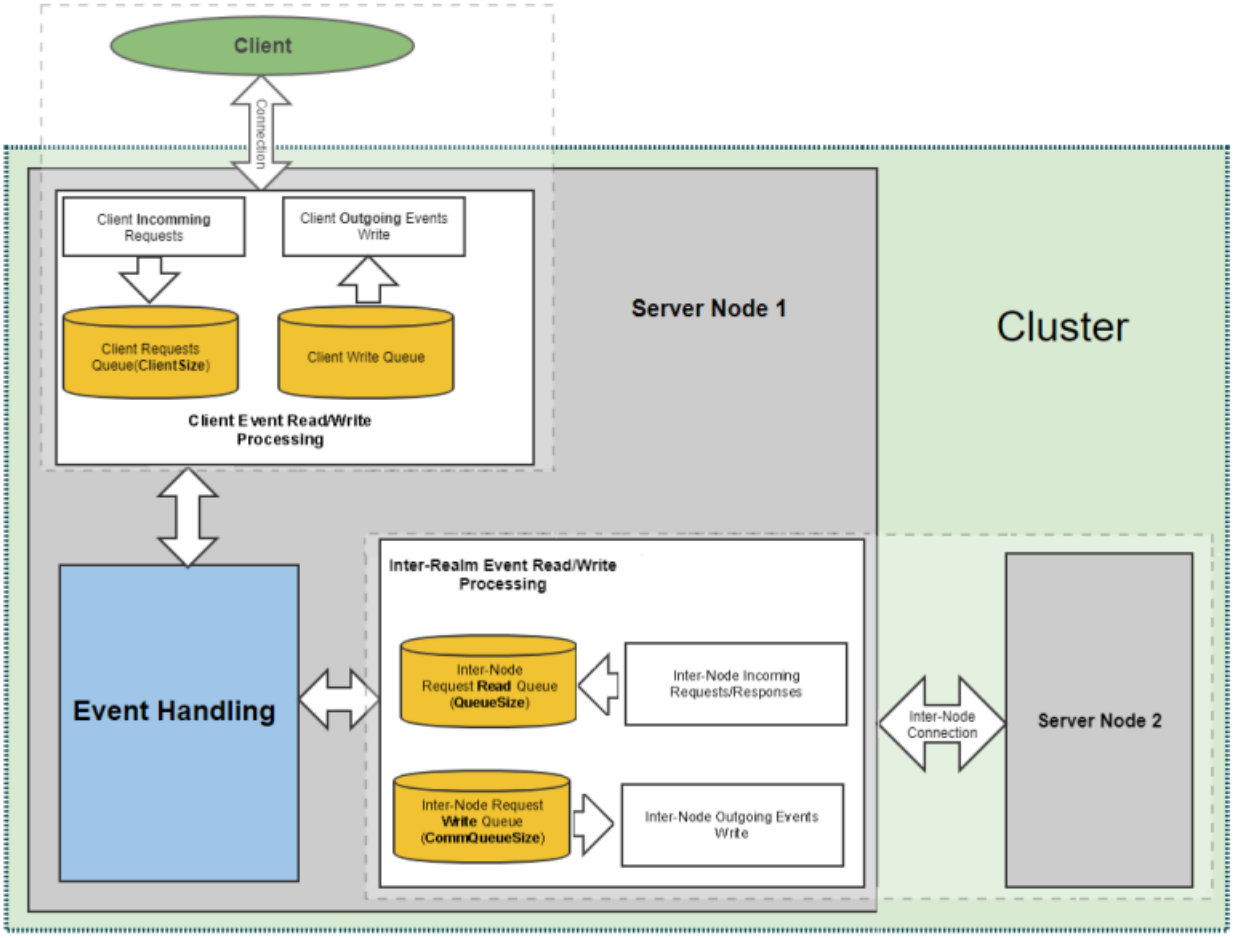
If the first event happens to be consumed by a consumer on the master realm, each slave realm will be notified of the consumption of the event from *somequeue* and thus remove the event from its own local copy of *somequeue*.

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.

## Inter-node data flow

The following diagram shows the data flows between nodes in an active/active cluster.




The numerical values `ClientsSize`, `QueueSize` and `CommQueueSize` shown in the graphic are metrics that can be viewed in the server log. For further information, refer to the section "Periodic Logging of Server Status" on page 58

## 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, it automatically moves 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 168). 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 standalone 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 Universal Messaging's 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.

**Note:**
Inter-cluster connections and realm federation are mutually exclusive, and they *cannot be used together* in any combination.

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

## Setting Up a High Availability (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 168) 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 set up 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 RNAME URLs:

```
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 RNAME URLs.

### 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 RNAME URLs (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.*

## Separating Client and Cluster Communication

When configuring Universal Messaging for clustering, it is not essential but it is often recommended to have a dedicated interface for client communications and a dedicated interface for cluster communications. There are various reasons for this, as described in this section.

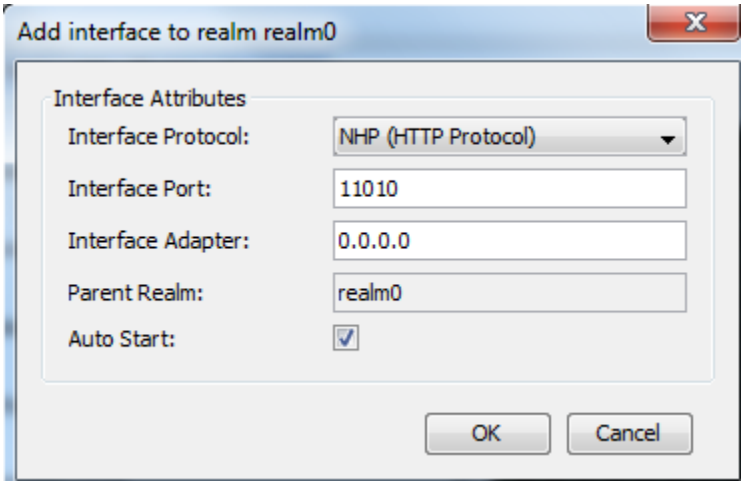
Occasionally it is necessary to stop a client connection while some operation is being carried out on the server or cluster. For example, you may want to delete and recreate a channel, so you need to stop your applications from publishing while this is done. However, you do not want to stop the cluster communication during this time, so it is good practice to separate your cluster communication from client communication. Replication of data, state and other cluster negotiation traffic is very high priority so we need to ensure that it is not delayed in any avoidable way. If the cluster cannot form, then no client traffic will be able to flow, so it is vital that the client traffic cannot impact the cluster formation. If you have lots of clients constantly trying to access the server and getting disconnected because the cluster is not formed, they will be competing to connect at the same time as nodes within the cluster are trying to connect to one another.

By adding separate interfaces onto your servers you can separate client communication from cluster communication and also disable client communication if necessary.

There is no server-defined limit on the number of interfaces that you can create, so it might be a good idea to have different interfaces for different applications. This will allow you to easily disable connections from particular applications.

## Adding a Dedicated Interface for Cluster Communication

The first thing to do is ensure that you have more than one interface per server. You can add an interface by selecting the server under the **Realms** node in Enterprise Manager, then opening the **Comms** panel. Click the **Add Interface** button to bring up the dialog for adding an interface.




In the dialog, enter the interface details as required. Also ensure that the **Auto Start** checkbox is enabled, so that if you stop and start the server, your interface is automatically enabled.

Now you will have two interfaces. In this example we have two NHP interfaces, one on port 11000 and one on port 11010. We are going to make the interface running on port 11010 the interface used for cluster communication. The default settings will allow the interface to be used for cluster communication (and the nodes will automatically be notified about this new interface). We recommend you to disable **Advertise Interface**, which is available as a checkbox under the panel **Comms > Interfaces > Basic**. This will stop the interface information being sent to clients.

**Note:**
Do *not* disable the checkbox **Allow Client Connections** under **Comms > Interfaces > Basic**; this is important because Administration API connections, e.g. Enterprise Manager, still count as client connections and we always want to be able to connect an Enterprise Manager. You can optionally disable **Allow for InterRealm** on the *other* interface but it is not necessary.

So now we have our interfaces set up correctly and the interface for cluster communication will not be advertised for clients to use. It is now the responsibility of the system administrator to ensure that the cluster interface is not used by clients for normal connections; all applications should use the other interface.

### Temporarily Disconnecting the Client Interface

Let us assume that we want to temporarily disable client connections. All we have to do is stop the interface that we are using for out client connections. This is the best way to stop client traffic, but an alternative is to disable the **Allow Client Connections** checkbox temporarily.

**Note:**
your Enterprise Manager may be connected to the client interface, so if you stop this interface or disable client communication, you will get disconnected. It is important that you still allow client connections on the cluster interface simply so that you can connect an Enterprise Manager to that interface while your other applications remain disconnected.

Once you are ready to allow clients back in, you can re-enable the client interface.

### Further Considerations

Network partitioning

Rather than relying on clients not knowing about the cluster interface, it is often a good idea to bind that interface to a different network or VLAN. This is done by specifying an adapter to bind to rather than using 0.0.0.0. This way you restrict connections on this interface to users within that network. It is in fact best practice to bind to a specific network interface rather than 0.0.0.0, especially if running containers.

Stricter Access Control

In the example above we simply ensure that clients do not know about the cluster interface. To enforce this you could use network partitioning as explained above, or you could restrict access using ACLs or VIA entries. It is not possible to set interface-specific ACL entries, so to restrict connections you would need to temporarily stop them connecting using the server level ACL. To do this you would add a specific ACL entry for your administration-related connection and then disable access for everyone else.

Another alternative is to use VIA entries, which are set on individual interfaces. A VIA entry lets you restrict what interfaces a user and host are allowed to connect to. You would need to add an entry that covers all clients into the VIA list for the client interface and leave the VIA list empty for the cluster interface.

For more information on VIA entries, see *About Interface VIA Lists* in the Administration Guide.

One more alternative is to use three interfaces, an administration interface, a cluster interface and a client interface. You could disable client communication on the cluster interface to limit traffic to cluster communication only. You could then use SSL on the administration interface and only provide administration users with the certificate required to connect. This way you can disable the client interface to stop client communication and no client will be able to connect to the cluster interface at all.

### Deleting a Cluster

If you decide to delete a cluster that you created previously, all cluster resources that exist in all cluster member realms will also be deleted.

Deleting a cluster is a simple operation that can be achieved using the Enterprise Manager or Command Central.

# Active/Passive Clustering

## About Active/Passive Clustering

### Introduction

Active/passive clustering is a concept that uses clustering software and special purpose hardware to minimize system downtime. Active/passive clusters are groups of computing resources that

are implemented to provide high availability of software and hardware computing services. Active/passive clusters operate by having redundant groups of resources (such as CPU, disk storage, network connections, and software applications) that provide service when the primary system resources fail.

In a high availability active/passive clustered environment, one of the nodes in the cluster will be active and the other nodes will be inactive. When the active node fails, the cluster can fail over to one of the inactive nodes. This can be automated using dedicated third-party software, allowing you to start the resources on the redundant node in a predefined order (or resource dependency) to ensure that the entire node comes back up correctly.

Universal Messaging can run in an active/passive cluster environment, under Windows or UNIX. This approach does not provide load balancing or scalability.

### What is active/passive clustering?

An active/passive cluster has multiple UM server nodes that share the same "data directory", the storage area where the configuration and persistent message data for the cluster are kept. In contrast to the active/active approach, only one of the nodes in an active/passive cluster will be running at once.

If the running (active) cluster node fails, one of the non-running (passive) nodes can be started to replace it. Identification of the failure of the active node and the starting of the replacement node must be orchestrated by a system external to UM such as third-party clustering software or virtual machine infrastructure. Because all of the cluster nodes access the same data directory, the new node will be able to resume processing where the failed node left off. As with the active/active approach, all clients connected to the failed node will need to fail over to the new one. Again, provided that the clients are correctly configured this process is transparent to the applications and no messages will be lost.

Typically, active/passive clusters will use load balancers and third-party clustering software to manage failover and hide the fact that only one node is active from clients, but this is not required.

The active/passive clustering capability provided by Universal Messaging is essentially identical to that offered by the webMethods Broker. Any approach that has been used successfully to implement active/passive clustering with the Broker should also work with UM, with minimal changes.

### Active/passive clustering requirements

You need the following to configure a Software AG Universal Messaging active/passive cluster:

- Cluster control software to manage the clusters on Windows or UNIX.
- Shared Storage for sharing data files.
- IP address for running the Universal Messaging cluster service.
- Universal Messaging installed on the cluster nodes in the same directory path (for example, C:\SoftwareAG). In the installations, the data directory path for the shared storage must be the same.

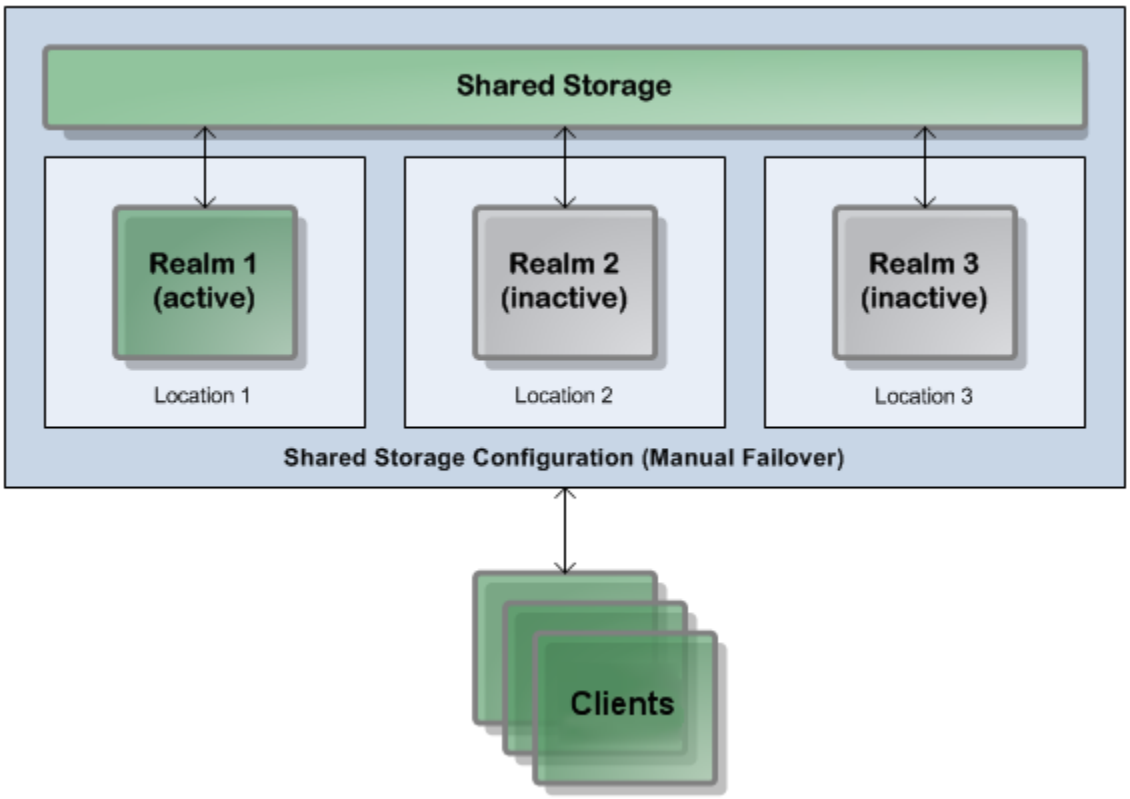
**Note:**
Universal Messaging installation must be identical on all cluster nodes. All instances of Universal Messaging must point to the same Universal Messaging storage files on the shared storage.

## Shared storage configurations

In a Shared Storage configuration, multiple realm servers point to the same data directory (i.e. they share a single virtual or physical disk), but only one of the realm servers is online at any time.

In fact, shared storage must only be accessed by the active node at any one time! Multiple realm server processes accessing the same shared storage directory are likely to lead to data loss.

Shared Storage configurations are not technically a cluster, but they do provide the basic mechanism for rapid failover between realms:




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

## Universal Messaging capabilities for active/passive clustering

The following capabilities of Universal Messaging enable the vendor-specific cluster control software to monitor and manage Universal Messaging in an active/passive cluster.

- Functionality to start, stop, and monitor the servers.
- Ability to store the server's state information and data on a shared disk.
- Ability to survive a crash and restart itself in a known state.

- Ability to meet license requirements and host name dependencies.

### How should an active/passive cluster be deployed?

As with an active/active cluster, it is essential that each node in an active/passive cluster is deployed on physically separate hardware, to reduce the risk of a single hardware failure affecting multiple cluster nodes. When virtualization is being used, this means that each node must run on a virtual machine that is pinned to a different physical host. In the case of blade servers, it is recommended that each cluster node runs on a blade in a different chassis.

However, in contrast to the active/active case, the storage used by active/passive cluster nodes must be shared. Every node in the cluster will be configured to use the same network-accessible storage location for its data directory. This storage should be resilient against failures so it does not become a single point of failure for the cluster. Typically this means that a replicated SAN with a dedicated network should be used rather than a simple NAS.

When virtualization is used, it is highly recommended to allocate enough virtual resources to every node to handle the maximum expected load on the cluster, and to ensure these resources are not shared with any other virtual machine. This will help to prevent outages caused by a shortage of shared hardware resources during periods of high load

## Working with an Active/Passive Cluster

Active/passive clustering is a solution that uses third-party clustering software and special purpose hardware to minimize system downtime. Active/passive clusters are groups of computing resources that are implemented to provide high availability of software and hardware computing services. Active/passive clusters operate by having redundant groups of resources (such as CPU, disk storage, network connections, and software applications) that provide service when the primary system resources fail.

The procedures required to set ting up the active/passive cluster involve steps that are dependent on the third-party solution you have chosen (for example, Windows Server®, Veritas™, HP ServiceGuard®, IBM® HACMP™, or Oracle® Solaris Cluster) . Ensure that you have all required information about the third party product before you begin.

The procedures for setting up the active/passive cluster are described in the section *Setting up Active/Passive Clustering with Shared Storage* in the *Administrator Guide* .

# Active/Active Clustering with Sites

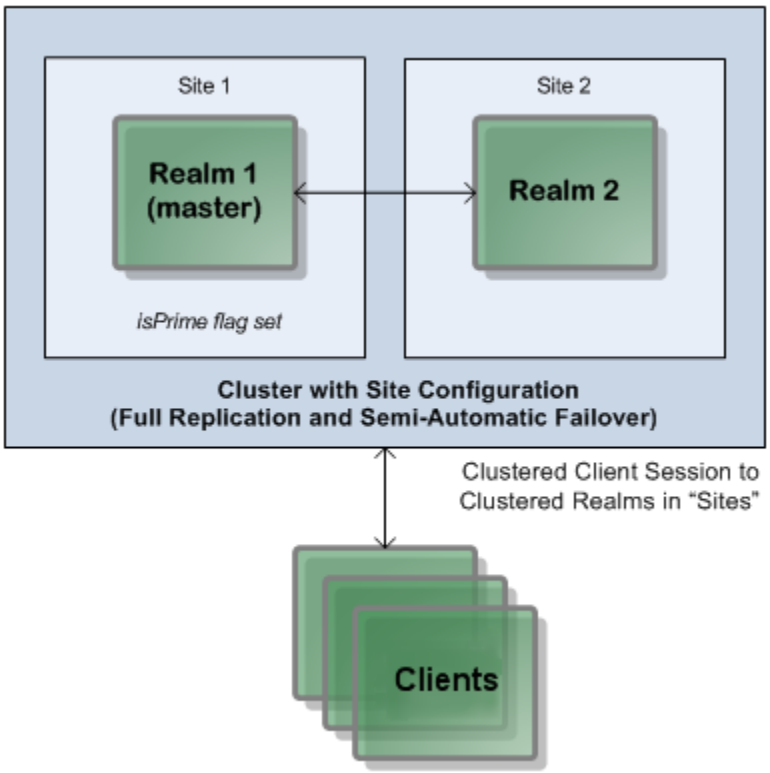
## About Active/Active Clustering with Sites

*Sites - an exception to the Universal Messaging Cluster Quorum Rule (see* "Quorum" on page 180*).*

### The requirement for clusters with sites

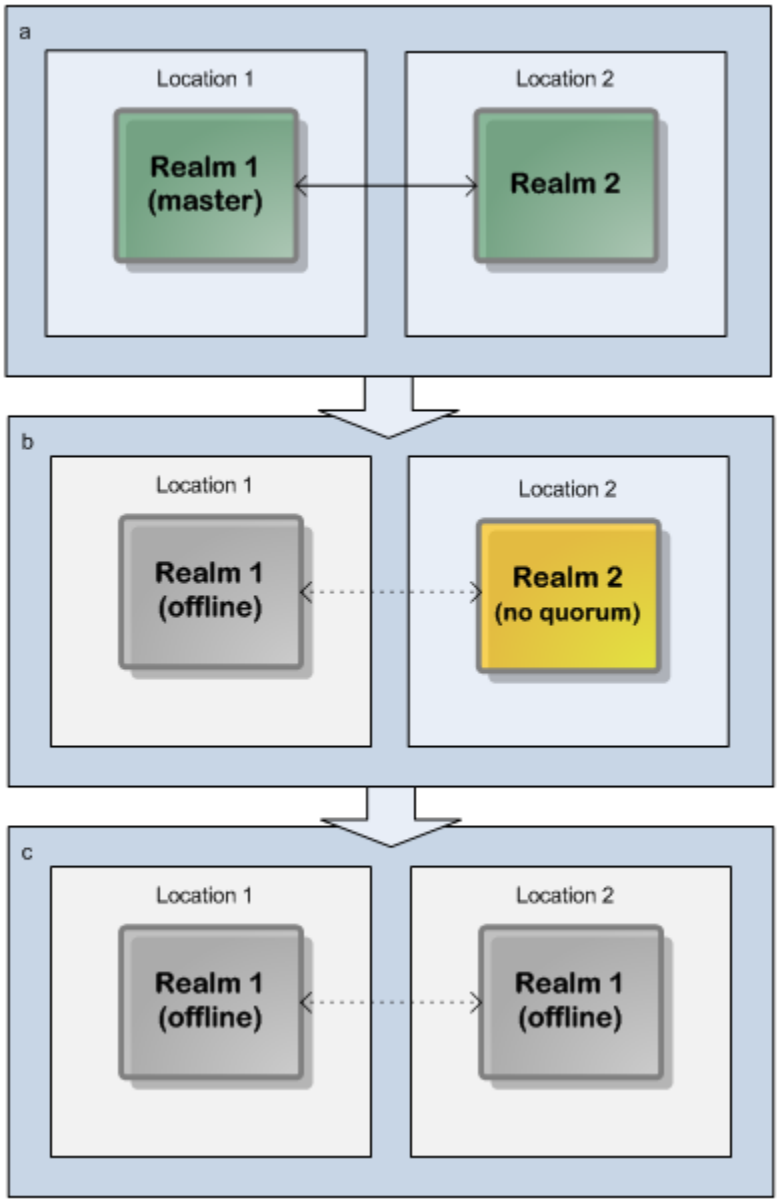
Although our recommended approach to deploying a cluster is a minimum of three locations and an odd number of nodes across the cluster, not all organizations have three physical locations with

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 site and a backup site:




*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 not achievable 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 is addressed by *Universal Messaging Clusters with Sites*.

## Overview of clusters with sites

Universal Messaging sites can be considered if an active/active cluster is being deployed across multiple geographically distributed data centers. A typical sites-based deployment has two cluster nodes in each of two sites. The basic concept of clusters with sites is that if only two physical locations are available, and Universal Messaging clustering is used to provide high availability and disaster recovery, it is possible to allow a single site to continue to operate when the other one is unavailable.

This is achieved by allowing an additional vote to be allocated to either of the sites in order to achieve the required cluster quorum of 51%.

If, for example, you have two sites and each site contains just one realm, making a total of two realms in the cluster, the additional vote for one of the sites raises the availability from "one out of two" to "two out of three", that is 67%, so the required quorum of 51% is achieved.

As a further example, if you have two sites and each site contains two realms, making a total of four realms in the cluster, the additional vote for one of the sites raises the availability from "two out of four" to "three out of five", that is 60%, so the required quorum of 51% is achieved.

The site that gets the additional vote is called the *primary* site. There can only be one primary site at any given time in a cluster, and any other sites in the cluster are *non-primary* sites. If the sites become disconnected from each other, the primary site will continue to operate and the whole cluster will re-synchronize when the connection is restored.

This approach provides:

- Transparent client failover
- Semi-transparent server failover
- Load balancing and scalability

The quorum rule of availability of more than 50% servers in the cluster is achieved by defining the servers in two sites (primary and backup), and by allocating an additional vote (the "IsPrime" flag) to one of these sites.

The value of the IsPrime flag in a site indicates whether the primary site or the backup site as a whole can cast an additional vote. The failover is automatic if the site where the IsPrime flag is set to "false" fails. If the site where the IsPrime flag is set to "true" fails, you must manually set the IsPrime flag to "true" on the active site and perform manual failover.

It is essential that the IsPrime flag is only ever applied to a single site at once. This means that the flag should only ever be set manually, after confirming that the other site is genuinely down, not just disconnected. Having two IsPrime sites can lead to an unrecoverable situation called “split brain” that will lead to data loss, so extreme care must be taken to avoid this when using sites.

In a cluster with a production site and a disaster recovery site, you can make either site the primary site, but Software AG recommends that you make the production site the primary site due to the following considerations (assuming a setup with equal numbers of realms on each site):

| Primary Site | Effect |
|---|---|
| Production site | If the disaster recovery site fails, the production site continues to provide 50% of the available realms in the cluster. Since the production site is the primary site, it gets an additional vote, so the cluster can continue to run.<br><br>If the production site fails, the disaster recovery site cannot take over automatically because it cannot achieve the quorum of 51%. This situation requires manual intervention to set the disaster recovery site to be the primary site, so that the cluster can be restarted on the disaster recovery machine. |
| Disaster recovery site | The idea behind setting the disaster recovery site to be the primary site is that if the production site fails, the disaster recovery site can achieve quorum and immediately take over from the production site.<br><br>This may appear at first to be a good setup, but has a big disadvantage: If the production site is not the primary site, a failure on the disaster recovery site would cause the production site to halt, since the production site by itself cannot achieve a quorum of 51%. This is clearly not what a disaster recovery setup is intended for - a failure in the recovery machine shouldn't halt the production machine. |

Note that Universal Messaging does not support selecting the master among the realms of the primary site. If the master node is stopped, it is not guaranteed that the new master node will be a node from the primary site.

### How sites determine if a cluster can form

The general rule regarding the effect of sites when forming a cluster is as follows: If *exactly* 50% of *all* realms in the cluster are online, and if *all* primary site realms are contactable, a cluster can be formed. In all other cases, a cluster can only form if at least 51% of the cluster's realms are contactable. All other quorum and voting rules are identical, with or without sites.

### Defining Sites in the API and in the Enterprise Manager

Within the Universal Messaging Admin API, and specifically within a cluster node, you can 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. When the *isPrime* flag is activated for a site, the site becomes the *primary* site.

Setting the *isPrime* flag can also be achieved using the Enterprise Manager.

### Examples: Achieving quorum using Universal Messaging clusters with sites

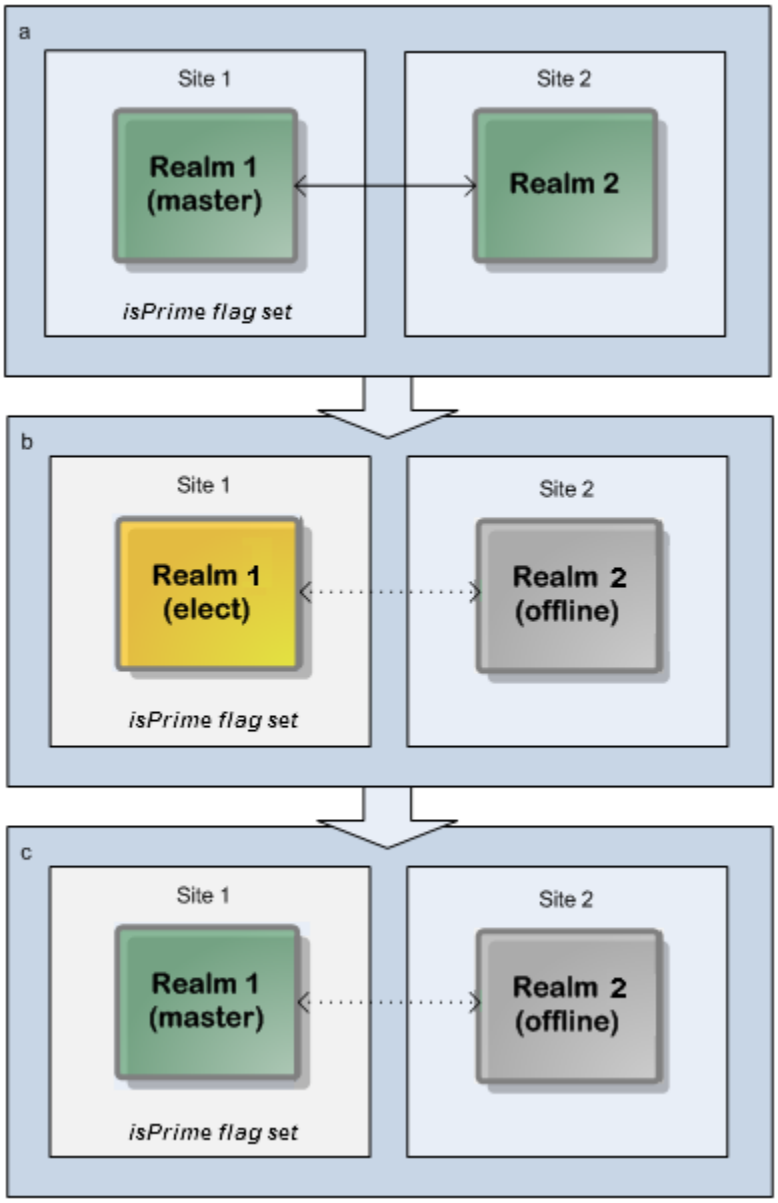
Consider an example scenario where we have a cluster across two physical locations: the default production site and a disaster recovery site, and each site has one realm. Without using sites, this configuration wouldn't be able to satisfy the 51% quorum rule in the event of the loss of one location/realm. The same technique can be used for sites with as many realms as required.

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

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.

**Example with production site as primary site**

Here is the situation if the production site is the primary site (the recommended setup) and the disaster recovery node fails. In the diagram, two servers are configured in just two sites: primary (master) and backup (slave). The IsPrime flag is set to true in the primary site. If the server in the backup site becomes unavailable, the cluster continues to work with the server in the primary site because the primary site has an additional vote to achieve the quorum rule of more than 50% available servers.



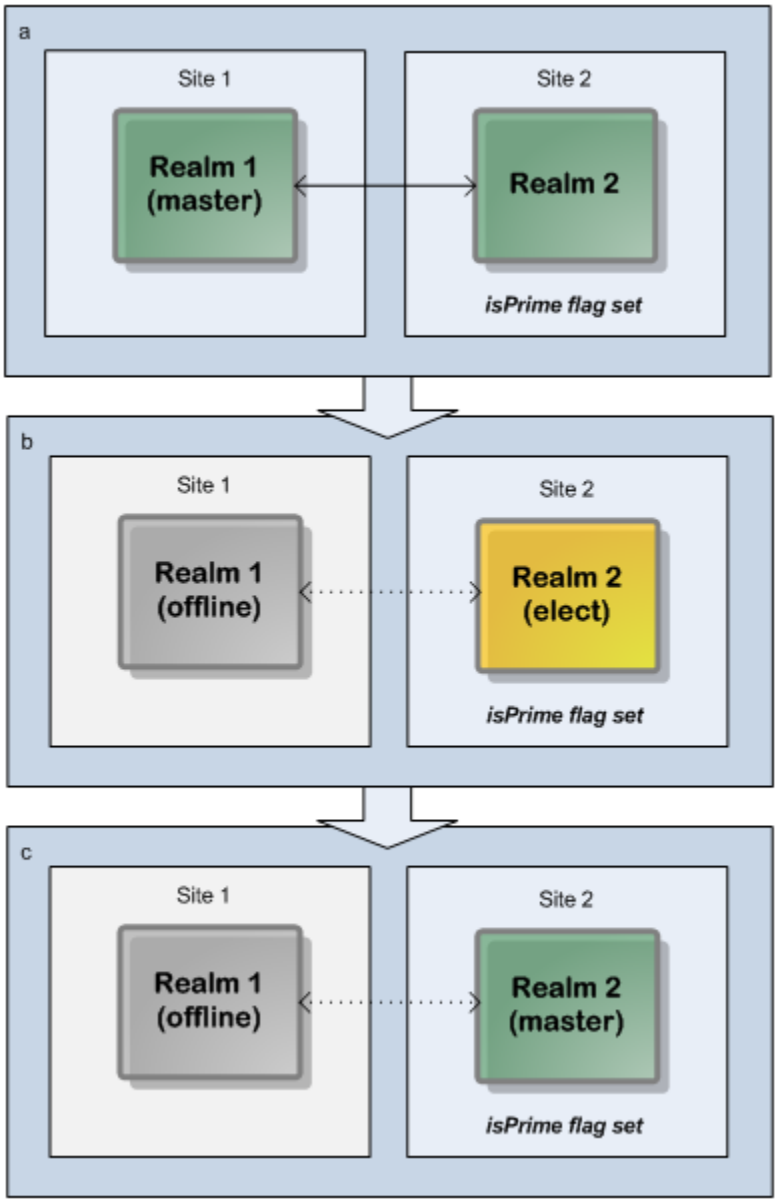

*Two-realm cluster over two sites: Production site as primary site makes a 51% quorum achievable if non-primary site fails.*

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

In this scenario, the cluster's master realm is initially on the production node and stays on the production node after the election of the new master.

**Example with disaster recovery site as primary site**

Here is the situation if the disaster recovery site is the primary node (as stated earlier, this is not the recommended setup) and the production node fails:




*Two-realm cluster over two sites: Disaster recovery site as primary site makes a 51% quorum achievable if non-primary site fails.*

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

In this scenario, the cluster's master realm is initially on the production site but the new master realm is on the disaster recovery site.

If the connection to the server in the primary site is lost when the server on the backup site is active, you must manually set the IsPrime flag to true in the backup site so that the server in the backup site can achieve quorum.

**Note:**

Switching the primary site MUST be a manual operation by an administrator who can confirm that the previous primary site is indeed down and not merely disconnected from the other sites. Attempts to automate this process raises the risk of "split brain" situations, in which loss of data is very likely.

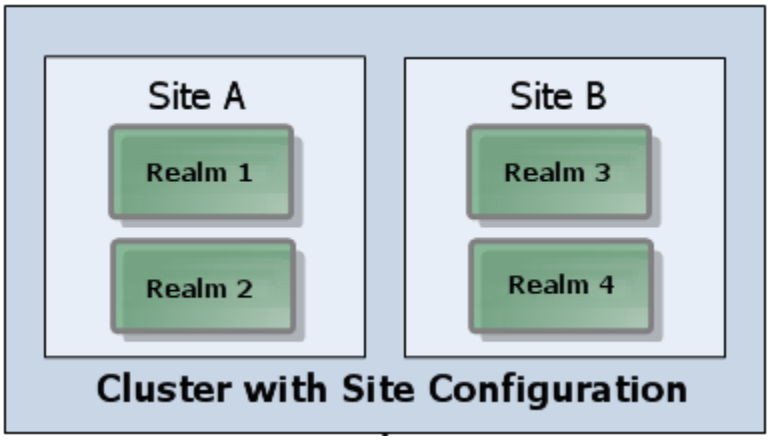
## Disaster Recovery using Clusters with Sites

Disaster recovery situations can be managed by using *sites* in Universal Messaging. In these situations, provided that 50% of the nodes are online, it is possible for the cluster to remain operational.

A cluster with sites allows a standard cluster to operate with as little as 50% of the active cluster members (as opposed to the standard 51% quorum in effect for clusters without sites), and provides administrators with a mechanism to prevent the split brain scenario that would otherwise be introduced when using exactly half of a cluster's realms.

When using sites, you should always segregate a cluster into exactly two sites. Each site within the cluster should have exactly the same number of nodes present. You also define one of the sites to be the *primary site*. Note that Universal Messaging does not support selecting the master among the realms of the primary site. If the master node is stopped, it is not guaranteed that the new master node will be a node from the primary site.

In the sample scenario below, the instructions for disaster recovery using sites assume a 4-node cluster split across two sites: Site A and Site B. Site A is initially the prime site.




### Disaster Recovery when the Non-Prime site is lost

When the non-prime site is lost, Universal Messaging will automatically ensure that the cluster is formed with the prime site. The administrator of the system does not need to undertake any action and the cluster will continue to function as normal, providing that at least 50% of the nodes are online to establish a quorum.

When the realms within the non-prime site are brought back online, they will rejoin the cluster and continue to function as normal.

## Disaster Recovery when the Prime site is lost

When the prime site is lost, Universal Messaging will not automatically reform the cluster. This is to ensure that a split-brain scenario, where both sites continue to operate independently of each other with conflicting state, does not occur.

Instead the administrator of the system must switch the prime site of the Universal Messaging cluster manually to allow the cluster to reform.

The steps to do this are:

1. Shut down all members of the former prime site (Site A).
2. Change the value of the prime site to the non-prime site (Site B).
3. Observe that a quorum has been achieved using members of new prime site (Site B) only.

When the realms within the former prime site (Site A) are ready to be brought back online, there are additional steps to ensure that they rejoin the existing cluster correctly:

1. Bring each member of the former prime site (Site A) up individually.
2. Ensure that each member joins the new cluster as a slave and acknowledges the change in the prime site flag before bringing up further members of the cluster.
3. Once all members of the cluster are online, ensure that a quorum is achieved and each node agrees on the master and prime site.

It is important that these instructions are carried out exactly to ensure that only one cluster remains active at any one point in time. Situations where two prime sites exist at the same time may cause 2 separate clusters to form. In these situations it is difficult for these nodes to resynchronize and agree state.

For information on how to perform these steps, see the following topics in the section *Cluster Administration* of the documentation of the Enterprise Manager:

- "Creating Sites for a Cluster"
- "Viewing Cluster Information"

# 9 MQTT: An Overview

MQTT (Message Queuing Telemetry Transport), is a publish/subscribe, simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth by IBM / Eurotech in 1999. The simplicity and low overhead of the protocol make it ideal for the emerging "machine-to-machine" (M2M) or "Internet of Things" (IoT) 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, your application needs to use a tcp://host:port URL (NSP Interfaces) or ssl://host:port URL (NSPS Interfaces). MQTT connections are treated in the same way as any other connections by the Universal Messaging realm. If the username is present, the Universal Messaging subject is `username@hostname`, otherwise the subject is `anonymous@hostname`.

***Figure 1: Connection List with an MQTT connection***

| protocol | user | host | connection | language |
|---|---|---|---|---|
| mqtt | anonymous | localhost | 127.0.0.1:61401 | MQTT 4 |
| nsp | krital | localhost | 127.0.0.1:61201 | Java 1.7.0_60 |

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 without a username/password to connect to `tcp://localhost:1883` will result in a Universal Messaging subject of `anonymous@localhost`.

## Publishing

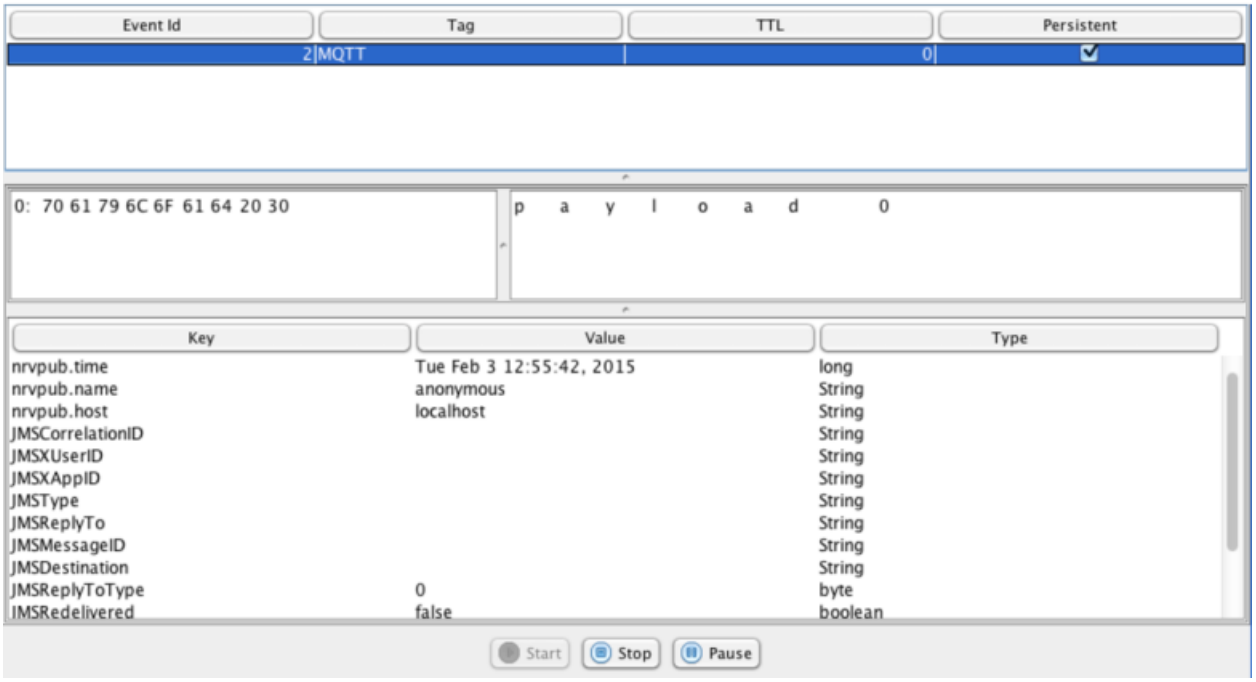
MQTT applications can publish events to channels. If the specified channels do not already exist in the Universal Messaging realm, they will be automatically created by the server as MIXED type with a JMS engine.

These channels are regular Universal Messaging channels but all events published from MQTT will be persisted to disk. While it is possible to create other channel types using the Administration API / Enterprise Manager, the mixed type with JMS engine is the recommended combination.

**Note:**
The set of permissible characters you can use to name a channel is the same as for standard Universal Messaging channels. This character set is described in the section *Creating Channels* in the Enterprise Manager section of the *Administration Guide*. Therefore it is possible that a channel name could be valid according to MQTT naming rules but invalid, and therefore would be rejected, according to Universal Messaging naming rules.

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***




All messages published from MQTT are flagged as JMS BytesMessage objects.

**Note:**
Publishing to queues via MQTT is not supported.

## Subscribing

MQTT applications can subscribe to channels. If the specified channels do not already exist in the Universal Messaging realm, they will be automatically created by the server as MIXED type with a JMS engine.

These channels are regular Universal Messaging channels with all messages being persistent, regardless of whether they are published by MQTT or Universal Messaging applications.

**Note:**
Subscribing to queues via MQTT is not supported.

## Quality of Service

Universal Messaging supports QOS levels 0 and 1 as 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 (*At most once delivery*): The Universal Messaging realm will deliver the message once with no confirmation
2. QOS 1 (*At least once delivery*): The Universal Messaging realm will deliver the message at least once, with confirmation required.

**Note:**
Universal Messaging does not support QOS level 2 (*Exactly once delivery*). Connections requesting QoS level 2 will be downgraded to QoS level 1 at connection time, as allowed by the MQTT specification.

## Will

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

# 10 AMQP


- Overview of AMQP ..... 210
- AMQP Guide ..... 216

# Overview of AMQP

## About AMQP

The Advanced Message Queuing Protocol (AMQP) is an open internet protocol for business messaging. The specification of the AMQP protocol is available at http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html. It defines a binary wire-level protocol that allows for the reliable exchange of business messages between two parties.

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

### Connecting

In order to connect to a Universal Messaging server using AMQP, your application needs to use an `amqp://host:port` URL (NSP Interfaces) or `amqps://host:port` URL (NSPS Interfaces). AMQP connections are treated in the same way as any other connections by the Universal Messaging realm. If the username is present, the Universal Messaging subject is `username@hostname`, otherwise the subject is `<AnonymousUser>@hostname` where *<AnonymousUser>* is configured under **Configuration > Protocol AMQP Config > AnonymousUser**.

Software AG recommends that you establish an AMQP JMS connection using the QPID Proton, QPID Legacy, or Swift MQ client libraries, and using a SASL plain over TLS connection.

***Figure 1: Connection List with an AMQP connection***

| protocol | user | host | connection | language |
|---|---|---|---|---|
| AMQP | anonymous | 127.0.0.1 | 127.0.0.1:64839 | AMQP 1.0.0 |
| nsp | dada | 127.0.0.1 | 127.0.0.1:64224 | Java 1.8.0_51 |

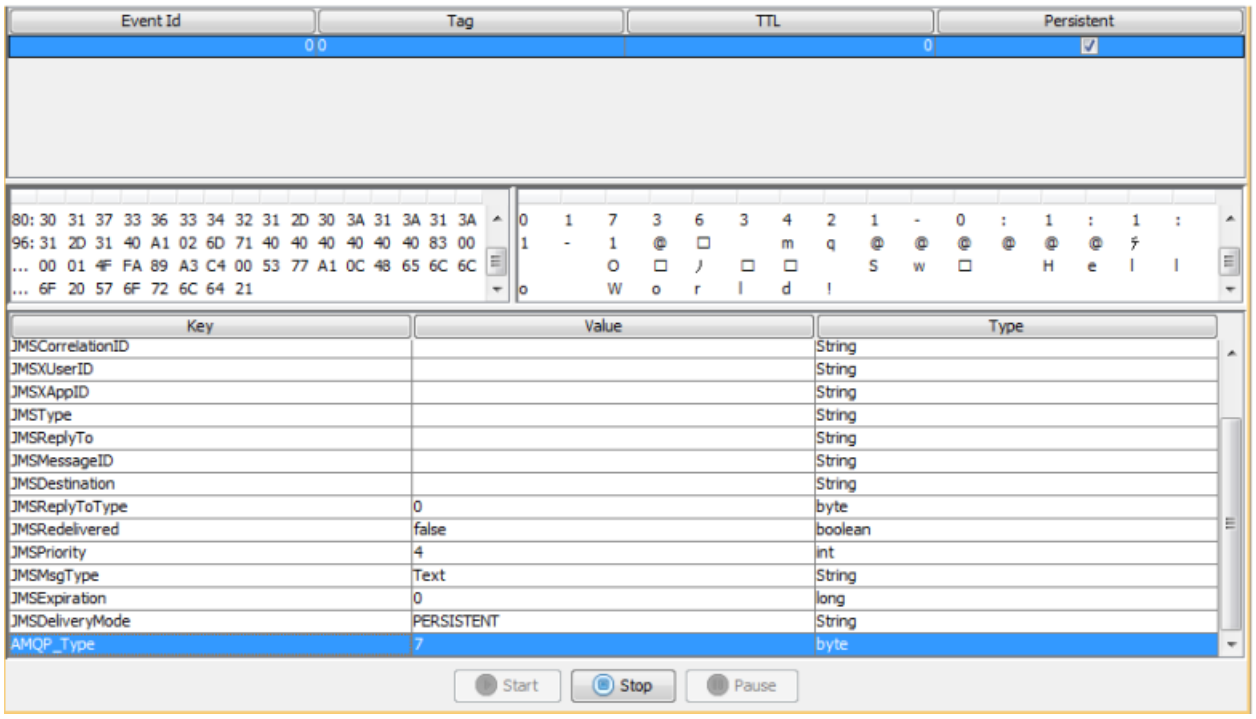
This way you can define realm and channel / queue ACLs as you would for any Universal Messaging connection. For example, using the one of the JMS Sample applications with AMQP without a username/password to connect to `amqp://localhost:10000` will result in a Universal Messaging subject of `<AnonymousUser>@localhost`.

### Publishing

AMQP applications can publish events to channels and queues, which should exist in advance.

Events published via AMQP 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 AMQP message***




## Subscribing

AMQP applications can subscribe to pre-existing channels and queues.

## Temporary Topics and Queues

The Universal Messaging server supports the creation of temporary topics and queues using an AMQP JMS session's createTemporaryTopic() and createTemporaryQueue() methods. Currently only the "delete-on-close" lifetime policy is supported; a Node dynamically created with this lifetime policy will be deleted at the point that the Link which caused its creation ceases to exist.

## Transactions over AMQP

Client applications can perform transactional work over AMQP. Universal Messaging implements the local transaction operations defined in the AMQP specification. For example, if an application communicates to the realm server using a JMS AMQP client library (e.g. Apache Qpid JMS client) it can take advantage of the local transaction functionalities defined in the JMS specification.

**Note:**
We do not currently support the Transactional Acquisition operation defined in the AMQP specification. However, this sets no limitations on using JMS transactions over AMQP.

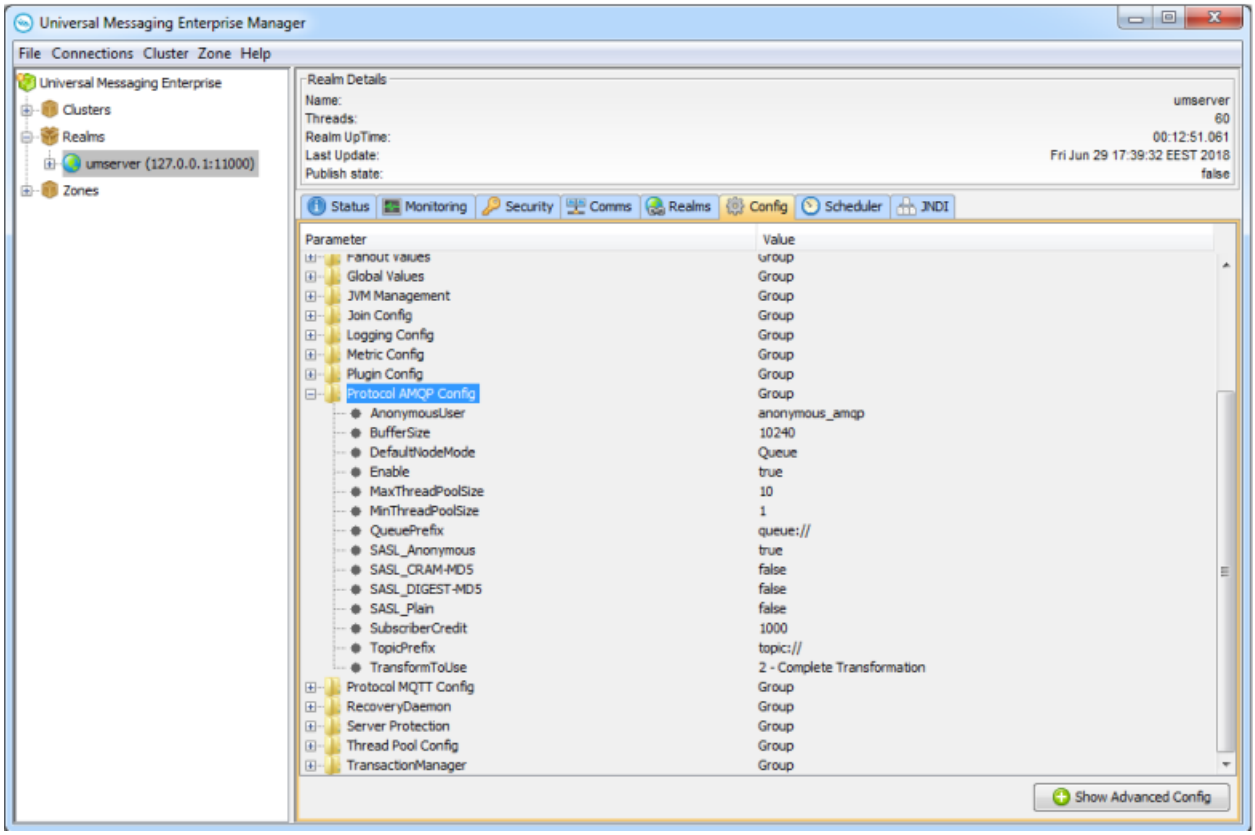
## JMS sample applications over AMQP

The JMS sample applications provided with the distribution of the Universal Messaging server can be configured to communicate over the AMQP protocol.

For more information, see the page " Using the AMQP Protocol" in the JMS code examples in the Developer Guide.

## AMQP Plugin Configuration

You can configure AMQP behavior using the configuration parameters that are shown under the **Config** tab in the Enterprise Manager.




**Note:**
The Universal Messaging realm properties for AMQP are applied on a per-connection basis, meaning that clients must re-connect to pick up a change in any realm property.

The configuration parameters are described in the following table:

| Parameter | Description |
|---|---|
| AnonymousUser | The user name which the server will use for anonymous users. |
| BufferSize | The size of the buffer in bytes that will be used to read/write on the AMQP connection. |
| DefaultNodeMode | The default type of node (topic or queue) if the server is not able to detect it. |
| Enable | If "true", the server will accept incoming AMQP connections. |

| Parameter | Description |
|---|---|
| MaxFrameSize | Maximum size of an AMQP frame in bytes. |
| QueuePrefix | The address prefix for specifying queue nodes as required by some clients. |
| SASL_Anonymous | Enable Anonymous SASL authentication. |
| SASL_CRAM-MD5 | Enable CRAM-MD5 SASL authentication. |
| SASL_DIGEST-MD5 | Enable DIGEST-MD5 SASL authentication. |
| SASL_Plain | Enable Plain SASL authentication. |
| SubscriberCredit | Sets initial credit for AMQP subscribers. |
| Timeout | Sets the network timeout in milliseconds. |
| TopicPrefix | The address prefix for specifying topic nodes as required by some clients. |
| TransformToUse | Select which transformation to use on AMQP messages. For more information on transformers see the section "AMQP Message Transformation" on page 214. |

**Note:**
The `idle time` of the AMQP connection for the Universal Messaging server can be configured by modifying the **Connection Config > KeepAlive** property.

**Note:**
We recommend that clients use SASL plain over TLS connections.

## AMQP Security Configuration

The Universal Messaging server supports the following security configurations when the AMQP protocol is used:

1. Plain AMQP
2. AMQP over SASL
3. AMQP over alternative TLS

Currently, AMQP over negotiated TLS is not supported.

### AMQP over SASL

The Universal Messaging server supports the following SASL mechanisms:

1. Anonymous

2. CRAM-MD5
3. DIGEST-MD5
4. Plain

You can enable and disable these mechanisms by using the AMQP configuration options in Enterprise Manager as explained in the section "AMQP Plugin Configuration" on page 212.

We recommend using the SASL Plain mechanism over a TLS Connection.

By default, the SASL anonymous authentication is used and the client can connect without providing a username or password. After the connection has been established, the ability to perform various operations is derived from the realm's ACL lists. If any other SASL type is used, then Basic Authentication also needs to be configured on the realm.

For more information on configuring basic authentication, see the page "Basic Authentication > Server-side Authentication" in the Java section of the Universal Messaging Developer Guide.

### AMQP over alternative TLS

The AMQP connection over alternative TLS can be established when the server has a running NSPS interface. In that case, the client should set the following system variables:

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

Then use the `amqps://<hostname>:<port>` URL to establish the connection.

## AMQP Message Transformation

The Universal Messaging server can be configured to perform transformations on any messages received and sent over the AMQP protocol. The transformations are controlled from the `TransformToUse` setting of the AMQP plugin configuration. The following settings are available:

| Setting | Description |
|---|---|
| 0 - No transformation | When this option is selected, no transformation is applied to the message, and it is passed as a byte array between the clients in the communication. This option is best used when only AMQP clients exchange messages. |
| 1 - Basic Transformation | With this option selected when a message is exchanged over the AMQP protocol, the following header entries are converted:<br>1. priority<br>2. delivery count / redelivery count<br>3. durable / delivery mode |

| Setting | Description |
|---|---|
| | This option is again best used when only AMQP clients exchange messages, but it also allows for server side filtering, based on the event's header and properties. |
| 2 - Complete Transformation | With this option selected when a message is exchanged over the AMQP protocol, the following header entries are converted:<br>1. priority<br>2. delivery count / redelivery count<br>3. durable / delivery mode<br>4. Time To Live (TTL)<br>5. First Acquirer / JMS_AMQP_FirstAcquirer<br>The footer of the AMQP message is converted to an event dictionary object and put under the "footer" key in the UM event dictionary and vice versa. If a UM event sent to an AMQP client has a "footer" dictionary object in the event dictionary, the content of that dictionary will be put as application properties of the AMQP message.<br>The delivery annotation, message annotations, application properties, and other standard properties of the AMQP message are also converted. For more information about mapping AMQP messages to Universal Messaging messages, see "Appendix B: AMQP Messages to Universal Messaging Message Mappings" on page 256.<br>**Important:**<br>If you are transporting JMS messages over AMQP, you should use this transform. It ensures that the JMS headers will be properly converted. |
| 3 - User Configurable | This option allows you to provide your own message converters. In order to use this option, do the following:<br>1. Place the MyTransformer.jar file into the *Universal Messaging_directory* `/plugins/ext` directory. Create the `/ext` directory if it does not exist.<br>2. In the Server_Common.conf configuration file, specify the following option:<br>`wrapper.java.additional.26=-DAMQP_TRANSFORM=foo.bar.MyTransformer`<br>*Alternatively*, provide a full class name as an `AMQP_TRANSFORM` JVM parameter to the server, for example:<br>`-DAMQP_TRANSFORM=foo.bar.MyTransformer` |

| Setting | Description |
|---|---|
| | Extend the abstract class by:<br>`com.pcbsys.nirvana.server.plugins.protocols .amqp.transformation.aTransformation`<br>If `AMQP_TRANSFORM` is not specified, this option will default to a null transform. |

# AMQP Guide

## Universal Messaging Support for AMQP

Advanced Message Queueing Protocol (AMQP) 1.0 is a binary networking protocol that standardizes messaging middleware communications as defined by standards body OASIS and *ISO/IEC 19464:2014*.

The following table describes the AMQP features that Universal Messaging version 10.7 supports:

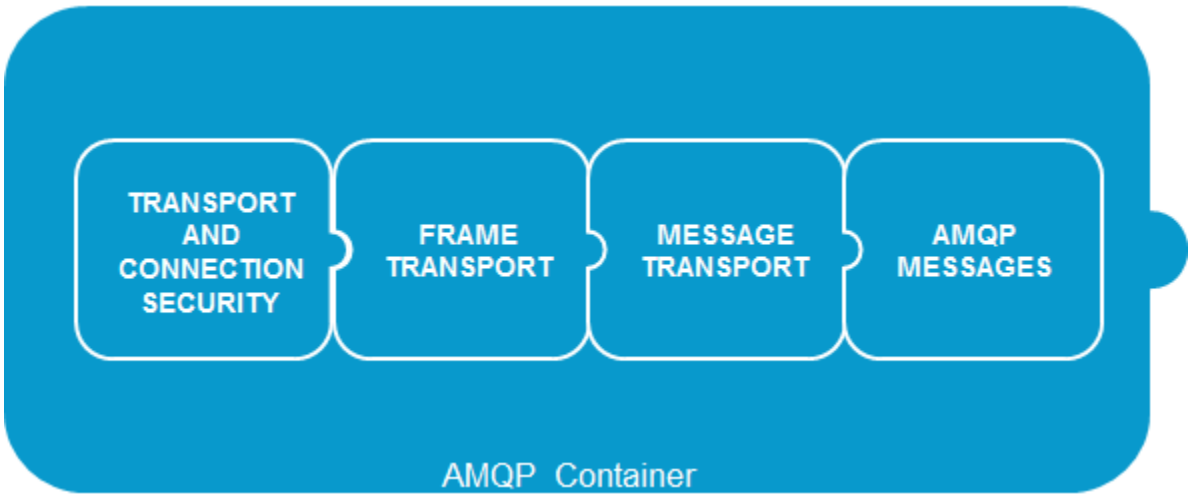
| AMQP Feature | Comments |
|---|---|
| TCP and TLS transport | Alternative support only, *not* negotiated TLS |
| Transactional publishing to topics and queues | Restricted only to local transactions because Universal Messaging does not support distributed transaction coordinators. |
| New Universal Messaging protocol layer threading module | AMQP performance and stability improvements |
| Plain and SASL authentication | SASL mechanisms are limited to the ones provided by the underlying JVM. |
| Non-transactional publishing to topics and queues | |
| Synchronous and asynchronous subscription to topics and queues | |
| Durable subscribers | Not all durable types. |
| Interoperability with Universal Messaging publishers and subscribers using any of the available client APIs | Requires complete transformation (needs protobuf support). |
| Temporary topics and queues | Only session lifetime policy. |
| Connection keep alives | |
| Credit-based flow control | Credit values < 100 cannot be reliably enforced |

| AMQP Feature | Comments |
|---|---|
| Interoperability with existing AMQP JMS clients (SwiftMQ, Apache QPID, Apache Legacy QPID) | |
| Event transformation API | Reduces marshalling on pure AMQP deployments, allows custom mapping definitions against the default AMQP message format. |

## AMQP Overview

The AMQP protocol offers a layered model consisting of transport and connection security, frame transfer and message transfer semantics without any assumptions on source / destination models or deployment topologies.

The result is a portable, secure, binary, symmetric message exchange between applications, regardless of whether a classic MOM broker, a cloud messaging infrastructure instance or a peer to peer message exchange service are involved.

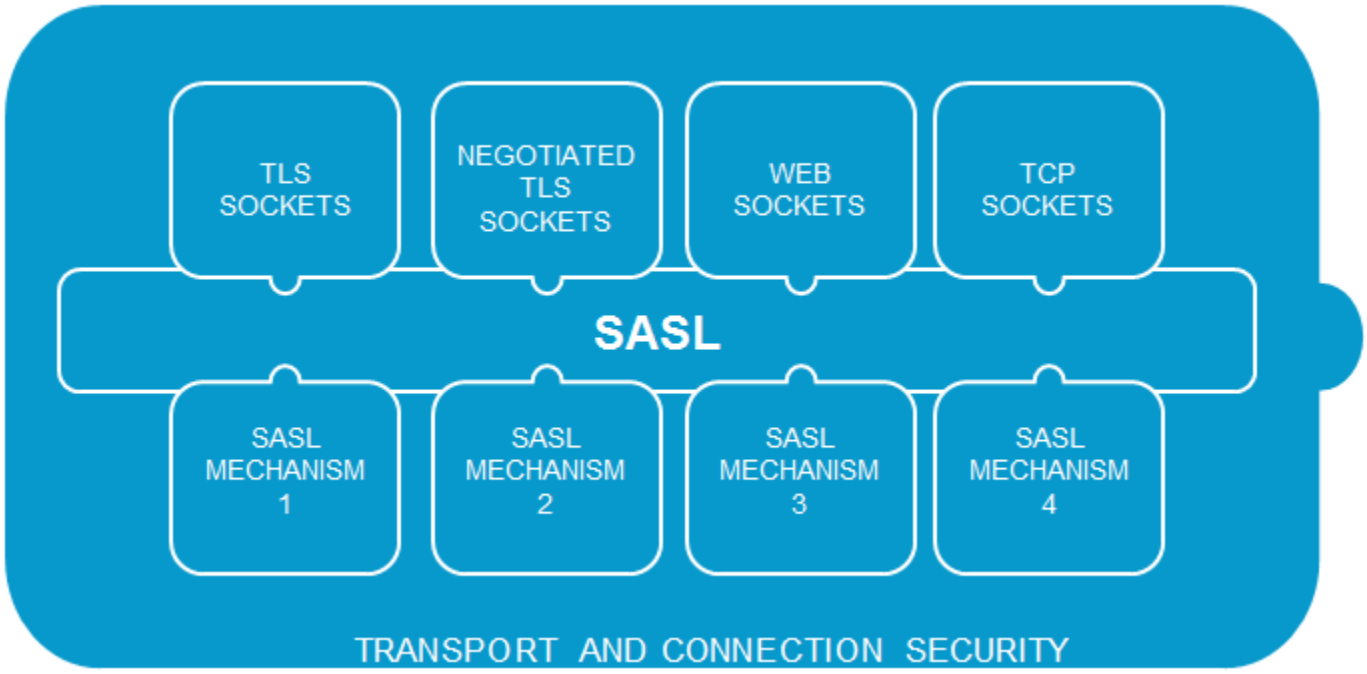



Universal Messaging supports AMQP 1.0 protocol only at the UM realm level. Users are expected to use an AMQP 1.0 compliant client library to connect.

**Note:**
Universal Messaging currently aims to cover JMS use cases. Although any compliant AMQP 1.0 client may be used to connect, certain header and message properties are mapped to JMS.

## Transport and Connection Security

AMQP defines the following with regard to Transport and Connection Security layers:




### TLS/SSL Socket

The node that accepts the connection uses a dedicated TLS port (Default `5671`). The node that initiates the connection needs to successfully establish a TLS session before any AMQP traffic is exchanged.

**Note:**
In Universal Messaging, the node that accepts the connection is the UM realm which needs to use an NSPS Interface. See the section *NSPS Ports* in the *Administration Guide* for related information.

### Negotiated TLS

The node that accepts the connections uses a single port for plain TCP & TLS sockets. The node that initiates the connection indicates the desire for TLS using a protocol flag, followed by AMQP traffic and upgrade of streams to TLS.

**Note:**
Universal Messaging does not currently support Negotiated TLS for AMQP 1.0.

### WebSockets

The node that accepts the connections uses a dedicated TCP or TLS port. The node that initiates the connection requests a WS upgrade, followed by AMQP traffic. For more information check the *AMQP over Websocket binding*, described on the OASIS web site https://www.oasis-open.org/org.

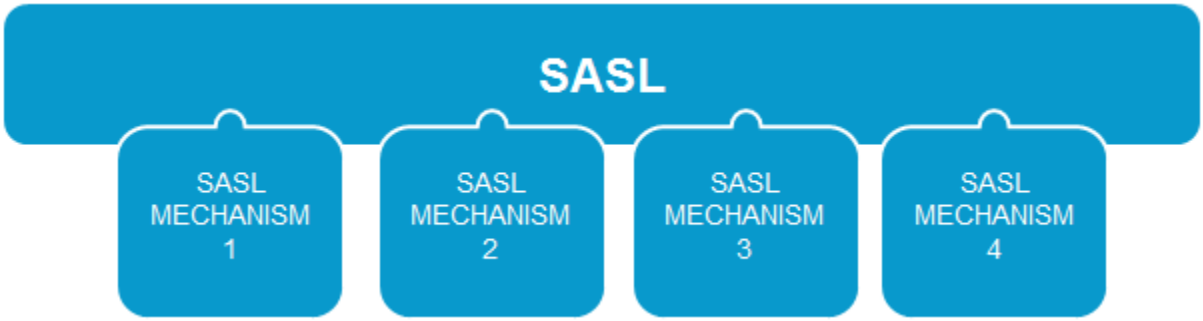
**Note:**
Universal Messaging does not currently support the AMQP over WS/S draft. There are plans for support in a future product release.

### TCP Sockets

The node that accepts the connection uses a dedicated TCP port (Default `5672`).

NOTE: In Universal Messaging, the node that accepts the connection is the UM realm which needs to use an NSP Interface (see "Native Communication Protocols" on page 301 for information about NSP).

### SASL




AMQP optionally allows support for SASL authentication. The node that accepts the connections can use a TCP or TLS port and negotiate a SASL mechanism using a protocol flag. The node that initiates the connection needs to negotiate a SASL mechanism.

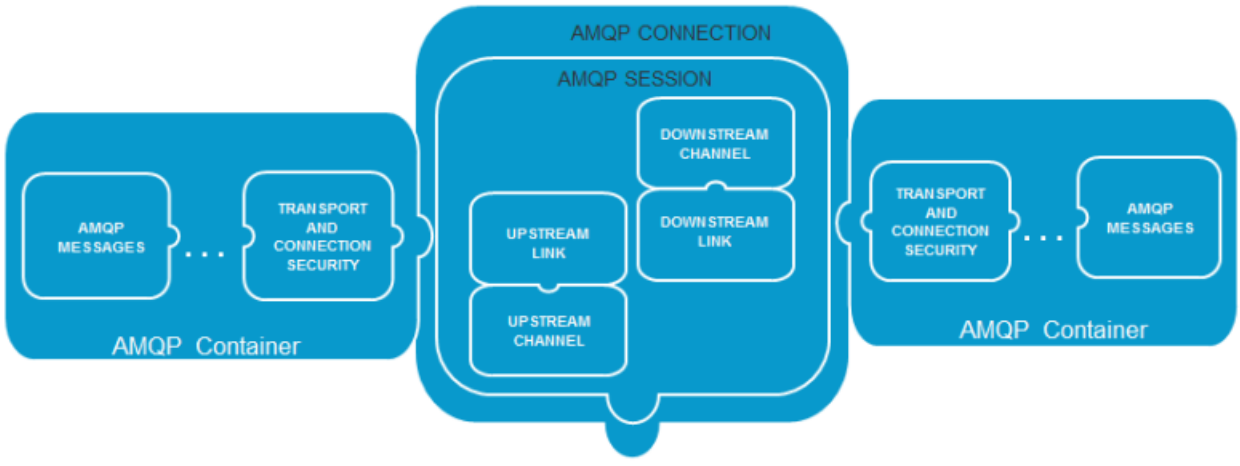
**Note:**

In Universal Messaging, the node that accepts the connections is the UM realm using an NSP or NSPS interface respectively. Universal Messaging SASL support is restricted by the underlying JVM SASL restrictions with regard to mechanisms supported.

See the section *TCP Interfaces, IP Multicast and Shared Memory* in the *Administration Guide* for information about interfaces.

**Note:**
In Universal Messaging, using SASL authentication only makes sense if authentication is enabled on the UM realm (see "Using SASL " on page 88 for related information).

## AMQP Frame Transport



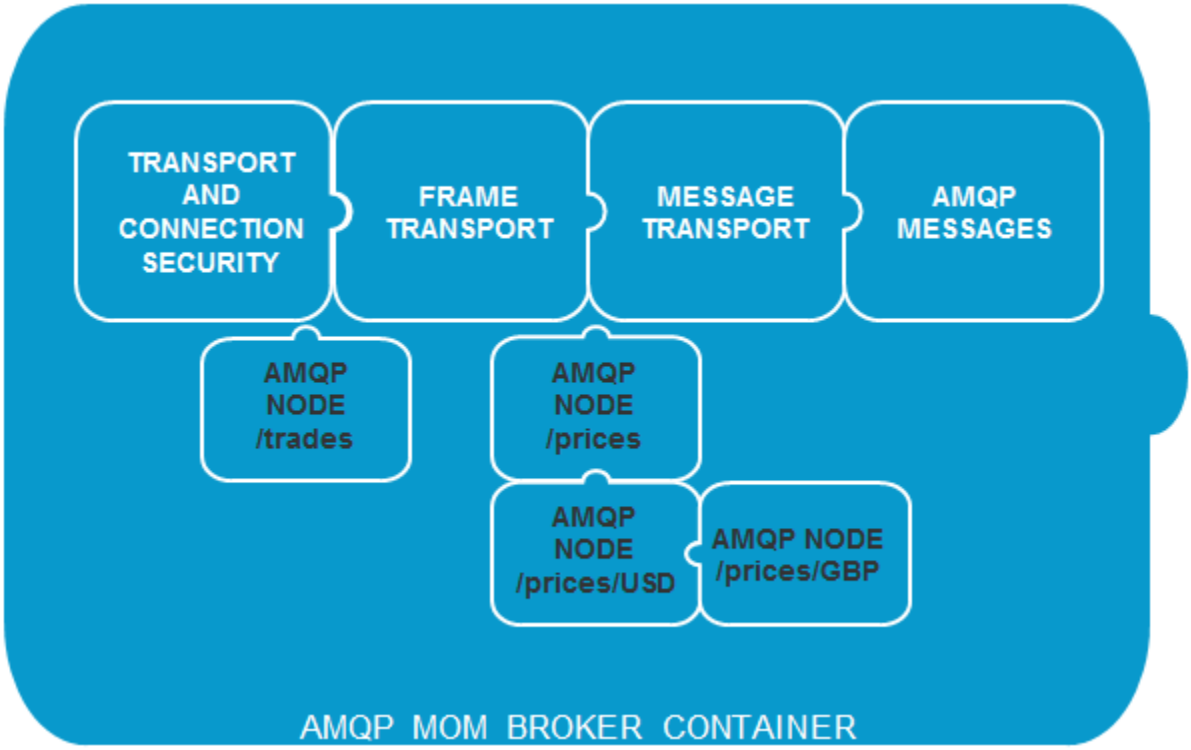

AMQP defines the following components with regard to frame transfers:

- Containers

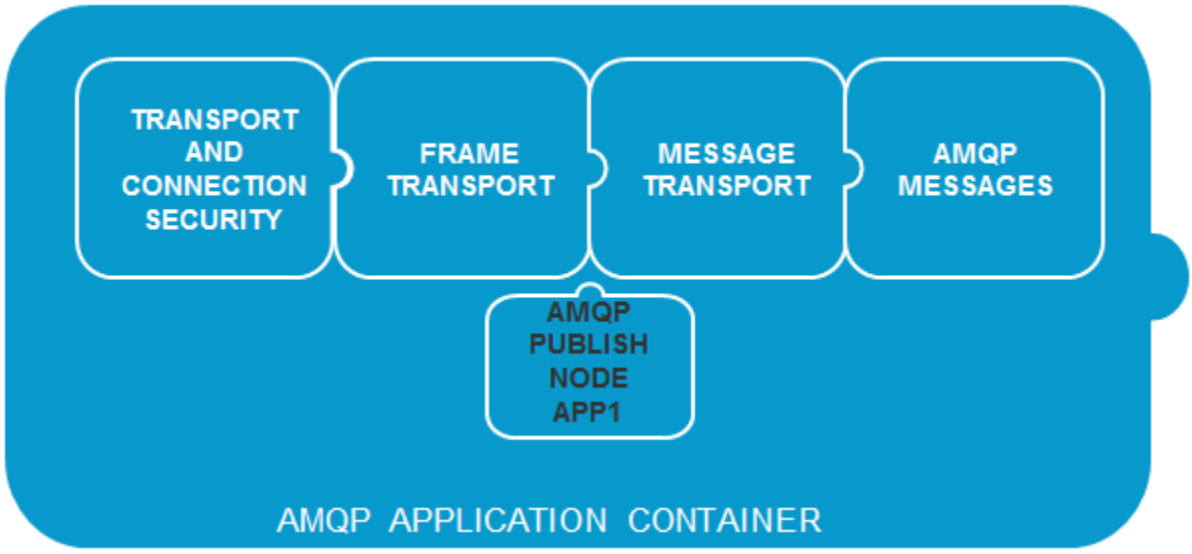
- Connections
- Channels
- Sessions
- Multiplexing
- Nodes
- Links
- Link Credit
- Link Recovery

## AMQP Containers

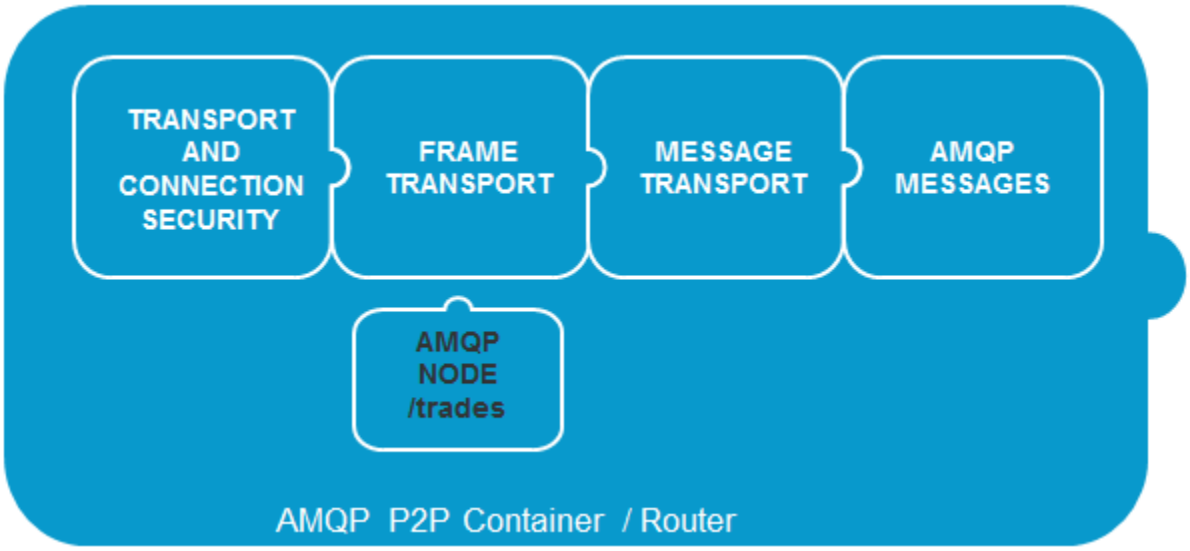
An AMQP container is an application that may send or receive messages. An example would be an AMQP message broker or an application using an AMQP client.



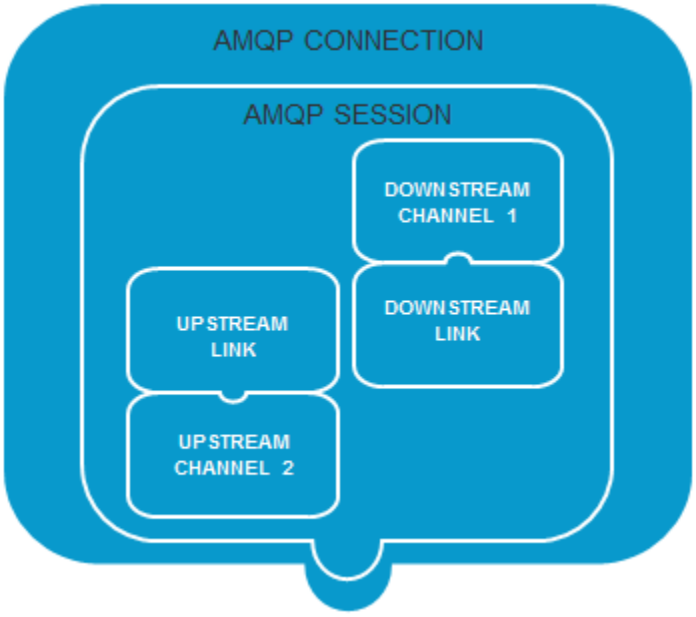

Another example would be an application using an AMQP client:




Or it could be an AMQP router, proxy or P2P container:



## AMQP Connections




AMQP connections offer a reliably ordered frame sequence restricted by a negotiable maximum frame size, with idle timeout semantics.

**Note:**
In Universal Messaging, the maximum frame size for AMQP connections must be less than or equal to the UM maximum buffer size.

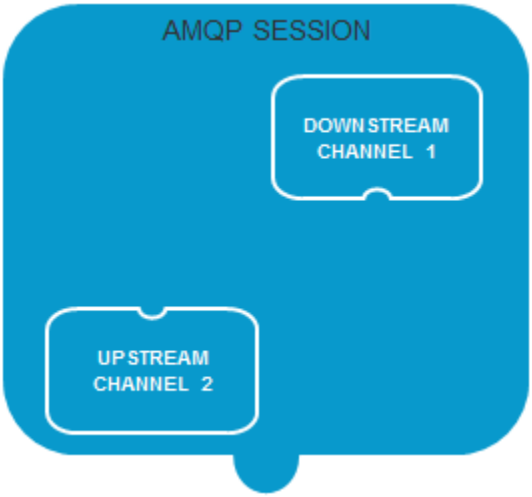
**Note:**
In Universal Messaging, the AMQP connection idle timeout semantics are mapped to native UM connection keep alive.

## AMQP Channels

AMQP frame sequences are transferred over a negotiated numbered of unidirectional channels.

**Note:**
In Universal Messaging, the maximum number of AMQP channels per connection is 2, excluding the reserved control channel 0.

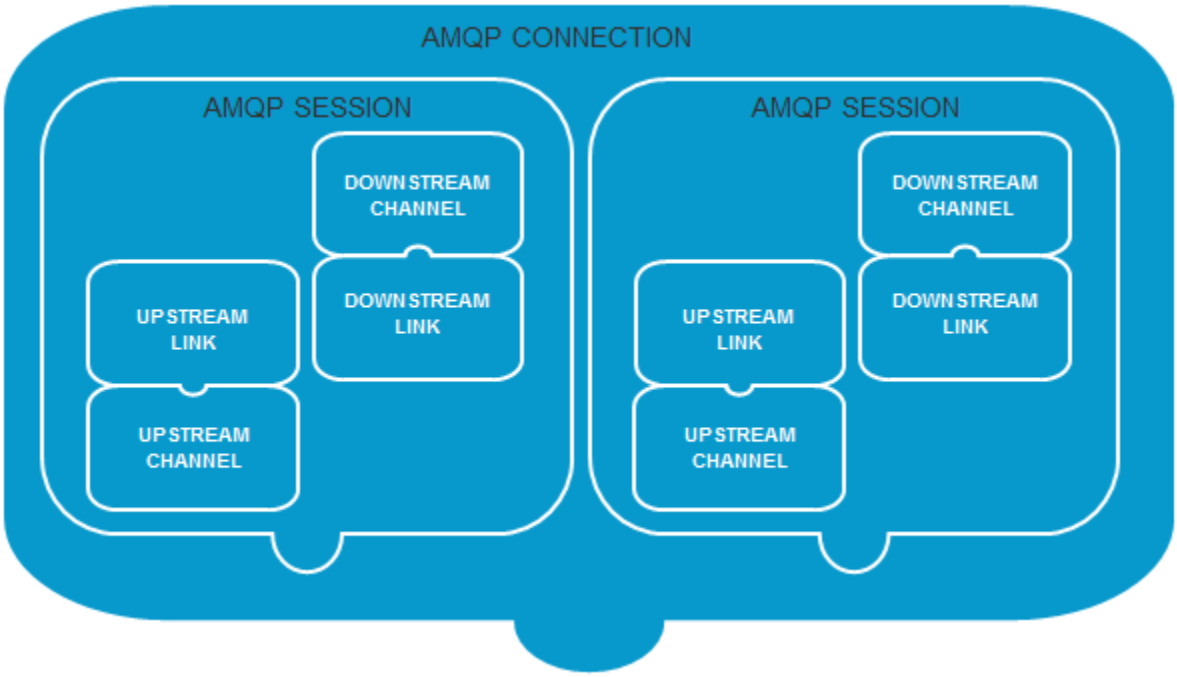
## AMQP Sessions




An AMQP session binds two unidirectional AMQP channels to define an ordered bi-directional byte stream.

**Note:**
In Universal Messaging, AMQP sessions are logically mapped to UM sessions. As multiplexing is currently unsupported, an AMQP connection can only have a single session when connecting to a UM realm.

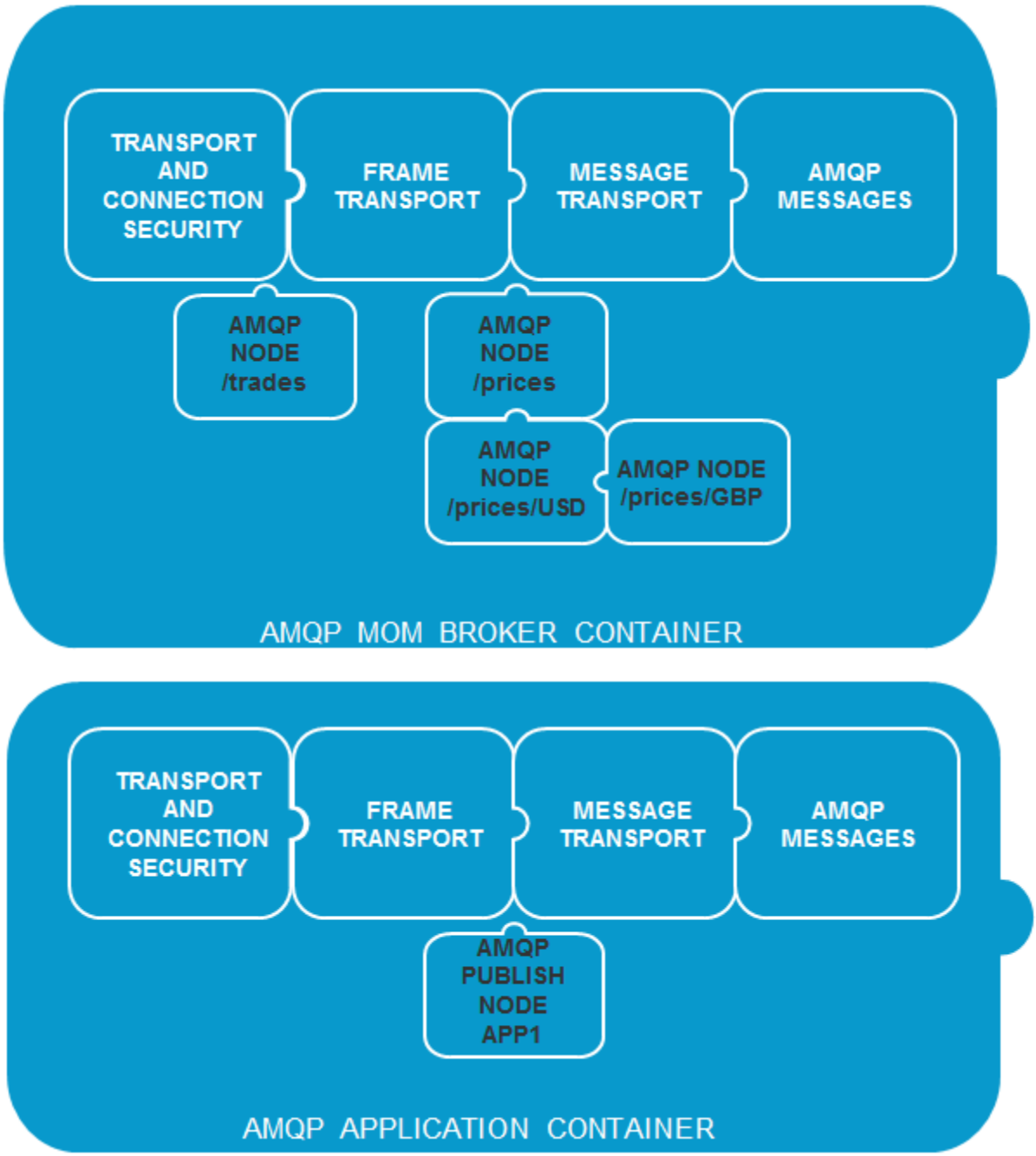
## AMQP Multiplexing



AMQP sessions can be optionally multiplexed over a single AMQP connection.

**Note:**
In Universal Messaging, session multiplexing is not supported at the moment. There are plans to add support in a future release by mapping to native UM multiplexing.

## AMQP Nodes

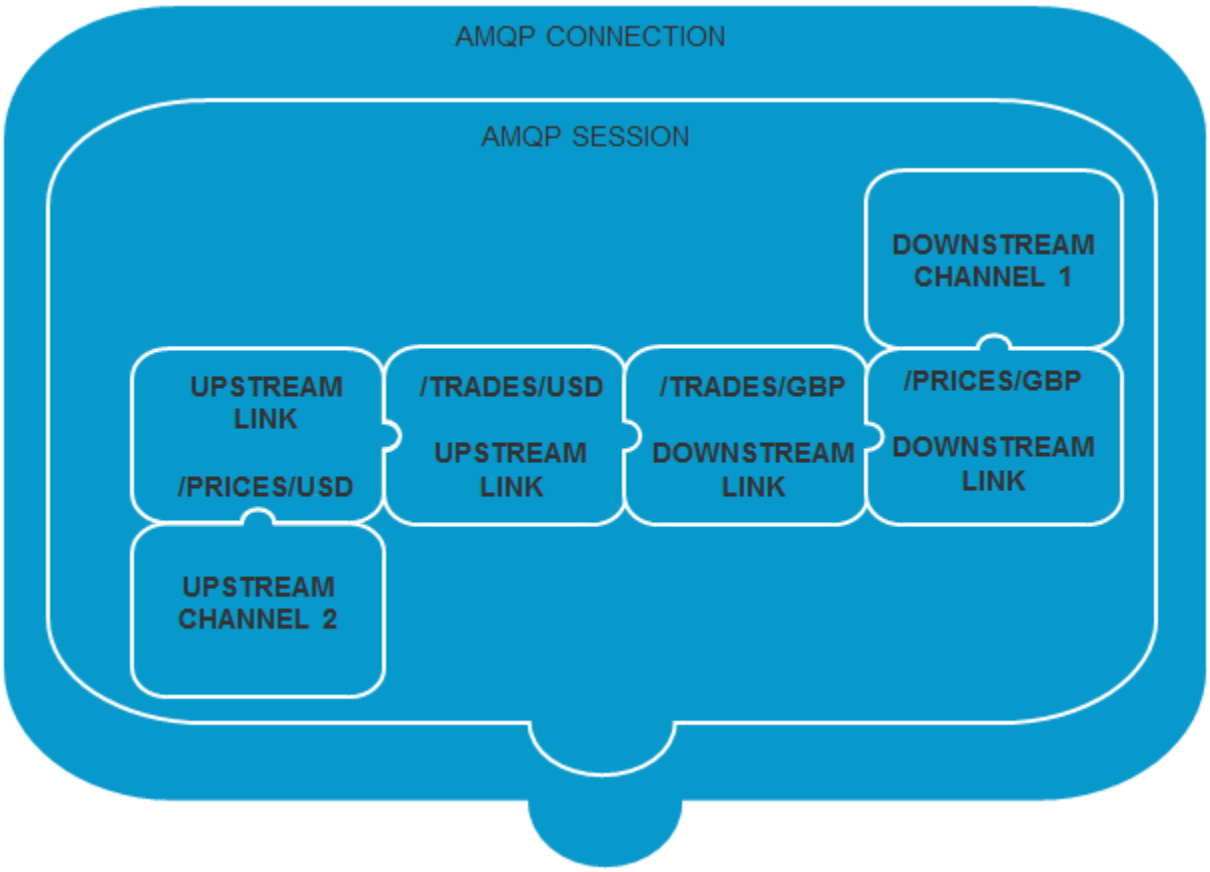



AMQP nodes are addressable logical endpoints with an AMQP container without any assumptions or requirements about the model (e.g. hierarchical, federated, graph etc).

**Note:**
In Universal Messaging, server side AMQP nodes can be either topics or queues but not data groups. As AMQP does not define any such semantics, a prefix model is widely adopted and supported by many vendors, including Software AG. Topic and queue nodes are expected to be prefixed with a configurable value agreed to imply node semantics (default: `topic://` and `queue://` respectively).

Furthermore, as Universal Messaging maps them to topic and queue stores respectively, if a prefix is not matched the default behavior will be honoured (see"AMQP Plugin Configuration" on page 212 for related information). Finally please note that some AMQP client implementations choose to define AMQP target capabilities to indicate the node type; Universal Messaging will try to use these as hints by looking for topic and queue strings in the capability name.

## AMQP Links




AMQP Links define a named path that defines a unidirectional flow between two AMQP nodes, the source and the target (e.g. a publisher application and a topic).

**Note:**
In a Universal Messaging realm, an incoming link to a topic or queue node is mapped to a publisher and an outgoing link to an application container is mapped to a subscriber.

**Note:**

An AMQP link from a sending or receiving AMQP application / container to a UM topic or queue store has a name equal to the absolute UM path (e.g. publishing messages to a topic node called `/trades/gbp` will flow over an incoming link to topic `/trades/gbp` named after the full path.

### AMQP Link Credit

AMQP links manage flow through link credits, allowing the receiving side of the link to control how fast messages flow through it. This allows a broker slowing down a publisher as well as a subscriber for controlling memory usage, pre-fetching etc.

**Note:**
In Universal Messaging, if an AMQP link is created without requesting a specific link credit, a configurable default value (1000) is assigned by the UM realm (see "AMQP Plugin Configuration" on page 212 for related information).

**Note:**
In Universal Messaging, all AMQP subscription flows are mapped to asynchronous consumers currently. As a result of that, link credit values < 100 will be honored at a best effort basis but cannot be guaranteed. We plan to improve on this in future versions of UM.

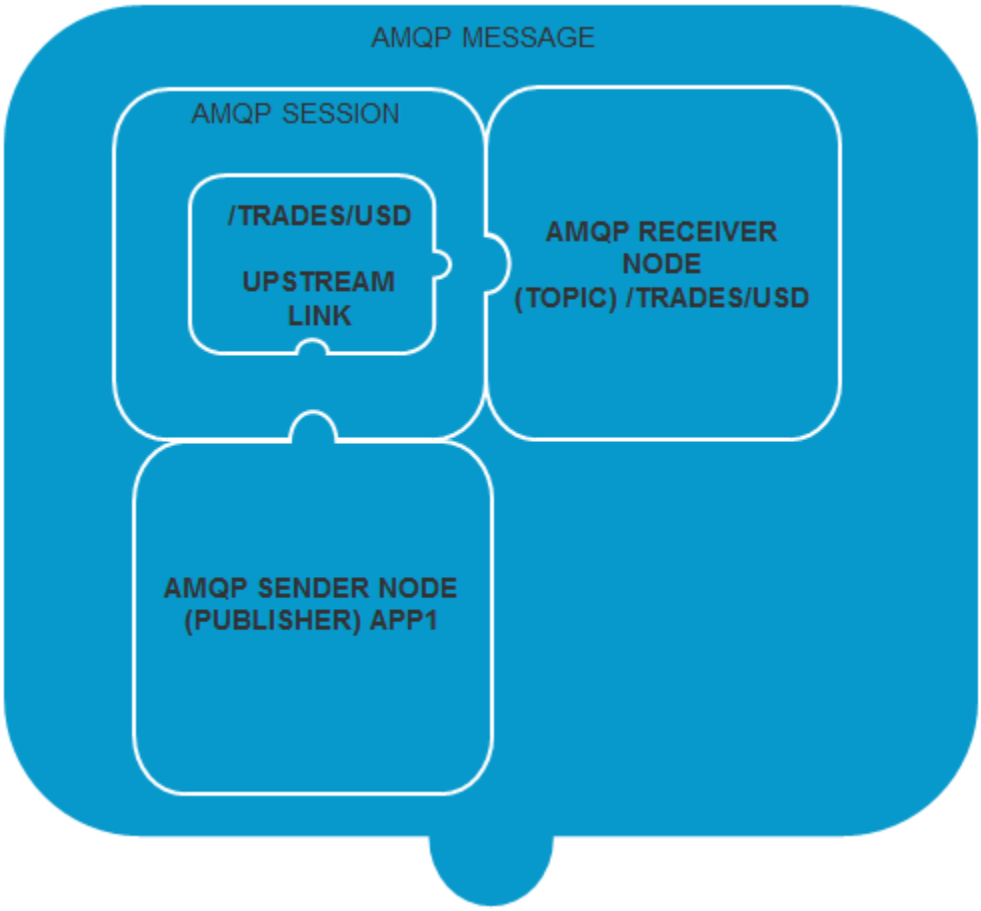
### AMQP Link Recovery

AMQP links can have an independent lifetime from AMQP sessions and AMQP connections and can be recovered.

**Note:**
In Universal Messaging, a pair of AMQP links are coupled with a single AMQP session and a single AMQP connection at the moment, while link recovery has not been implemented yet.

## Message Transport

AMQP message transfers (also called *deliveries*) can occur in either direction, with AMQP links created in pairs from the parent AMQP session.



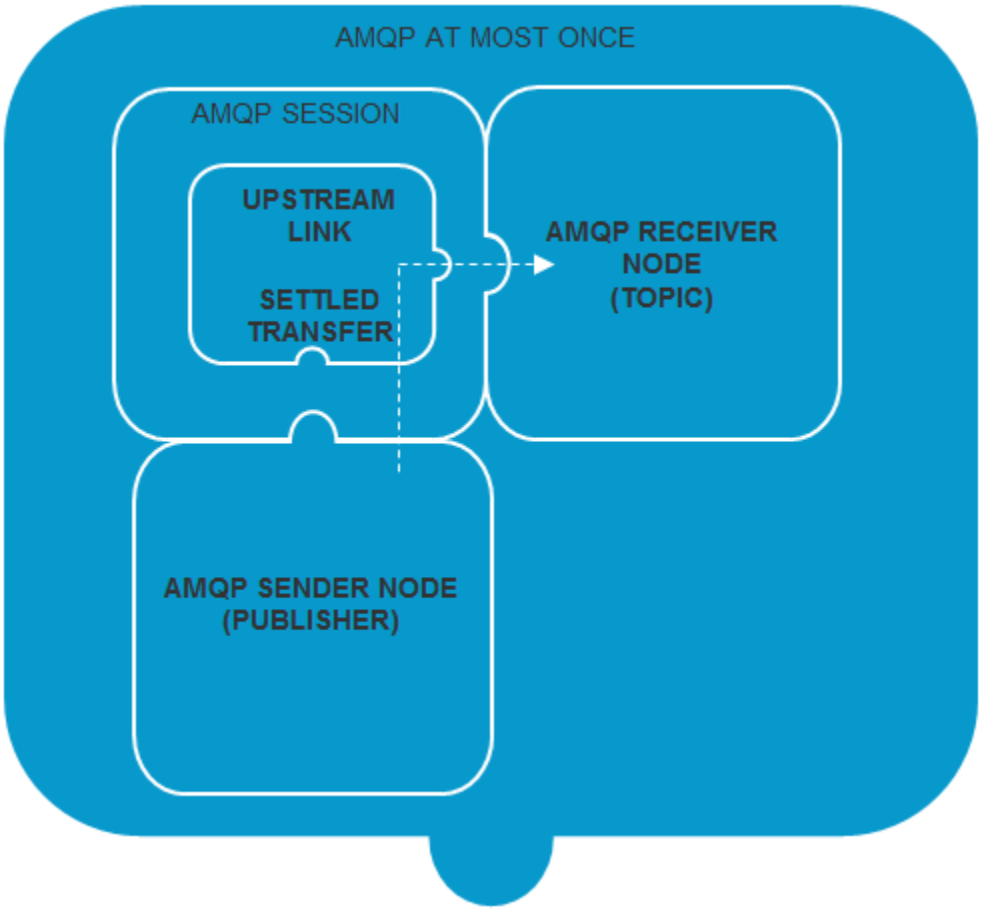

**Note:**
In Universal Messaging, delivery tags are mapped to a `<long>:<long>` naming scheme implying `<um store unique id>:<UM event EID>`.

AMQP defines the following approaches with regard to message transfers (AMQP deliveries):

- At Most Once
- At Least Once

## At Most Once

AMQP deliveries occurring via settled message transfers can be used for a "fire and forget" publishing / subscribing model, as they do not require an explicit acknowledgment.




## Publishing

AMQP publishing occurs when an AMQP application container (Sender) attaches a link to an AMQP node (Receiver) and initiates message deliveries. In the at-most-once model, message deliveries are sent pre-settled by the sender with no expectations of acknowledgment from the receiver.

**Note:**
In Universal Messaging, pre-settled message transfers are mapped to UM reliable publishing.

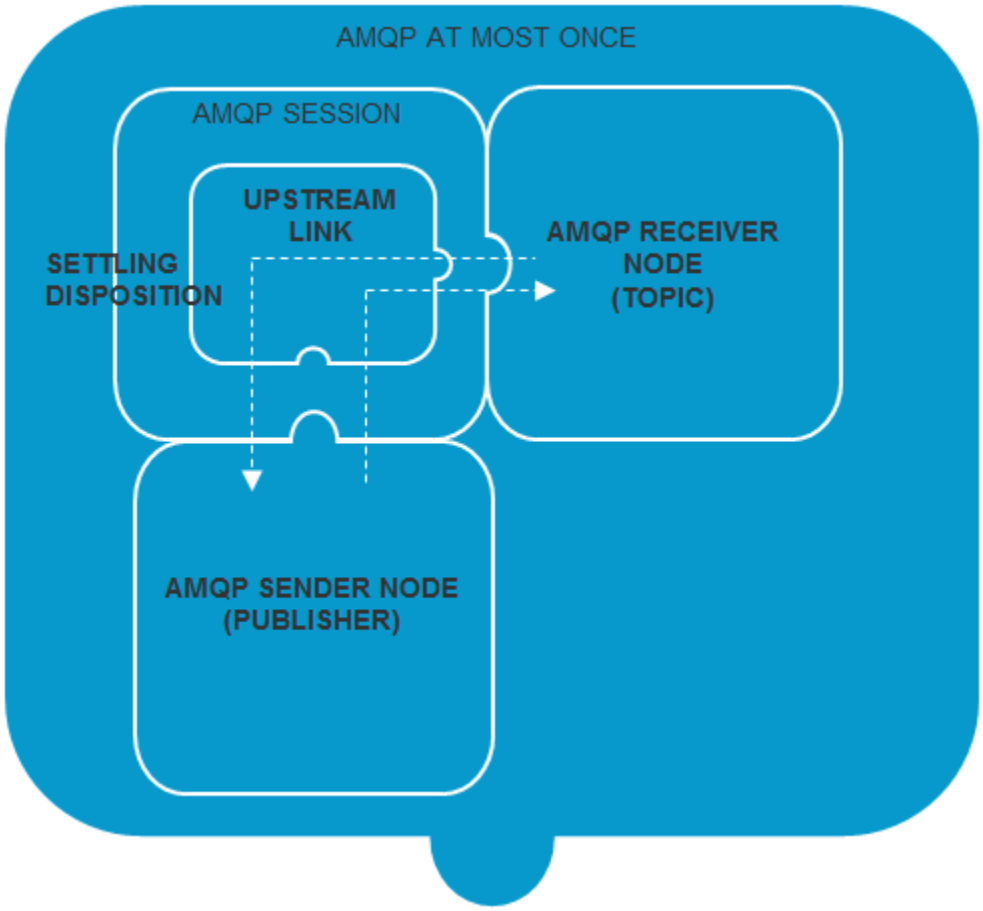
## Subscribing

AMQP subscribing occurs when an AMQP application container (Receiver) attaches a link to an AMQP node (Sender) causing a flow of deliveries from the sender node to the application node. In the at-most-once model, message deliveries are sent pre-settled by the sender with no expectations of acknowledgment from the receiver.

**Note:**
In Universal Messaging, pre-settled message transfers are mapped to UM reliable subscriptions (no Durability / ACKs).

## At Least Once

AMQP deliveries occurring via unsettled message transfers require an explicit settlement via acknowledgments, whether initiated by an application container or a broker container.




## Publishing

AMQP publishing occurs when an AMQP application container (Sender) attaches a link to an AMQP node (Receiver) and initiates message deliveries. In the at-least-once model, message deliveries are sent unsettled by the sender with an expectation of acknowledgment from the receiver.

**Note:**
In Universal Messaging, un-settled message transfers are mapped to UM single event TX publishing.

## Subscribing

AMQP subscribing occurs when an AMQP application container (Receiver) attaches a link to an AMQP node (Sender) causing a flow of deliveries from the sender node to the application node. In the at-least-once model, message deliveries are sent un-settled by the sender with an expectation

of acknowledgment from the receiver. The AMQP source durability configuration indicates whether the durable subscription state should be in-memory or persistent (CONFIGURATION or UNSETTLED_STATE respectively).

**Note:**
In Universal Messaging, unsettled message transfers are mapped to synchronous, individual ACK subscriptions.

# Delivery States

## Supported Delivery States

AMQP deliveries can have various delivery states, which can be either *intermediary* or *terminal* as described in the following table:

| AMQP Delivery States | UM 10.7 | Comments |
|---|---|---|
| received | ✗ | Currently link recovery is not supported in Universal Messaging. |
| accepted | ✓ | This state is reached when message transfers are accepted by the receiver node. |
| rejected | ✓ | This state is reached when message transfers are rejected by the receiver node. |
| released | ✓ | This state is reached when message transfers are abandoned by the receiver node, signalling a redelivery request. |
| modified | ✓ | This state is reached when message transfers are abandoned by the receiver node, signalling a redelivery request after modification. |

## Intermediate Delivery States

## Intermediate Delivery States

### Received

AMQP defines a single intermediary delivery state for message transfers that are in progress called *received* which is used for AMQP link recovery negotiation.

**Note:**
In Universal Messaging, Link recovery is currently not supported, so no explicit use of the received state is performed.

## Terminal Delivery States

AMQP defines four terminal delivery states for message transfers that can occur after completion of the transfer: *accepted, rejected, released* and *modified*.

## Accepted

The accepted terminal delivery state indicates that the destination node has accepted the transfer.

**Accepted: Subscription Mapping**

The following table illustrates how the accepted delivery state is mapped to Universal Messaging server (Sender) communicating with an AMQP subscriber (Receiver):

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| null | null | N/A | N/A | N/A | ✓ |

Description: When UM receives a disposition frame without a local and remote state, it responds by accepting the delivery, disposing using the source's default outcome and advancing the link.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | true | N/A | N/A | ✓ |

Description: When UM receives a disposition frame with a remote state of accepted but already settled, no further action is taken.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | false | ANY | ANY | ✓ |

Description: When UM receives a link attachment the link address is examined and the destination node verified. Following that an asynchronous UM native sub is queued. Finally when the UM server has completed the subscription, the delivery is accepted and settled. This is equivalent to a UM regular subscribe.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | false | ReceiverSettleMode.FIRST | NONE | ✓ |

Description: When UM receives an unsettled disposition frame with a remote state of accepted, durability NONE and ReceiverSettleMode.FIRST, it accepts and settles it with no further action. This is equivalent to a UM reliable subscriber (no durability, no ACKs).

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | false | ReceiverSettleMode.FIRST | CONFIGURATION OR UNSETTLED_STATE | ✓ |

Description: When UM receives an unsettled disposition frame with a remote state of accepted, durability CONFIGURATION (NON PERSISTENT) or UNSETTLED_STATE (PERSISTENT) and ReceiverSettleMode.FIRST, it first extracts the delivery tag which is expected in the form <um store uniqueid>:<um event EID>, followed by extracting the durable name from the link. Following that an asynchronous UM native ACK (SYNC=TRUE, INDIVIDUAL=TRUE) is queued with the mapped fields. Finally when the UM server has completed the ACK, the delivery is accepted and settled. This is equivalent to a UM durable subscriber ACK.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | false | ReceiverSettleMode.SECOND | NONE | ✓ |

Description: When UM receives an unsettled disposition frame with a remote state of accepted, durability NONE and ReceiverSettleMode.FIRST, it accepts and settles it with no further action. This is equivalent to a UM reliable subscriber (no durability, no ACKs).

**Note:**
Normally in this mode, UM should use 2 phase commit semantics to communicate the acceptance before setting. As UM does not currently support 2 phase commit, this currently behaves exactly like ReceiverSettleMode.FIRST.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | accepted | false | ReceiverSettleMode.SECOND | CONFIGURATION OR UNSETTLED_STATE | ✓ |

Description: When UM receives an unsettled disposition frame with a remote state of accepted, durability CONFIGURATION or UNSETTLED_STATE and ReceiverSettleMode.FIRST, it first extracts the delivery tag which is expected in the form <um store uniqueid>:<um event EID>, followed by extracting the durable name from the link. Following that an asynchronous UM native ACK (SYNC=TRUE, INDIVIDUAL=TRUE) is queued with the mapped fields. Finally when the

UM server has completed the ACK, the delivery is accepted and settled. This is equivalent to a UM durable subscriber ACK.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| N/A | transactional | N/A | N/A | N/A | ✓ |

Description: When UM receives an unsettled disposition frame with a remote state of transactional, it first extracts the delivery tag which is expected in the form <um store uniqueid>:<um event EID>, followed by extracting the durable name from the link. Following that it updates internal data structures in the sender session context. This is equivalent to a UM transactional durable subscriber ACK.

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|---|
| null | null | N/A | N/A | N/A | ✓ |

When UM receives a disposition frame without a local and remote state, it responds by accepting the delivery, disposing using the source's default outcome and advancing the link.

**Accepted: Publishing Mapping**

The following table illustrates how the accepted delivery state is mapped to a Universal Messaging server (Receiver) communicating with an AMQP publisher (Sender):

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| true | null | N/A | N/A | ✗ |

Description: When UM receives a partial delivery, it rejects it with an amqp:not-implemented error and settles, as multi frame transfers are not currently supported in UM. We have plans to add support for this in a future release of the product.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| false | N/A | null | N/A | ✓ |

Description: When UM receives a non-partial delivery it tries to examine and transform its contents using the configured UM transformer. If the result of this operation is null, the delivery is rejected with an amqp:precondition-failed error and settled. This is unexpected outcome which indicates that an unexpected error occurred during transformation.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| false | NOT transactional | NOT null | SETTLED | ✓ |

Description: When UM receives a non-partial and non-transactional delivery it tries to examine and transform its contents using the configured UM transformer. Following that, an asynchronous UM native publish is queued. Finally, when the UM server has processed the native publish and Sender Settle mode is SETTLED, no further action is taken. This is directly equivalent to a UM reliable publisher.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| false | NOT transactional | NOT null | UNSETTLED | ✓ |

Description: When UM receives a non-partial and non-transactional delivery it tries to examine and transform its contents using the configured UM transformer. Following that, an asynchronous UM native publish is queued. Finally, when the UM server has processed the native publish and Sender Settle mode is UNSETTLED, it will accept it and advance the link. This is equivalent to a UM reliable publisher with a response from the server but no transactions.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| false | NOT transactional | NOT null | MIXED | ✓ |

Description: When UM receives a non-partial and non-transactional delivery it tries to examine and transform its contents using the configured UM transformer. Following that, an asynchronous UM native publish is queued. Finally, when the UM server has processed the native publish and Sender Settle mode is MIXED, it will accept it and advance the link ONLY if it is not already settled. This is equivalent to a UM reliable publisher with a response from the server for some publisher controlled transfers and no transactions involved.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|---|
| false | transactional | NOT null | N/A | ✓ |

Description: When UM receives a non-partial delivery transactional delivery, it tries to examine and transform its contents using the configured UM transformer. Following that, an asynchronous UM TX native publish request is queued. Finally, when the UM server has processed the native TX publish request, it will accept it with a transaction state set to the TX ID and advance the link ONLY if it is not already settled. If it is already settled, no further action will be taken. This is equivalent to a UM TX publisher .

## Rejected

The *rejected* terminal delivery state indicates that the destination node has rejected the transfer.

### Subscription Mapping

The following table illustrates how the rejected delivery state is mapped to Universal Messaging server (Sender) communicating with an AMQP subscriber (Receiver):

| Local Disposition State | Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|---|
| N/A | accepted | false | ANY | ANY | amqp:unauthorized-access (fatal)<br>amqp:internal-error (fatal)<br>amqp:not-found (fatal)<br>amqp:invalid-field (fatal)<br>amqp:illegal-state (fatal) | ✓ |

As discussed in the accepted state mappings, an AMQP subscriber with these settings is mapped to an asynchronous UM native subscription. When the UM server has completed processing the subscription request and an error has occurred the delivery is rejected and settled as follows:

- amqp:unauthorized-access : UM ACLs do not allow subscriptions for this subject. This is a fatal error so the link is closed.
- amqp:internal-error: An unexpected UM internal error has occurred. This is a fatal error so the link is closed.
- amqp:not-found: The UM server is unable to find the destination node. This is fatal error so the link is closed.
- amqp:invalid-field: The message selector specified under key jms-selector raised a selector parse error.
- amqp:illegal-state: The UM server rejected the subscription because the connection is already subscribed.

| Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|
| accepted | false | ReceiverSettleMode.FIRST<br>OR<br>ReceiverSettleMode.SECOND | CONFIGURATION<br>OR<br>UNSETTLED_STATE | amqp:unauthorized-access (fatal)<br>amqp:not-found (fatal) | ✓ |

In addition to the above rejections applying for all subscriptions and accepted state mappings, an AMQP subscriber with these settings is mapped to an asynchronous UM native ACK (SYNC=TRUE, INDIVIDUAL=TRUE). When the UM server has completed processing the ACK, and an error has occurred the delivery is rejected and settled as follows:

- amqp:unauthorized-access : UM ACLs do not allow durable subscribing for this subject. This is a fatal error so the link is closed.
- amqp:not-found: The UM server is unable to find the durable name on the destination node. This is fatal error so the link is closed.

**Publishing Mapping**

The following table illustrates how the rejected delivery state is mapped to a Universal Messaging server (Receiver) communicating with an AMQP publisher (Sender):

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|
| true | null | N/A | N/A | amqp:not-implemented (fatal) | ✓ |

Description: When UM receives a partial delivery, it rejects it and settles it, as multi frame transfers are not currently supported in UM. We have plans to add support for this in a future release of the product.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|
| false | NOT transactional | NOT null | UNSETTLED OR SETTLED | amqp:unauthorized-access (fatal)<br>amqp:resource-limit-exceeded (transient)<br>amqp:internal-error (fatal) | ✓ |

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|
| | | | | amqp:not-found (fatal) | |

Description: As discussed in the accepted state mappings, an AMQP publisher with these settings is mapped to an asynchronous UM native publish getting queued. When the UM server has completed processing the native publish, and an error has occurred the delivery is rejected and settled as follows:

- amqp:unauthorized-access : UM ACLs do not allow publishing for this subject. This is a fatal error so the link is closed.
- amqp:resource-limit-exceeded: UM store capacity has been reached. This is a transient error so the link is advanced.
- amqp:internal-error: An unexpected UM internal error has occurred. This is a fatal error so the link is closed.
- amqp:not-found: The UM server is unable to find the destination node. This is fatal error so the link is closed.

| Partial Delivery | Remote Disposition State | UM Transformer Result | Sender Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|---|
| false | transactional | NOT null | N/A | amqp:unauthorized-access (fatal)<br>amqp:resource-limit-exceeded (transient)<br>amqp:internal-error (fatal)<br>amqp:not-found (fatal) | |

Description: As discussed in the accepted state mappings, a transactional AMQP publisher with these settings is mapped to an asynchronous UM TX native publish request getting queued. When the UM server has completed processing the native TX publish request, and an error has occurred, the delivery is rejected and settled as follows:

- amqp:unauthorized-access : UM ACLs do not allow publishing for this subject. This is a fatal error so the link is closed.
- amqp:resource-limit-exceeded: UM store capacity has been reached. This is a transient error so the link is advanced.
- amqp:internal-error: An unexpected UM internal error has occurred. This is a fatal error so the link is closed.

- amqp:not-found: The UM server is unable to find the destination node. This is fatal error so the link is closed.

**Released**

The *released* terminal delivery state indicates that the destination node no longer knows anything about the transfer, expecting redelivery.

NOTE: In Universal Messaging Link no explicit use of the released state is performed as it is internally handled by the Apache Qpid™ Proton-J protocol engine. See https://qpid.apache.org/proton/index.html for related information.

**Modified**

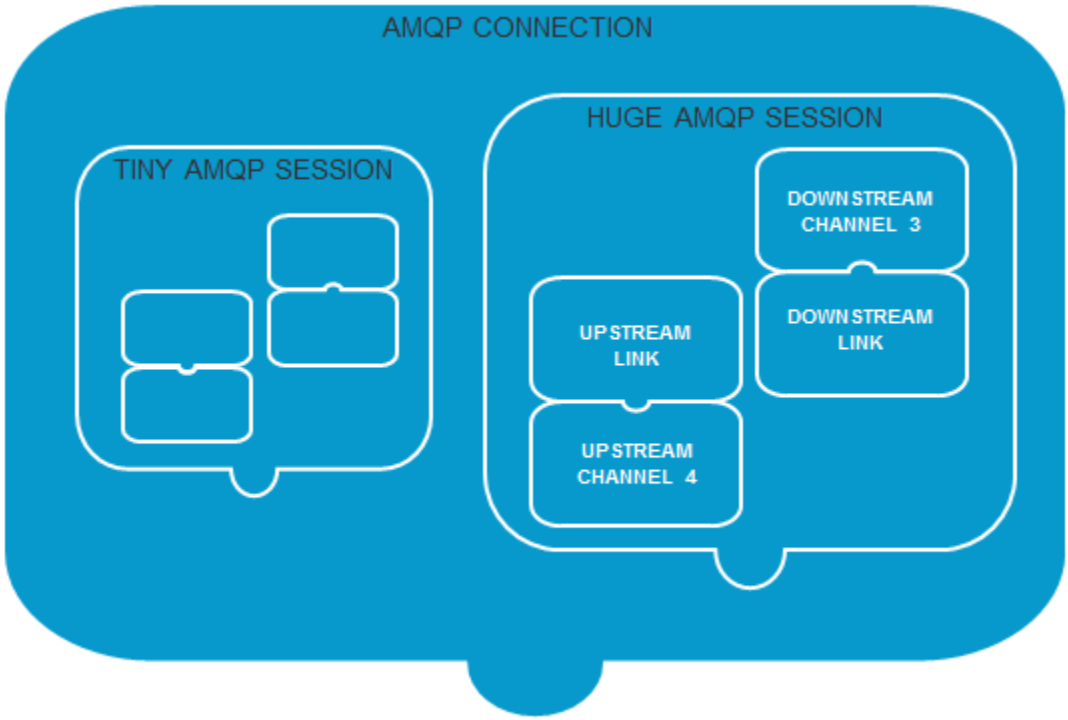
The *modified* terminal delivery state indicates that the destination node no longer knows anything about the transfer, expecting redelivery but only after modification by the sender.

NOTE: In Universal Messaging Link no explicit use of the modified state is performed as it is internally handled by the Apache Qpid™ Proton-J protocol engine. See https://qpid.apache.org/proton/index.html for related information.

# Flow Control

## Session Flow Control (Platform Backpressure)

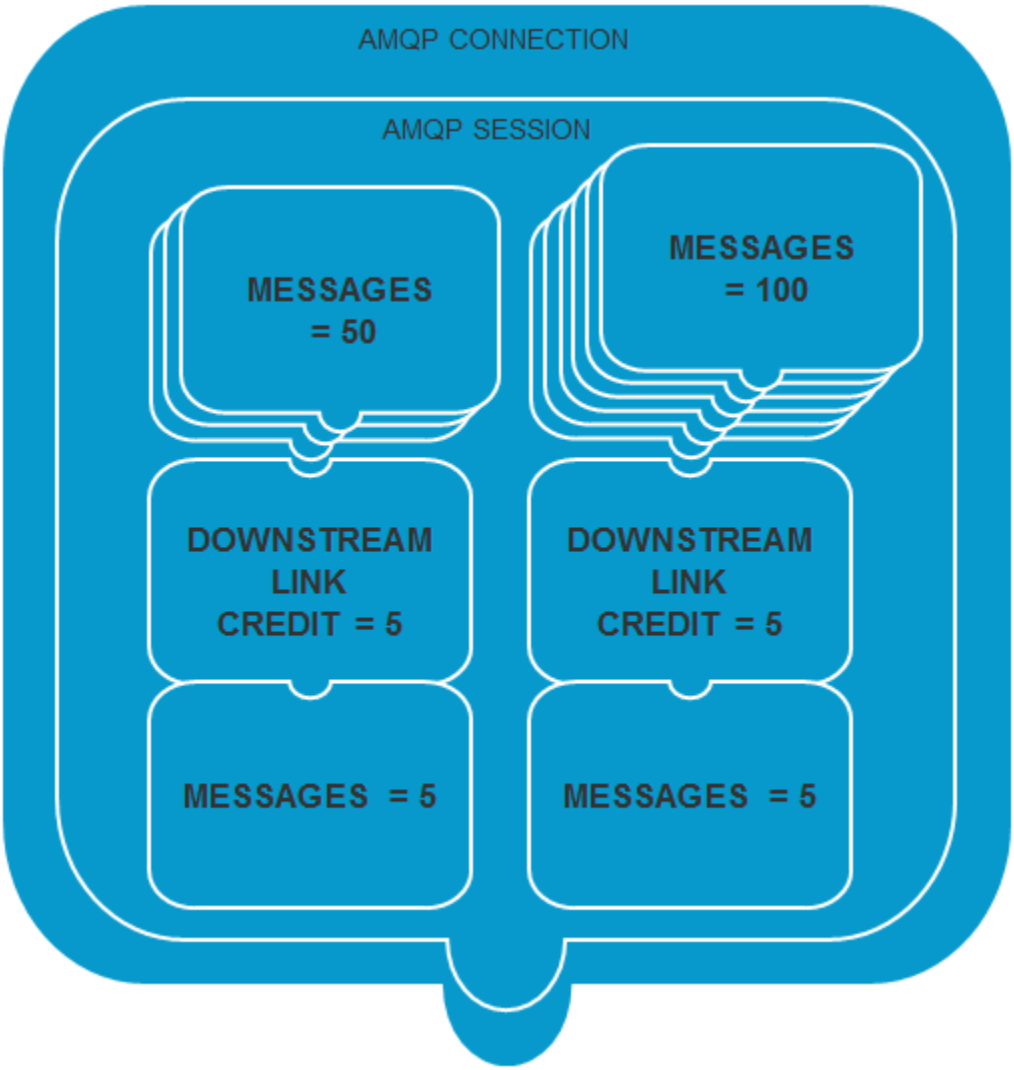
AMQP sessions have an incoming and outgoing transfer window offering session flow control semantics expressed as frame count. The window sizes are managed by participating incoming / outgoing transfers with 0 indicating a suspension offering back pressure protection to the underlying platform, shielding it from higher than expected bursts of messages.




NOTE: In Universal Messaging, session windows are not mapped directly to a UM structure but are manageable via the embedded Proton-J protocol engine (default: 16384). As we currently do not support session multiplexing, this configuration is not relevant for Universal Messaging servers who continue to offer platform backpressure via the native UM protocol.

### Link Flow Control (Application Backpressure)

AMQP Links have a link credit value indicating the maximum number of deliveries the receiving end is prepared to accept. The link credit value is only modifiable by the receiving end of the link with 0 indicating a suspension offering back pressure protection to the application logic, shielding it from receiving more messages than it can handle.




NOTE: In Universal messaging, link credit is decremented by the participating transfers, with a configurable default value set if unspecified (refer to the section "AMQP Plugin Configuration" on page 212 for details). Currently, Universal Messaging realms are unable to reliably honor link credit values of less than 100. We have plans to improve this in a future release of the product but until then, we recommend the use of store capacity to achieve such strict control.

## AMQP Messages

### AMQP Types

AMQP defines a set of commonly used primitive types aimed towards a cross-platform interoperable data representation

These primitive types are divided into the following categories:

- Primitive

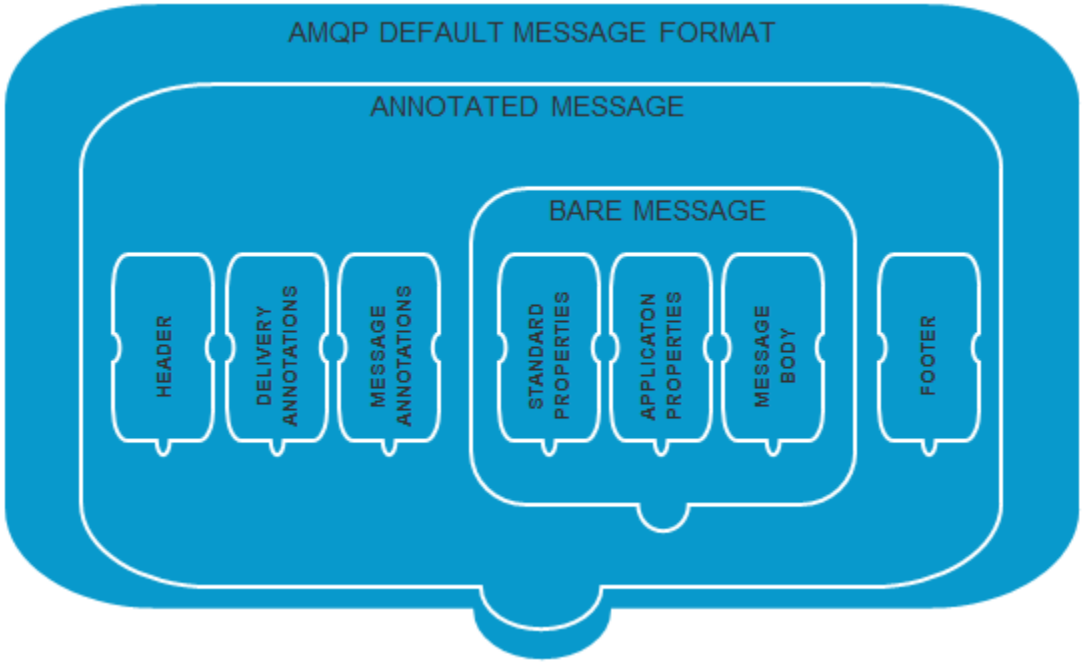
- Described
- Composite
- Restricted

NOTE: In Universal Messaging, an embedded Apache Qpid Proton-J protocol engine is used for type marshalling so all type mappings occur as defined by the Proton-J API. See the Qpid Proton page at https://qpid.apache.org/proton/index.html for details.

## AMQP Message Format

AMQP offers a standard message format (default) which can be overridden by an AMQP node. It consists of a bare message (immutable end-end) and an annotated message which may be altered by peer / intermediate nodes.

The following figure illustrates the default AMQP message format:




NOTE: Universal Messaging supports the default message format and provides a transformation layer based on it. Currently no tests have been performed with a custom AMQP message format, at the very least transformers would fail.

## Transfer Headers

| AMQP header | UM Header | AMQP Type | Java Type | UM 10.7 |
|---|---|---|---|---|
| durable | nPublished.isPersistent | boolean | boolean | ✓ |
| priority | nHeader.Priority | ubyte (Default: 4) | byte | ✓ |

| AMQP header | UM Header | AMQP Type | Java Type | UM 10.7 |
|---|---|---|---|---|
| ttl | nHeader.TTL | milliseconds (uint) | long | ✓ |
| first-acquirer | nEventProperties.JMS_AMQP_FirstAcquirer | boolean | boolean | ✓ |
| delivery-count | nHeader.RedeliveredCount | uint (Default: 0) | long | ✓ |

**Note:**
Transfer headers are always mapped irrespective of whether an AMQP transformer is used or not.

## Delivery Annotations

AMQP Delivery annotations offer an optional non-normative section for AMQP vendor defined recipients by typically providing delivery related information from the sending node to the receiving node.

| AMQP Delivery Annotation | UM Header Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| SomeDeliveryAnnotation | JMS_AMQP_DA_<AMQP SYMBOL_NAME(SomeDeliveryAnnotation)> | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |

Universal Messaging currently does not make use of any delivery annotations but will preserve any received. The naming convention used is <JMS_AMQP_DA_><delivery annotation symbol>

## Message Annotations

AMQP Message annotations offer an optional section for AMQP properties aimed at AMQP infrastructure nodes. The following table illustrates the mappings to Universal Messaging:

| AMQP Message Annotation | UM Header Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| SomeMessageAnnotation | JMS_AMQP_MA_<AMQP SYMBOL_NAME(SomeMessageAnnotation)> | AMQP Symbol / AMQP Value | String / nEventProperties Value | |
| x-opt-jms-msg-type | JMS_AMQP_MA_x-opt-jms-msg-type | x-opt-jms-msg-type / AMQP byte | String / byte (nHeader.MessageType) | ✓ |
| x-opt-jms-dest | JMS_AMQP_MA_x-opt-jms-dest | x-opt-jms-dest / AMQP String | String / byte[] (nHeader.Type) | ✓ |

| AMQP Message Annotation | UM Header Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| x-opt-reply-type | JMS_AMQP_MA_x-opt-reply-type | x-opt-reply-type / AMQP byte | String / byte (nHeader.ReplyType) | ✓ |

**Note:**
Only applies when using the Complete AMQP Transformer.

### Standard Message Properties

AMQP standard message properties is a section used for a defined set of immutable standard properties of the message. The section is part of the bare message; therefore, if retransmitted by an intermediary, it must remain unaltered.

The following table illustrates how these standard message properties are mapped to Universal Messaging:

| AMQP Standard Property | UM Header Mapping | AMQP Type | Java Value Type | UM 10.7 |
|---|---|---|---|---|
| message-id | nHeader.MessageId | AMQP String | byte[] | ✓ |
| user-id | nHeader.UserId | AMQP Binary | byte[] | ✓ |
| to | nHeader.Destination | AMQP String | byte[] | ✓ |
| subject | nEventProperties.JMS_AMQP_SUBJECT | AMQP String | String | ✓ |
| reply-to | nHeader.ReplyToName | AMQP String | byte[] | ✓ |
| correlation-id | nHeader.CorrelationId | AMQP Value | byte[] | ✓ |
| content-type | nEventProperties.JMS_AMQP_CONTENT_TYPE | AMQP Symbol | String | ✓ |
| content-encoding | nEventProperties.JMS_AMQP_CONTENT_ENCODING | AMQP Symbol | String | ✓ |
| absolute-expiry-time | N/A (Currently unsupported) | AMQP Date | long | ✗ |
| creation-time | nHeader.Timestamp | AMQP Date | long | ✓ |
| group-id | nEventProperties.JMSXGroupID | AMQP String | String | ✓ |
| group-sequence | nEventProperties.JMS_AMQP_GROUP_SEQUENCE | AMQP uint | int | ✓ |

| AMQP Standard Property | UM Header Mapping | AMQP Type | Java Value Type | UM 10.7 |
|---|---|---|---|---|
| reply-to-group-id | nEventProperties[JMS_AMQP_ReplyToGroupID] | AMQP String | String | ✓ |

**Note:**
Only applies when using the Complete AMQP Transformer.

### Application Message Properties

AMQP offers an optional application-properties section, part of the bare message used for structured application data. Intermediaries can use the data within this structure for the purposes of filtering or routing.

The keys of this map are restricted to be of type `string` (which excludes the possibility of a null key) and the values are restricted to be of simple types only, that is, excluding `map`, `list`, and `array` types. The following table illustrates how these are mapped to Universal Messaging:

| AMQP Application Message Properties | UM Dictionary Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| MAP < STRING, SIMPLE VALUE > | | AMQP String / AMQP Value | String / nEventProperties Value | ✓ |
| example_application_key1 | example_application_key1 | AMQP String / AMQP Value | String / nEventProperties Value | ✓ |

**Note:**
In Universal Messaging the nEventProperties and nHeader objects are mapped to a few AMQP message format sections. Therefore there will be property keys mappable to different sections stored in the same structure, implying that this section refers to all keys that do not fall in the other sections.

### Message Body

AMQP message body is defined in 3 possible ways:

- One or more AMQP Data sections
- One or more AMQP Sequence sections
- A single AMQP Value section

Universal Messaging currently aims to offer support for JMS 1.1 use cases via AMQP and therefore has implemented some mappings via the Complete AMQP transformer.

The mappings are based on the following UM constants:

| UM nHeader.MessageType | UM Dictionary Key (transformed) | Value (byte) | UM 10.7 |
|---|---|---|---|
| JMS_BASE_MESSAGE_TYPE | AMQP_Type | 0 | ✓ |
| JMS_MAP_MESSAGE_TYPE | AMQP_Type | 1 | ✓ |
| JMS_BYTES_MESSAGE_TYPE | AMQP_Type | 2 | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | AMQP_Type | 3 | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP_Type | 4 | ✓ |
| JMS_TEXT_MESSAGE_TYPE | AMQP_Type | 5 | ✓ |

Universal Messaging extended message types, such as protobuf, are not supported by the transformation framework.

Additionally, mappings (depending on the direction) are based on the following UM AMQP Transformation constants:

| UM AMQP Transformation Constant | UM Dictionary Key (transformed) | Value (int) | UM 10.7 |
|---|---|---|---|
| sAMQPData | AMQP_Type | 0 | ✓ |
| sAMQPValue | AMQP_Type | 1 | ✓ |
| sAMQPList | AMQP_Type | 2 | ✓ |
| sAMQPEmpty | AMQP_Type | 3 | ✓ |
| sAMQPUnknown | AMQP_Type | -1 | ✓ |

### Message Body: AMQP payloads to UM Native payloads

Initially, the Universal Messaging realm will check for an AMQP message annotation x-opt-jms-msg-type on the AMQP message. If detected and within the range of the above mentioned table, then the following conversions are performed:

| x-opt-jms-msg-type Message Annotation | AMQP Body | UM Stamping | UM payload | UM 10.7 |
|---|---|---|---|---|
| JMS_BASE_MESSAGE_TYPE | Any | AMQP_Type=sAMQPEmpty | byte[0] | ✓ |
| JMS_BYTES_MESSAGE_TYPE | null | | byte[0] | ✓ |

| x-opt-jms-msg-type Message Annotation | AMQP Body | UM Stamping | UM payload | UM 10.7 |
|---|---|---|---|---|
| JMS_BYTES_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array | ✓ |
| JMS_BYTES_MESSAGE_TYPE | AMQP Value <Binary> | AMQP_Type=sAMQPValue | Binary.Array | ✓ |
| JMS_TEXT_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_TEXT_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array (UTF-8) | ✓ |
| JMS_TEXT_MESSAGE_TYPE | AMQP Value <Binary> | AMQP_Type=sAMQPValue | Binary.Array (UTF-8) | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | null | AMQP_Type=sAMQPData | byte[0] | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array | ✓ |
| JMS_STREAM_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP Value <List> | AMQP_Type=sAMQPValue | byte[](Serialized Vector<Binary>) | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP Sequence | AMQP_Type=sAMQPList | byte[](Serialized Vector<Binary>) | ✓ |
| JMS_MAP_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_MAP_MESSAGE_TYPE | AMQP Value <Map> | AMQP_Type=sAMQPValue | byte[](Externalized fEventDictionary) | ✓ |

**Note:**
Only applies when using the Complete AMQP Transformer

If the message annotation is not present, then the mapping starts looking for payload hints in message properties as follows:

| AMQP Message Property | AMQP Body | UM nHeader.MessageType | UM payload | UM 10.7 |
|---|---|---|---|---|
| **content-type** | | | | |
| text/plain | null | JMS_TEXT_MESSAGE_TYPE | byte[0] | ✓ |
| application/x-java-serialized-object | null | JMS_OBJECT_MESSAGE_TYPE | byte[0] | ✓ |
| application/octet-stream | null | JMS_BYTES_MESSAGE_TYPE | byte[0] | ✓ |

| AMQP Message Property | AMQP Body | UM nHeader.MessageType | UM payload | UM 10.7 |
|---|---|---|---|---|
| null or other | null | JMS_BASE_MESSAGE_TYPE | byte[0] | ✓ |
| text/plain | Data | JMS_TEXT_MESSAGE_TYPE | Binary.Array(UTF-8) | ✓ |
| application/x-java-serialized-object | Data | JMS_OBJECT_MESSAGE_TYPE | Binary.Array | ✓ |
| application/octet-stream or null | Data | JMS_BYTES_MESSAGE_TYPE | Binary.Array | ✓ |
| other | Data | JMS_BYTES_MESSAGE_TYPE | Binary.Array | ✓ |
| **AMQP Value <Type> body** | | | | |
| null | AMQP Value <null> | JMS_BASE_MESSAGE_TYPE | byte[0] | ✓ |
| String | AMQP Value <String> | JMS_TEXT_MESSAGE_TYPE | Binary.Array (UTF-8) | ✓ |
| Binary | AMQP Value <Binary> | JMS_BYTES_MESSAGE_TYPE | Binary.Array | ✓ |
| List | AMQP Value <List> | JMS_STREAM_MESSAGE_TYPE | byte[](Serialized Vector<Binary>) | ✓ |
| Map | AMQP Value <Map> | JMS_MAP_MESSAGE_TYPE | byte[](Externalized fEventDictionary) | ✓ |
| other | AMQP Value <Object> | JMS_OBJECT_MESSAGE_TYPE | Binary.Array | ✓ |
| **AMQP Sequence <Object> body** | | | | |
| Binary | AMQP Sequence<Binary> | JMS_OBJECT_MESSAGE_TYPE | Binary.Array | ✓ |

**Note:**
Only applies when using the Complete AMQP Transformer.

### Message Body: UM Native Payloads to AMQP payloads

Initially, the Universal Messaging realm will check for an nEventProperties keys called AMQP_Type which is expected to be within the range defined by the AMQP Transformation Constants table. Typically these are AMQP messages previously transformed to native but as there are multiple possibilities with regard to the type of payload this should be transformed to, this value gives us a hint. If not present, it defaults to sAMQPUnknown.

Based on the UM nHeader.MessageType values the following conversions are performed:

| UM nHeader.MessageType | UM Dictionary Key AMQP_Type | AMQP Payload | UM 10.7 |
|---|---|---|---|
| JMS_BASE_MESSAGE_TYPE | sAMQPEmpty or sAMQPUnknown or sAMQPData | Data(byte[0]) | ✓ |
| JMS_BASE_MESSAGE_TYPE | sAMQPValue | AMQPValue(null) | ✓ |
| JMS_BASE_MESSAGE_TYPE | sAMQPList | | ✗ |
| JMS_MAP_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | ✓ |
| JMS_MAP_MESSAGE_TYPE | sAMQPValue or sAMQPUnknown | AMQPValue<Map> (from byte[](Externalized fEventDictionary) | ✓ |
| JMS_MAP_MESSAGE_TYPE | sAMQPList or sAMQPData | | ✗ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | ✓ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPData orsAMQPUnknown | Data(Binary(byte[])) (from UM native payload) | ✓ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPValue | AMQPValue(Binary(byte[])) (from UM native payload) | ✓ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPList | | ✗ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPData orsAMQPUnknown | Data(Binary(byte[])) (from UM native payload) | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPList or sAMQPValue | | ✗ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | ✓ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPList | AMQPSequence<Vector> (byte[](Serialized Vector<Binary>) | ✓ |

| UM nHeader.MessageType | UM Dictionary Key AMQP_Type | AMQP Payload | UM 10.7 |
|---|---|---|---|
| JMS_STREAM_MESSAGE_TYPE | sAMQPValue or sAMQPUnknown | AMQPValue<Vector> (byte[](Serialized Vector<Binary>) | ✓ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPData | | ✗ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPData | Data(Binary(byte[])) (from UM native payload) | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPValue orsAMQPUnknown | AMQPValue<String> (from UM native payload) | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPList | | ✗ |

**Note:**
Only applies when using the Complete AMQP Transformer.

### Message Footer

AMQP Message footer offers an optional non-normative section for delivery or message metadata that can be calculated / validated only after the whole message has been processed (e.g digital signature verification, encryption etc).

The following table shows how these are mapped in Universal Messaging:

| AMQP Message Footer | UM Sub Dictionary Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| MAP < SYMBOL, VALUE > | Footer | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |
| example_footer_key1 | Footer.example_footer_key1 | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |

**Note:**
Only applies when using the Complete AMQP Transformer.

# AMQP Transactions

Transactional messaging allows for the coordinated outcome of otherwise independent transfers, extending to an arbitrary number of message transfers spread across any number of distinct links in either direction.

An AMQP transaction scenario extends the roles of the two AMQP containers enclosing the Sender and Receiver nodes, to additionally act as a transactional controller and transactional resource respectively. This is established over a control link initiated by the controller, while allocation and completion are communicated using the AMQP `declare` and `discharge` type messages.

In an brokered environment such as one that involves Universal Messaging, AMQP transactions are involved in transactional publishing or transactional subscribing where the UM realm is the transactional resource and the AMQP application is the transactional controller.

## Transactional Capabilities

To initiate an AMQP transaction, the container acting as the transactional controller establishes a control link to the container acting as the transactional resource. During this handshake, the containers exchange capability maps to determine if the transactional capability can be supported. The following table illustrates these capabilities:

| AMQP Transactional Capability | AMQP Symbol | Description | UM 10.7 |
|---|---|---|---|
| Local Transactions | amqp:local-transactions | Indicates support local transactions. | ✓ |

Description: When a UM realm acts as an AMQP container, it advertises AMQP local transaction capability.

| AMQP Transactional Capability | AMQP Symbol | Description | UM 10.7 |
|---|---|---|---|
| Distributed Transactions | amqp:distributed-transactions | Indicates support AMQP Distributed Transactions | ✗ |

Description: Distributed transactions are not currently defined by the AMQP specification.

| AMQP Transactional Capability | AMQP Symbol | Description | UM 10.7 |
|---|---|---|---|
| Promotable Transactions | amqp:promotable-transactions | Indicates support AMQP Promotable Transactions | ✗ |

Description: Promotable transactions are not currently supported in Universal Messaging.

| AMQP Transactional Capability | AMQP Symbol | Description | UM 10.7 |
|---|---|---|---|
| Multi Transactions Per Session | amqp:multi-txns-per-ssn | Indicates support multiple active transactions on a single session | ✗ |

Description: Multiple transactions per session are not currently supported in Universal Messaging.

| AMQP Transactional Capability | AMQP Symbol | Description | UM 10.7 |
|---|---|---|---|
| Multi Sessions Per Transaction | amqp:multi-ssns-per-txn | Indicates support transactions whose txn-id is used across sessions on one connection | ✗ |

Description: Multiple sessions per transaction are not currently supported in Universal Messaging.

### Transaction Declare

To begin transactional work, a transaction needs to be declared. After the control link is established, the transaction controller container sends a special AMQP message to the transaction `coordinator` container, whose body consists of the `declare` type in a single `amqp-value` section. A successful transaction declare, concludes with the coordinator container settling the declaration with a disposition outcome of `declared`, allowing the controller container to use it for message transfers.

### Transaction Discharge

To conclude transactional work, a transaction needs to be discharged. This is done by the transactional controller container sending a special AMQP message to the transaction coordinator container, whose body consists of the `discharge` type in a single `amqp-value` section. In order to indicate a successful or failing discharge, the transaction controller container uses the fail flag on indicates that it wishes to commit or rollback the transactional work by setting the fail flag on the `discharge` body. Please note that the transaction coordinator can always also indicate a failing discharge if it is unable to honor the transaction controller's request.

### Transactional Delivery State

AMQP transactions introduce an additional message delivery state called transactional state, which effectively wraps a regular message delivery state with a transaction ID. As discussed in previous sections, AMQP message deliveries consist of a notion of settlement which indicates when a node gives up trying to deliver a transfer and forgets all state. AMQP transactions do not change this model but interact with it as defined in the following sections.

NOTE: In Universal Messaging, AMQP delivery tags are uniquely identified using a long:long naming scheme consisting of <store unique ID>:<EID> . In transactional scenarios, the UM native TX ID is mapped to the AMQP TX ID.

### Transactional Publishing

When the AMQP transaction controller container wishes to publish transactionally, it sets the delivery state to `transactional-state`, as explained in the previous section and pointing to a declared TX ID. This transactional state has the effect of the message not being available at the destination node of the transactional resource container, until after the transaction has been successfully discharged.

On receiving a non-settled delivery associated with a TX id, the transactional resource container sends a `disposition` response similar to the when non transactional publishing is used, with the delivery state being a `transactional-state` .

The following table illustrates how this is mapped to a successful Universal Messaging transactional publish, which is disposed with an accept state:

| Remote Disposition State | UM Transformer Result | Sender Settle Mode | UM 10.7 |
|---|---|---|---|
| transactional | NOT null | N/A | ✓ |

Description: When UM receives a non-partial delivery transactional delivery, it tries to examine and transform its contents using the configured UM transformer. Following that, an asynchronous UM TX native publish request is queued. Finally, when the UM server has processed the native TX publish request, it will accept it with a transaction state set to the TX ID and advance the link ONLY If it is not already settled. If it is already settled, no further action will be taken. This is equivalent to a UM TX publisher.

The following table illustrates how this is mapped to an unsuccessful Universal Messaging transactional publish, which is disposed with a rejected state:

| Remote Disposition State | UM Transformer Result | Sender Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|
| transactional | NOT null | N/A | amqp:unauthorized-access (fatal) amqp:resource-limit-exceeded (transient) amqp:internal-error (fatal) amqp:not-found (fatal) | ✓ |

As discussed in the accepted state mappings, a transactional AMQP publisher with these settings is mapped to an asynchronous UM TX native publish request getting queued. When the UM server has completed processing the native TX publish request, and an error has occurred, the delivery is rejected and settled as follows:

1. amqp:unauthorized-access : UM ACLs do not allow publishing for this subject. This is a fatal error so the link is closed.
2. amqp:resource-limit-exceeded: UM store capacity has been reached. This is a transient error so the link is advanced.
3. amqp:internal-error: An unexpected UM internal error has occurred. This is a fatal error so the link is closed.

4. amqp:not-found: The UM server is unable to find the destination node. This is fatal error so the link is closed.

### Transactional Subscribing

When an AMQP transaction controller container wishes to associate the outcome of a delivery with a transaction it sets the state of the delivery to a `transactional-state` as explained in previous sections. and pointing to a declared TX id. This transactional state has the effect of the message not being released from the transactional resource container, until after the transaction has been successfully discharged.

On receiving a non-settled delivery associated with a TX id, the transactional controller container sends a `disposition` response similar to the when non transactional subscribing is used, with the delivery state being a `transactional-state`.

The following table illustrates how this is mapped to an successful Universal Messaging durable topic ack / transactional queue reader commit, which is disposed with an accept state:

| Remote Disposition State | Delivery Settled | Receiver Settle Mode | Source Durability | UM 10.7 |
|---|---|---|---|---|
| transactional | N/A | N/A | N/A | ✓ |

Descruption: When UM receives an unsettled disposition frame with a remote state of transactional, it first extracts the delivery tag which is expected in the form <um store uniqueid>:<um event EID>, followed by extracting the durable name from the link. Following that it updates internal data structures in the sender session context resulting in a native durable subscriber ACK or UM native transactional queue reader commit getting queued. When the UM server has completed processing the request, the delivery is accepted and settled.

The following table illustrates how this is mapped to an unsuccessful Universal Messaging durable topic ack / transactional queue reader commit, which is disposed with a rejected state:

| Remote Disposition State | Delivery Settled | Receiver Settle Mode | AMQP Errors | UM 10.7 |
|---|---|---|---|---|
| transactional | N/A | N/A | amqp:internal-error (fatal) | ✓ |

Description: As discussed in the accepted state mappings, a transactional AMQP consumer with these settings is mapped to a topic durable consumer or a queue transactional reader commit getting queued. When the UM server has completed processing the request, and an error has occurred, the delivery is rejected and settled as follows:

- amqp:internal-error: An unexpected UM internal error has occurred. This is a fatal error so the link is closed.

## Supported AMQP Client Libraries

Universal Messaging has been tested against a limited number of AMQP client libraries. However, you can still choose to work with many free and commercial AMQP libraries that are currently available. If you want to use an untested client library, contact Software AG Global Support.

Universal Messaging 10.7 has been tested against the following Apache Qpid client libraries for AMQP 1.0, located in the *Software AG_directory* \UniversalMessaging\lib\apache-qpid directory:

- Qpid JMS client version 0.3.0
- Qpid Legacy JMS version 0.32

## Appendix A: AMQP Terminology to Universal Messaging Terminology

| AMQP Term | Universal Messaging Term | Comments |
|---|---|---|
| Container | AMQP Client or UM Realm | UM Realms receive but do not initiate connections over AMQP |
| Connection | AMQP Connection | TCP and SSL<br>Web socket, Session Multiplexing and negotiated TLS not supported |
| Session | Logical mapping to UM session | An AMQP session is mapped to a native UM session. Currently this is limited to one per connection (no multiplexing) |
| Link (inbound) | Publications | Delivery resumption and Message fragmentation not supported |
| Link (outbound) | Subscriptions | Link credit values < 100 not reliably supported<br>Delivery resumption and Message fragmentation not currently supported |
| Source (on broker) | Queue or Topic on realm | Data groups are not supported |
| Target (on broker) | Queue or Topic on realm | Data groups are not supported |
| Delivery | The delivery of a message in either direction (pub or sub) | Link recovery is currently unsupported. |
| Message | A UM message | Message fragmentation is currently unsupported. |

| AMQP Term | Universal Messaging Term | Comments |
|---|---|---|
| **Not addressed in AMQP specification** | Clustering | UM specific capability.<br>There is no automatic failover of AMQP clients to a new AMQP server in a cluster if the connection to the current AMQP server is lost. |
| **Not addressed in AMQP specification** | Follow the master<br>Paused publishing | UM specific capability<br>Requires specific support in the client |
| **Not addressed in AMQP specification** | Zones<br>Joins (including filtered) | UM specific capability<br>Requires complete transformation of AMQP messages<br>Requires specific testing for UM |

## Appendix B: AMQP Messages to Universal Messaging Message Mappings

| AMQP Transfer Header | UM Header | AMQP Type | Java Type | UM 10.7 |
|---|---|---|---|---|
| durable | nPublished.isPersistent | boolean | boolean | ✓ |
| priority | nHeader.Priority | ubyte (Default: 4) | byte | ✓ |
| ttl | nHeader.TTL | milliseconds (uint) | long | ✓ |
| first-acquirer | nEventProperties.JMS_AMQP_FirstAcquirer | boolean | boolean | ✓ |
| delivery-count | nHeader.RedeliveredCount | uint (Default: 0) | long | ✓ |

| AMQP Delivery Annotation | UM Header Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| SomeDeliveryAnnotation | JMS_AMQP_DeliveryAnnotation.SomeDeliveryAnnotation | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |

| AMQP Message Annotation | UM Header Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| SomeMessageAnnotation | JMS_AMQP_MA_<AMQP SYMBOL_NAME(SomeMessageAnnotation)> | AMQP Symbol / AMQP Value | String / nEventProperties Value | |
| x-opt-jms-msg-type | JMS_AMQP_MA_x-opt-jms-msg-type | x-opt-jms-msg-type / AMQP byte | String / byte (nHeader.MessageType) | ✓ |
| x-opt-jms-dest | JMS_AMQP_MA_x-opt-jms-dest | x-opt-jms-dest / AMQP String | String / byte[] (nHeader.Type) | ✓ |
| x-opt-reply-type | JMS_AMQP_MA_x-opt-reply-type | x-opt-reply-type / AMQP byte | String / byte (nHeader.ReplyType) | ✓ |

| AMQP Standard Property | UM Header Mapping | AMQP Type | AMQP Type | UM 10.7 |
|---|---|---|---|---|
| message-id | nHeader.MessageId | AMQP String | byte[] | ✓ |
| user-id | nHeader.UserId | AMQP Binary | byte[] | ✓ |
| to | nHeader.Destination | AMQP String | byte[] | ✓ |
| subject | nHeader.Type | AMQP String | String | ✓ |
| reply-to | nHeader.ReplyToName | AMQP String | byte[] | ✓ |
| correlation-id | nHeader.CorrelationId | AMQP Value | byte[] | ✓ |
| content-type | nEventProperties.JMS_AMQP_CONTENT_TYPE | AMQP Symbol | String | ✓ |
| content-encoding | nEventProperties.JMS_AMQP_CONTENT_ENCODING | AMQP Symbol | String | ✓ |
| absolute-expiry-time | N/A | AMQP Date | long | ✗ |
| creation-time | nHeader.Timestamp | AMQP Date | long | ✓ |
| group-id | nEventProperties.JMSXGroupID | AMQP String | String | ✓ |
| group-sequence | nEventProperties.JMS_AMQP_GROUP_SEQUENCE | AMQP uint | int | ✓ |
| reply-to-group-id | nEventProperties.JMS_AMQP_ReplyToGroupID | AMQP String | String | ✓ |

| AMQP Application Properties | UM Dictionary Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| MAP < STRING, SIMPLE VALUE > | | AMQP String / AMQP Value | String / nEventProperties Value | ✓ |
| example_application_key1 | example_application_key1 | AMQP String / AMQP Value | String / nEventProperties Value | ✓ |
| UM nHeader.MessageType | UM Dictionary Key (transformed) | AMQP Type | Java Value | |
| JMS_BASE_MESSAGE_TYPE | AMQP_Type | ubyte | 0 | ✓ |
| JMS_MAP_MESSAGE_TYPE | AMQP_Type | ubyte | 1 | ✓ |
| JMS_BYTES_MESSAGE_TYPE | AMQP_Type | ubyte | 2 | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | AMQP_Type | ubyte | 3 | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP_Type | ubyte | 4 | ✓ |
| JMS_TEXT_MESSAGE_TYPE | AMQP_Type | ubyte | 5 | ✓ |

| UM AMQP Transformation Constant | UM Dictionary Key (transformed) | AMQP Type | Java Value | UM 10.7 |
|---|---|---|---|---|
| sAMQPData | AMQP_Type | N/A | 0 | ✓ |
| sAMQPValue | AMQP_Type | N/A | 1 | ✓ |
| sAMQPList | AMQP_Type | N/A | 2 | ✓ |
| sAMQPEmpty | AMQP_Type | N/A | 3 | ✓ |
| sAMQPUnknown | AMQP_Type | N/A | -1 | ✓ |

| x-opt-jms-msg-type Message Annotation | AMQP Body | UM Stamping | UM payload | UM 10.7 |
|---|---|---|---|---|
| JMS_BASE_MESSAGE_TYPE | Any | AMQP_Type=sAMQPEmpty | byte[0] | ✓ |
| JMS_BYTES_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_BYTES_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array | ✓ |

| x-opt-jms-msg-type Message Annotation | AMQP Body | UM Stamping | UM payload | UM 10.7 |
|---|---|---|---|---|
| JMS_BYTES_MESSAGE_TYPE | AMQP Value <Binary> | AMQP_Type=sAMQPValue | Binary.Array | ✓ |
| JMS_TEXT_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_TEXT_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array (UTF-8) | ✓ |
| JMS_TEXT_MESSAGE_TYPE | AMQP Value <Binary> | AMQP_Type=sAMQPValue | Binary.Array (UTF-8) | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | null | AMQP_Type=sAMQPData | byte[0] | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | Data | AMQP_Type=sAMQPData | Binary.Array | ✓ |
| JMS_STREAM_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP Value <List> | AMQP_Type=sAMQPValue | byte[](Serialized Vector<Binary>) | ✓ |
| JMS_STREAM_MESSAGE_TYPE | AMQP Sequence | AMQP_Type=sAMQPList | byte[](Serialized Vector<Binary>) | ✓ |
| JMS_MAP_MESSAGE_TYPE | null | | byte[0] | ✓ |
| JMS_MAP_MESSAGE_TYPE | AMQP Value <Map> | AMQP_Type=sAMQPValue | byte[](Externalized fEventDictionary) | ✓ |

| AMQP Message Property | AMQP Body | UM nHeader.MessageType | UM payload | UM 10.7 |
|---|---|---|---|---|
| **content-type** | | | | |
| text/plain | null | JMS_TEXT_MESSAGE_TYPE | byte[0] | ✓ |
| application/x-java-serialized-object | null | JMS_OBJECT_MESSAGE_TYPE | byte[0] | ✓ |
| application/octet-stream | null | JMS_BYTES_MESSAGE_TYPE | byte[0] | ✓ |
| null or other | null | JMS_BASE_MESSAGE_TYPE | byte[0] | ✓ |
| text/plain | Data | JMS_TEXT_MESSAGE_TYPE | BinaryArray(UTF8) | ✓ |
| application/x-java-serialized-object | Data | JMS_OBJECT_MESSAGE_TYPE | Binary.Array | ✓ |
| application/octet-stream or null | Data | JMS_BYTES_MESSAGE_TYPE | Binary.Array | ✓ |

| AMQP Message Property | AMQP Body | UM nHeader.MessageType | UM payload | UM 10.7 |
|---|---|---|---|---|
| other | Data | JMS_BYTES_MESSAGE_TYPE | Binary Array | ✓ |
| **AMQP Value <Type> body** | | | | |
| null | AMQP Value <null> | JMS_BASE_MESSAGE_TYPE | byte[0] | ✓ |
| String | AMQP Value <String> | JMS_TEXT_MESSAGE_TYPE | Binary Array (UTF-8) | ✓ |
| Binary | AMQP Value <Binary> | JMS_BYTES_MESSAGE_TYPE | Binary Array | ✓ |
| List | AMQP Value <List> | JMS_STREAM_MESSAGE_TYPE | byte[](Serialized Vector<Binary>) | ✓ |
| Map | AMQP Value <Map> | JMS_MAP_MESSAGE_TYPE | byte[](Externalized fEventDictionary) | ✓ |
| other | AMQP Value <Object> | JMS_OBJECT_MESSAGE_TYPE | Binary Array | ✓ |
| **AMQP Sequence <Object> body** | | | | |
| Binary | AMQP Sequence<Binary> | JMS_OBJECT_MESSAGE_TYPE | Binary Array | ✓ |
| JMS_BASE_MESSAGE_TYPE | sAMQPEmpty or sAMQPUnknown or sAMQPData | Data(byte[0]) | | ✓ |
| JMS_BASE_MESSAGE_TYPE | sAMQPValue | AMQPValue(null) | | ✓ |
| JMS_BASE_MESSAGE_TYPE | sAMQPList | | | ✗ |
| JMS_MAP_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | | ✓ |
| JMS_MAP_MESSAGE_TYPE | sAMQPValue or sAMQPUnknown | AMQPValue<Map> (from byte[](Externalized fEventDictionary) | | ✓ |
| JMS_MAP_MESSAGE_TYPE | sAMQPList or sAMQPData | | | ✗ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | | ✓ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPData orsAMQPUnknown | Data(Binary(byte[])) (from UM native payload) | | ✓ |

| AMQP Message Property | AMQP Body | UM nHeader.MessageType | UM payload | UM 10.7 |
|---|---|---|---|---|
| JMS_BYTES_MESSAGE_TYPE | sAMQPValue | AMQPValue(Binary(byte[])) (from UM native payload) | | ✓ |
| JMS_BYTES_MESSAGE_TYPE | sAMQPList | | | ✗ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPData orsAMQPUnknown | Data(Binary(byte[])) (from UM native payload) | | ✓ |
| JMS_OBJECT_MESSAGE_TYPE | sAMQPList or sAMQPValue | | | ✗ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | | ✓ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPList | AMQPSequence<Vector> (byte[](Serialized Vector<Binary>) | | ✓ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPValue or sAMQPUnknown | AMQPValue<Vector> (byte[](Serialized Vector<Binary>) | | ✓ |
| JMS_STREAM_MESSAGE_TYPE | sAMQPData | | | ✗ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPEmpty | Data(byte[0]) | | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPData | Data(Binary(byte[])) (from UM native payload) | | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPValue orsAMQPUnknown | AMQPValue<String> (from UM native payload) | | ✓ |
| JMS_TEXT_MESSAGE_TYPE | sAMQPList | | | ✗ |

| AMQP Message Footer | UM Sub Dictionary Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| MAP < SYMBOL, VALUE > | Footer | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |

| AMQP Message Footer | UM Sub Dictionary Key | AMQP Key/Value Type | Java Key/Value Type | UM 10.7 |
|---|---|---|---|---|
| example_footer_key1 | Footer.example_footer_key1 | AMQP Symbol / AMQP Value | String / nEventProperties Value | ✓ |

# Appendix C: Universal Messaging Support for AMQP 1.0

| AMQP Term | Support Comment |
|---|---|
| Types | ProtonJ implementation which has all types implemented and we depend on that |
| 2.1 Transport | |
| 2.1.1 Conceptual Model | Universal Messaging supports AMQP concepts relevant to a messaging broker |
| 2.1.2 Communication Endpoints | UM provides support for 2 channels per AMQP connection (no multiplexing) |
| 2.1.3 Protocol Frames | Supported |
| 2.2 Version Negotiation | UM only supports version 1.0.0 |
| 2.3 Framing | |
| 2.3.1 Frame Layout | Supported |
| 2.3.2 AMQP Frames | AMQP Maximum Frame size is limited to the configured UM Maximum Buffer Size |
| 2.4 Connections | |
| 2.4.1 Opening A Connection | Not supported - UM is a broker |
| 2.4.2 Pipelined Open | Not supported |
| 2.4.3 Closing A Connection | Supported |
| 2.4.4 Simultaneous Close | Supported |
| 2.4.5 Idle Timeout Of A Connection | Supported - timeout is limited to the configured UM Connection Keep Alive |
| 2.4.6 Connection States | Supported - ProtonJ |
| 2.4.7 Connection State Diagram | |
| 2.5 Sessions | |

| AMQP Term | Support Comment |
|---|---|
| 2.5.1 Establishing A Session | Not supported - UM is a broker |
| 2.5.2 Ending A Session | Supported |
| 2.5.3 Simultaneous End | Supported |
| 2.5.4 Session Errors | Supported - ProtonJ |
| 2.5.5 Session States | Supported - ProtonJ |
| 2.5.6 Session Flow Control | Supported - ProtonJ |
| 2.6 Links | |
| 2.6.1 Naming A Link | Supported - ProtonJ |
| 2.6.2 Link Handles | Supported - ProtonJ |
| 2.6.3 Establishing Or Resuming A Link | Not supported - UM is a broker |
| 2.6.4 Detaching And Reattaching A Link | Supported |
| 2.6.5 Link Errors | Supported - ProtonJ |
| 2.6.6 Closing A Link | Supported |
| 2.6.7 Flow Control | Supported - UM currently is unable to strictly honour window sizes < 100 |
| 2.6.8 Synchronous Get | Supported |
| 2.6.9 Asynchronous Notification | Supported |
| 2.6.10 Stopping A Link | Supported |
| 2.6.11 Messages | |
| 2.6.12 Transferring A Message | Supported |
| 2.6.13 Resuming Deliveries | Not supported - UM will re-publish all un-settled transfers if the connection is dropped |
| 2.6.14 Transferring Large Messages | Supported - AMQP Max Frame size is limited to the configured UM Max Buffer Size |
| 2.7 Performatives | |
| 2.7.1 Open | Supported - ContainerID is mapped to UM realm name and subject to previously noted limitations |
| 2.7.2 Begin | Supported -Subject to previously noted limitations |

| AMQP Term | Support Comment |
|---|---|
| 2.7.3 Attach | Supported -Subject to previously noted limitations |
| 2.7.4 Flow | Supported -Subject to previously noted limitations |
| 2.7.5 Transfer | Supported -Subject to previously noted limitations |
| 2.7.6 Disposition | Supported -Subject to previously noted limitations |
| 2.7.7 Detach | Supported |
| 2.7.8 End | Supported |
| 2.7.9 Close | Supported |
| 2.8 Definitions | |
| 2.8.1 Role | Supported - UM is a broker |
| 2.8.2 Sender Settle Mode | Supported |
| 2.8.3 Receiver Settle Mode | Supported |
| 2.8.4 Handle | Supported |
| 2.8.5 Seconds | Supported |
| 2.8.6 Milliseconds | Supported |
| 2.8.7 Delivery Tag | Supported |
| 2.8.8 Delivery Number | Supported |
| 2.8.9 Transfer Number | Supported |
| 2.8.10 Sequence No | Supported |
| 2.8.11 Message Format | Supported |
| 2.8.12 IETF Language Tag | Supported |
| 2.8.13 Fields | Supported |
| 2.8.14 Error | Supported |
| 2.8.15 AMQP Error | Supported |
| 2.8.16 Connection Error | Supported |
| 2.8.17 Session Error | Supported - Connection re-directs not supported |
| 2.8.18 Link Error | Supported |
| 2.8.19 Constant Definitions | Supported |

| AMQP Term | Support Comment |
|---|---|
| 3 Messaging | |
| 3.2 Message Format | |
| 3.2.1 Header | Supported |
| 3.2.2 Delivery Annotations | Supported |
| 3.2.3 Message Annotations | Supported |
| 3.2.4 Properties | Supported - mapped to the equivalent JMS properties if present |
| 3.2.5 Application Properties | Supported - mapped to UM event dictionary |
| 3.2.6 Data | Supported |
| 3.2.7 AMQP Sequence | Supported |
| 3.2.8 AMQP Value | Supported |
| 3.2.9 Footer | Supported |
| 3.2.10 Annotations | Supported |
| 3.2.11 Message ID Ulong | Supported |
| 3.2.12 Message ID UUID | Supported |
| 3.2.13 Message ID Binary | Supported |
| 3.2.14 Message ID String | Supported |
| 3.2.15 Address String | Supported |
| 3.2.16 Constant Definitions | Supported |
| 3.3 Distribution Nodes | Supported |
| 3.4 Delivery State | |
| 3.4.1 Received | Supported |
| 3.4.2 Accepted | Supported |
| 3.4.3 Rejected | Supported |
| 3.4.4 Released | Not Supported |
| 3.4.5 Modified | Not Supported |
| 3.4.6 Resuming Deliveries Using Delivery States | Not Supported |
| 3.5 Sources and Targets | |

| AMQP Term | Support Comment |
|---|---|
| 3.5.1 Filtering Messages | Supported - UM is a broker, only JMS message selectors supported for subscriptions |
| 3.5.2 Distribution Modes | Supported |
| 3.5.3 Source | Supported - UM is a broker |
| 3.5.4 Target | Supported - UM is a broker |
| 3.5.5 Terminus Durability | Supported - UM is a broker |
| 3.5.6 Terminus Expiry Policy | Supported - UM is a broker mapped to UM store TTL |
| 3.5.7 Standard Distribution Mode | Supported - Move used for queues, copy used for topics |
| 3.5.8 Filter Set | Supported - UM is a broker |
| 3.5.9 Node Properties | Supported - UM is a broker |
| 3.5.10 Delete On Close | Not Supported |
| 3.5.11 Delete On No Links | Not Supported |
| 3.5.12 Delete On No Messages | Not Supported |
| 3.5.13 Delete On No Links Or Messages | Not Supported |
| 4 Transactions | |
| 4.1 Transactional Messaging | |
| 4.2 Declaring a Transaction | Supported for UM controlled transactions (local) |
| 4.3 Discharging a Transaction | Supported for UM controlled transactions (local) |
| 4.4 Transactional Work | |
| 4.4.1 Transactional Posting | Supported for UM controlled transactions (local) |
| 4.4.2 Transactional Retirement | Supported for UM controlled transactions (local) |
| 4.4.3 Transactional Acquisition | XA Not Supported - UM controlled transactions (local) are supported, |
| 4.4.4 Interaction Of Settlement With Transactions | |
| 4.4.4.1 Transactional Posting | Not Supported |
| 4.4.4.2 Transactional Retirement | Not Supported |
| 4.4.4.3 Transactional Acquisition | Not Supported |

| AMQP Term | Support Comment |
|---|---|
| 4.5 Coordination | Supported - UM is a broker |
| 4.5.1 Coordinator | Supported - UM only supports local transactions |
| 4.5.2 Declare | Supported - UM only supports local transactions |
| 4.5.3 Discharge | Supported - UM only supports local transactions |
| 4.5.4 Transaction ID | Supported - UM only supports local transactions |
| 4.5.5 Declared | Supported - UM only supports local transactions |
| 4.5.6 Transactional State | Supported - UM only supports local transactions |
| 4.5.7 Transaction Capability | Supported - UM only supports local transactions |
| 4.5.8 Transaction Error | Supported - UM only supports local transactions - Subject to UM realm Transaction TTL configuration |
| 5 Security | |
| 5.1 Security Layers | Supported - Limited to JVM SASL support |
| 5.2 TLS | Negotiated TLS not supported, use alternative instead |
| 5.2.1 Alternative Establishment | Supported - Limited to JSSE / JVM TLS support, mapped to NSPS interface |
| 5.2.2 Constant Definitions | Supported |
| 5.3 SASL | |
| 5.3.1 SASL Frames | Supported - ProtonJ |
| 5.3.2 SASL Negotiation | Supported - ProtonJ |
| 5.3.3 Security Frame Bodies | Supported - ProtonJ |
| 5.3.3.1 SASL Mechanisms | Supported - ProtonJ |
| 5.3.3.2 SASL Init | Supported - ProtonJ |
| 5.3.3.3 SASL Challenge | Supported - ProtonJ |
| 5.3.3.4 SASL Response | Supported - ProtonJ |
| 5.3.3.5 SASL Outcome | Supported - ProtonJ |
| 5.3.3.6 SASL Code | Supported - ProtonJ |

# 11 Product Usage Metrics

The Universal Messaging server contains a metering feature that can be used to collect and report product usage metrics. These metrics can be used to meter your usage of Universal Messaging, and will allow transaction-based pricing to be applied.

If you are using a Universal Messaging transaction-enabled license, the collected metrics will be reported to Software AG's transaction metering (LMA) server at 10 minute intervals. Metrics will be reported as delta values.

If you are not using a Universal Messaging transaction-enabled license, Universal Messaging will collect the metrics, but will not report them to the metering server. The metrics will in this case only be logged at log level "Info".

The metrics collected, with or without the transaction-enabled license, are:

- BytesIn - the total number of bytes in all published events
- BytesOut - the total number of bytes in all consumed events
- MessagesIn - the total number of published messages
- MessagesOut - the total number of consumed messages
- ConnectionCount- the number of user connections established during the metering period

Metrics collected from connections established by Integration Server will not be reported to the LMA server but will still be collected and logged. This is done because the operations made by these connections are already metered by Integration Server as part of a high level transaction.

If a Universal Messaging transaction-based licence is installed but Universal Messaging can't properly initialize the local LMA agent, the Universal Messaging server will not start, and the exact reason will be logged in the Universal Messaging server log adjacent to the error message: "Error while initializing LMA metering agent". If this occurs, contact Software AG's customer support for information on how to resolve the issue.

# 12 Commonly Used Features


- Overview ..... 272
- Sessions ..... 272
- Channel Attributes ..... 272
- Storage Properties of Channels and Queues ..... 277
- Channel Publish Keys ..... 279
- Queue Attributes ..... 281
- Multi-File Disk Stores ..... 282
- Events ..... 288
- Native Communication Protocols ..... 301
- Comet Communication Protocols ..... 304
- Google Protocol Buffers ..... 306
- Using the Shared Memory Protocol ..... 308
- Rollback ..... 309
- Out-of-Memory Protection ..... 309
- Pause Publishing ..... 311
- Priority Messaging ..... 313

# 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.

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

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

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.

See the section *Communication Protocols and RNAMEs* in the Universal Messaging Concepts guide for more information on RNAMEs and communication protocols.

# Channel Attributes

Universal Messaging channels provide a set of attributes that define the behavior of the events published and stored by the Universal Messaging realm server. Each event published onto a channel has a unique *event ID* within the channel. 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 Name

This is the name that you assign to the channel.

For information about permissible channel names, see the topic *Creating Channels* in the Enterprise Manager section of the *Administration Guide*.

## Channel Type

Each channel has an associated channel type. Universal Messaging channels can be of the types described in the following table. The difference lies in the type of physical storage used for each type and the performance overhead associated with each type.

| Channel Type | Description |
|---|---|
| Persistent | 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 one or more files for each channel on that realm, thus facilitating migrating whole channels to different realms.<br><br>For information about setting up a multi-file disk store for persistent channels or persistent queues, see the section "Multi-File Disk Stores" on page 282.<br><br>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 the messages are purged or removed as a result of a "Time to Live" (TTL) or capacity policy.<br><br>Messages purged from a Persistent channel are marked as deleted; however the store size will not be reduced immediately. If the store is backed by a single file, the store size will not be reduced until manual maintenance is performed on the channel using the Enterprise Manager or an Administration API call, or when auto-maintenance is triggered. Stores backed by multiple files will be cleaned up through the removal of individual files, according to the conditions described in the section "Multi-File Disk Stores" on page 282. This augments the high performance of the Universal Messaging realm. |
| Mixed | Mixed channels allow you 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 you supply a TTL for an event, this is used instead of the channel TTL.<br><br>Events stored persistently are either stored on a single or multiple files in the same manner as a Persistent channel. |
| Reliable | 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 never to 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. |

### Channel TTL

The TTL ("time to live") 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 realm server. If you have configured a dead event store for the channel, the removed event will be placed in the dead event store.

**Note:**
The exact time at which an event is automatically removed by the realm server can be slightly later than the time resulting from the TTL. This is because the removal is done at regular intervals as scheduled by the realm server, rather than on an immediate basis.

For stores of type Mixed you can specify the TTL attribute for individual events. Events on queues and channels of type Persistent or Reliable use the TTL set on store level and ignore any event-level TTL.

### Channel Capacity

The capacity of a channel defines the maximum number of events that 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.

Example: 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 oldest event on the channel will be automatically removed by the server to make room for the new event.

If you have configured a dead event store for the channel, the removed event will be placed in the dead event store.

**Note:**
You can use the storage property `Honour Capacity` to prevent further events being published to the channel if the channel capacity has been reached. In this case, newly arriving events are discarded until there is free capacity on the channel. See the section "Storage Properties of Channels and Queues" on page 277 for details. If you have configured a dead event store for the channel, the discarded events will NOT be placed in the dead event store; instead, they are simply ignored.

### Dead Event Store

When events are removed automatically by the realm server, either by the capacity policy of the channel, or the age (TTL) policy of the channel, or the age (TTL) policy of individual events, 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 configured to use a dead event store, and any events that are removed automatically will be moved into the dead event store. The dead event store can itself be a channel or a queue and can be created with any attributes you wish.

**Note:**

When there are no consumers for a particular channel and if you publish events to that particular channel, the events will be automatically moved to the dead event store. See the section "Use JMS Engine / Use Merge Engine" on page 275 for related information.

**Dead Event Store restrictions for durable subscriptions**

If you have defined one or more durable subscriptions on the channel, the automatic mechanism for moving events from the channel to the dead event store can produce unpredictable results in the dead event store, in which case you should not rely on this mechanism in your production environment. These unpredictable results when using durable subscriptions can be summarized as follows:

- Durable subscriptions of type *Serial* or *Shared*: If you use *event filtering* for these types, the dead event store mechanism can produce unpredictable results. However, if you do not use event filtering, the dead event store mechanism will work as expected.
- Durable subscriptions of other types: In general, the behavior of the dead event store is unpredictable.

## Storage Properties

Each channel has a number of storage properties associated with it, that allow you to configure operational aspects of the channel. You can set values for these properties when you create or edit the channel attributes. See the section "Storage Properties of Channels and Queues" on page 277 for details of the storage properties.

## Protobuf Descriptor

If you intend to use Google protocol buffers, you can define which protobuf descriptor file will be associated with the channel.

For information about Google protocol buffers, see the section *Google Protocol Buffers* in the *Concepts* guide.

## Use JMS Engine / Use Merge Engine

By default, Universal Messaging retains all events on any given channel until the events reach their individual time-to-live limit (see the description of Channel TTL above).

This default behavior can be modified by selecting either of the following optional engines:

| Engine | Description |
|---|---|
| JMS Engine | The JMS Engine deals with JMS topics within Universal Messaging.<br><br>The JMS engine causes events to remain on the channel as long as there is "interest" in receiving the events. In this sense, "interest" means that at least one of the following conditions is met: |

| Engine | Description |
|---|---|
| | ■ there are one or more active non-durable subscribers on the channel who have not yet consumed the events;<br>■ there are one or more durable subscribers on the channel who have not yet consumed the events.<br>When there is no more "interest" for an event, the event will be deleted automatically.<br>For further information on the JMS engine, see the section *Fanout Engine* in the Developer Guide. |
| Merge Engine | The merge engine is similar to the default engine, in that events are retained until they reach their TTL limit.<br>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.<br>In this scenario, the first time an event is published, it is published with all of its data fields. Thereafter, if another event occurs in which only a small number of the data fields have changed compared to the original published event, Universal Messaging allows you to send an event containing just the modified data fields. This process is known as *delta delivery*, and the events that contain just the modified fields are called *event deltas*.<br>This pattern can be continued over a set of published events, so that each published event just contains the delta from the previous published event in the set. By publishing just delta events rather than full events, the amount of data being transferred on the channel can be considerably reduced.<br>For further details on the merge engine and event deltas, see the appropriate language-specific section *The Merge Engine and Event Deltas* in the Developer Guide. |

The merge engine and the JMS engine are mutually exclusive: you can choose one but not both of these engines. If you choose neither of these engines, the default engine is used.

## Channel Keys

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 section "Channel Publish Keys" on page 279.

## Note on differences between Administration API and Enterprise Manager for Channel Creation

If you use the Administration API to create a channel rather than using the Enterprise Manager, some channel attributes must be set explicitly that in the Enterprise Manager are set implicitly according to context.

For example, the API offers a "cluster wide" flag, indicating that the new channel will be created on all cluster realm nodes. In the Enterprise Manager, this flag is not available directly, but will be applied automatically to the channel if you create the channel in the context of the cluster node rather than in the context of a realm node.

## Storage Properties of Channels and Queues

Each channel or queue has a number of storage properties associated with it. You can set values for these properties when you create the channel or queue.

A summary of these properties can be seen below:

| Property | Description |
|---|---|
| Auto Maintenance | Controls whether the persistent store of the channel or queue will be maintained automatically. When this property is activated, then any time thereafter when the auto-maintenance threshold values are reached for the store, the following events are removed from the persistent store:<br>■ events which have been purged (i.e., marked as deleted but not yet physically removed)<br>■ events that have reached their TTL<br>The auto-maintenance threshold values are defined globally for all stores in a realm by realm configuration properties such as `AutoMaintenanceThreshold` and `MaintenanceFileSizeThreshold`. See the *Administration Guide* for information on these properties.<br>**Note:**<br>The auto-maintenance feature is available for single-file disk stores only. If you are using a multi-file disk store, the auto-maintenance feature is not available for that multi-file disk store, and any configuration properties that you set for auto-maintenance will be ignored for this store. For further information on single-file and multi-file disk stores, see the entry for `Events per Spindle` in this table. |
| Honour Capacity | Controls whether the channel / queue capacity setting will prevent publishing of any more data once the channel / queue is full.<br>If set to `true`, the client will get an exception on further publishes if a channel / queue is already full. A transactional publish will receive an exception on the commit call, a non-transactional publish will receive an asynchronous exception through the nAsyncExceptionHandler.<br>If set to `false`, the oldest event in the channel / queue will be purged to make room for the newest event.<br>**Note:** |

| Property | Description |
|---|---|
| | If this property is set to `true`, the capacity can nevertheless be exceeded by a transaction if the capacity had not yet been reached at the beginning of the transaction. |
| Enable Caching | Controls 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.<br><br>If you want to use caching with a multi-file store, set both the `EnableCaching` server property for the Universal Messaging server instance and the **Enable Caching** storage property for the channel to `true`. If the `EnableCaching` server property is set to `false`, the Universal Messaging server instance always uses `false` as the value of the **Enable Caching** storage property, which disables caching for the channel. |
| Cache on Reload | When a server restarts it will scan all file based stores and check for corruption. During this test the default behavior 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 | Controls the read buffering logic for the store on the server.<br><br>This property is deprecated and will be removed in a future product release. The planned future behavior is that store read buffering will always be activated. |
| Enable Multicast | This activates multicast processing, thereby allowing multicast clients to receive events over multicast connections. |
| Read Buffer Size | If Read Buffering is enabled, then this function sets the size in bytes of the buffer to use.<br><br>This property is deprecated and will be removed in a future product release. The planned future behavior is that the store read buffer size will be set using the "StoreReadBufferSize" global realm configuration setting. |
| Sync Each Write | Controls 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 | Controls 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 | Controls 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 | Controls whether all events fanned out are written to an archive. |
| Priority | Sets a default priority for all events in the channel. The priority is a numeric value in the range `0` (lowest priority) to `9` (highest priority). |

| Property | Description |
|---|---|
| | You can override the default priority for an event by setting its value explicitly.<br><br>For more information about event priorities, see the section *Priority Messaging* in the corresponding language-specific section of the Developer Guide. |
| Stamp Dictionary | Configures whether events on this channel/queue should be stamped by the server. Possible values are:<br><br>■ `nChannelAttributes.DICTIONARY_STAMPING_ENABLED` - event dictionary stamping is explicitly enabled<br><br>■ `nChannelAttributes.DICTIONARY_STAMPING_DISABLED` - event dictionary stamping is explicitly disabled<br><br>■ `nChannelAttributes.DICTIONARY_STAMPING_DEFAULT` - the server-wide configuration takes effect (default value in case the property is not set) |
| Events Per Spindle | Defines the maximum number of events allowed per file in a multi-file disk store for persistent channels or persistent queues. Each such disk store file is called a *spindle*. If all existing files are full and a new event is published, a new file will be created to append further events for this store.<br><br>If this value is set to `0`, then the store will persist all events in a single-file disk store.<br><br>For more information on multi-file disk stores, including size recommendations for events per spindle, see the section "Multi-File Disk Stores" on page 282. |

# 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 (*nEventProperties*) for the key defined as the **Key Name** are identical. The channel will store the specified number of most recent events whose values match the **Key Name**.

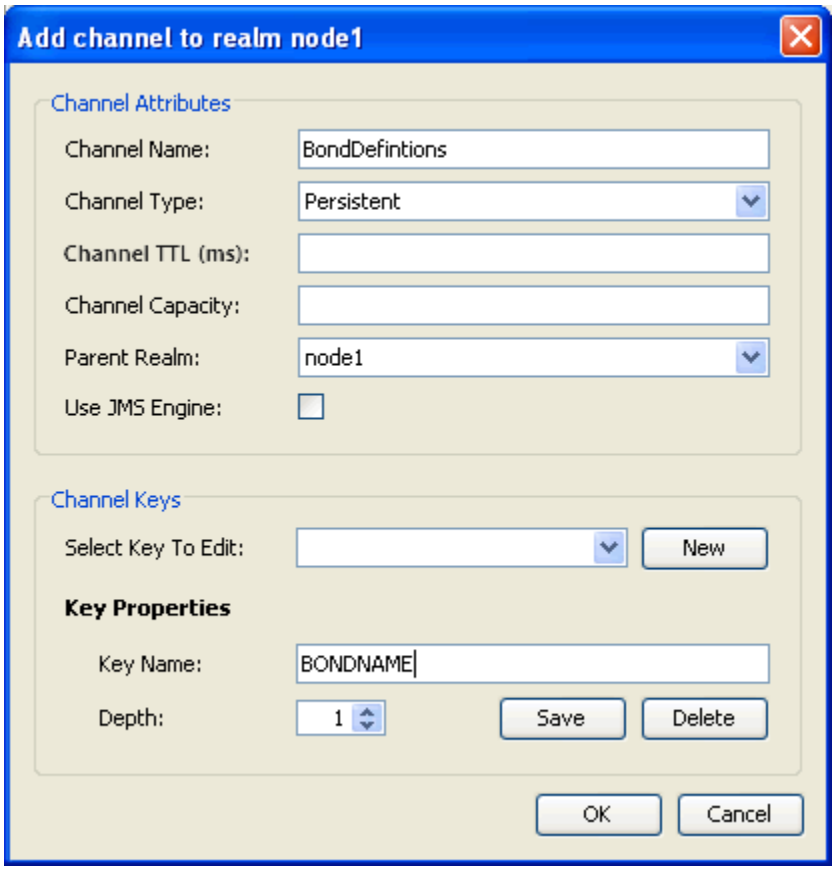
Channel Publish Keys enable the implementation of Last Value Caches. In a last value cache, only the most recent value for a given type of event is kept on the channel. In high-update situations, where only the most recent values are of interest, Channel Publish Keys can greatly improve efficiency in 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 that has a Channel Key called BONDNAME with a depth of 1, as shown in the following Enterprise Manager screen shot:




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 Channel Key, type the name of the key into the **Key Name** box on the dialog and click **Save**. If you want the key to have a depth of greater than 1 then click the up arrow adjacent to the **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, then you would create the channel key with depth 3 (implying that a maximum of 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 queues provide 3 main attributes. Depending on the options chosen, these define the behavior 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.

For stores of type Mixed you can specify the TTL attribute for individual events. Events on queues and channels of type Persistent or Reliable use the TTL set on store level and ignore any event-level TTL.

### Queue Capacity

The capacity of a queue defines the maximum number of events that 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

Each queue has an associated queue type. A Universal Messaging queue can be one of the types shown in the following table. The difference between the types lies in the type of physical storage used for each type and the performance overhead associated with each type.

| Queue Type | Description |
|---|---|
| Reliable | 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. |

| Queue Type | Description |
|---|---|
| Persistent | 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 one or more files for each queue on that realm, thus facilitating migrating whole queues to different realms.<br><br>For information about setting up a multi-file disk store for persistent channels or persistent queues, see the section "Multi-File Disk Stores" on page 282.<br><br>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 "Time to Live" (TTL) or capacity policy.<br><br>Messages purged from a Persistent queue are marked as deleted, however the store size will not be reduced immediately. If the store is backed by a single file, the store size will not be reduced until maintenance is performed on the queue using the Enterprise Manager or an Administration API call, or when auto-maintenance is triggered. Stores backed by multiple files will be cleaned up through the removal of individual files, according to the conditions described in the section "Multi-File Disk Stores" on page 282. This augments the high performance of the Universal Messaging realm. |
| Mixed | Mixed queues allow 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 queue, the TTL and Capacity can be set, and if the user supplies a TTL for an event this is used instead of the queue. Events stored persistently are either stored on a single file or on multiple files in the same manner as a Persistent queue. |

## Multi-File Disk Stores

Persistent events on channels and queues are stored in a file-based store that uses one or more files for each channel and queue on that realm. See the sections "Channel Attributes" on page 272 and "Queue Attributes" on page 281 for related information about the channel/queue type *Persistent*.

If you set the value of the storage property `Events Per Spindle` for any channel or queue to 0, the store will persist all events for that channel or queue in a single disk file. If you set this storage property to a value greater than 0, the events for the channel or queue will be persisted in multiple files, also known as *spindles*. The maximum number of events that can be stored in a spindle is given by `Events Per Spindle`. When the capacity of a spindle is reached, a new spindle will be created. Each spindle has the file type `mem`.

See the description of `Events Per Spindle` in the section "Storage Properties of Channels and Queues" on page 277 for related information.

**Note:**
It is possible that events in a spindle get flagged as "purged" (for example, if an event's time-to-live setting has been reached and the event is therefore no longer required). This can happen while there is still capacity to append new events at the end of the spindle, or even after

the spindle capacity has been reached. However, space for purged events in a spindle is never freed up for storing new events; new events are only ever appended to the end of the most recent, unfilled spindle. Thus, a spindle is considered full, i.e. its capacity is reached, when the total number of purged and non-purged events in the spindle is equal to `Events Per Spindle`.

Having multiple spindles allows you to optimize the use of disk space by being able to delete individual spindles if they are no longer required. An individual spindle will be deleted only if BOTH of the following conditions are met:

- When the spindle is full, i.e. when the total number of purged and non-purged events defined by the `Events Per Spindle` property are stored in it. Before the spindle gets full it cannot be deleted and a new spindle will not be created.
- When all events from the spindle have been flagged as "purged". If there are non-purged events in the spindle, it will not be deleted or resized.

**Note:**
For stores of type Mixed, when only non-persistent events are used, new spindles will still be created; however, they will not contain the original events but instead just some metadata for each event.

## Sizing Recommendation

Depending of the size of the events in a channel or queue, the value of `Events Per Spindle` should be adjusted properly. A smaller setting for `Events Per Spindle` is recommended for big events and a bigger setting for `Events Per Spindle` is recommended for small events.

As a general guideline, we suggest you use a spindle size no bigger than 1GB, and ensure that number of events stored per spindle is at least 1000. If the spindle files are too big, this can affect disk usage. Similarly, if the spindle files are too small, this can affect read/write performance.

### Examples

Here are some examples for recommended values of events per spindle, based on event size:

- small events (<1KB per event): 100 000 events per spindle.
- medium events (>= 1KB and < 20KB per event): 50 000 events per spindle. This is the default value.
- large events (>= 20KB and =< 100KB per event): 10 000 events per spindle.
- very large events (>= 1MB per event): 1000 events per spindle (though you need to consider that 1000 events at 1MB each are going to require 1GB on disk).

The above values are just recommendations and using values different from the above will not break the system.

**Note:**

To override the default value of `Events Per Spindle` (currently 50 000), the following system Java property can be set on the client side:

```
com.softwareag.um.client.store.EventPerSpindle
```

## How to Calculate File Descriptor and Mapped Memory Requirements per Store

Calculating the minimum reqirements for file descriptors and mapped memory regions per store enables you to better control and allocate resources in your system. On Linux, you use the `vm.max_map_count` and `ulimits` parameters to do so. For more information about these parameters, see "Tuning the Linux Operating System" on page 123.

When you create a store of type Persistent, Universal Messaging creates a *store_uniqueID* folder in the *Software AG_directory* \UniversalMessaging\server*instance_name*\data directory. Each folder contains two types of mapped memory files:

- state.mem. A single file per store. Contains one file descriptor and one mapped memory region.
- *spindle*.mem. The number of spindle files depends on the `Events Per Spindle` property configured for the store. Contains two file descriptors, one for read operations and one for write operations, and one mapped memory region.

In addition, for persistent queues and channels with durables (shared and serial), Universal Messaging creates a *store_name*_durable folder that contains the following types of mapped memory files:

- *.idx. The number of index pages depends on the number of published events on the store. Contains one file descriptor and one memory mapped buffer per 4 096 events.
- *.txn. The number of transaction pages depends on the number of published events on the store. Contains one file descriptor and one memory mapped buffer per 4 096 events.
- *.nsb. Only for durables. Contains a single file descriptor and one memory mapped region per durable.

To calculate the requirements for file descriptors and mapped memory regions, you also consider the number of published events on the store, the events per spindle, the number of spindles, and in case of a channel with durables, the number of durables.

### Example: Simple Channel Store

You have a simple channel with the following parameters:

Event count (N): 100 000

Events per spindle (S): 10

Number of spindles: 10 000

Inside the *store_uniqueID* folder:

- 1 state.mem file containing:

1 file descriptor

1 mapped memory region (32 byte)

- 10000 *spindle*.mem files containing:

  2 file descriptors per file or 20 000 in total

  1 mapped memory region (S * 16 byte) per file or 10 000 in total

As a result, the total for minimum file descriptors and mapped memory regions for the channel is:

Minimum file descriptors: 1 + 20 000 = 20 001

Minimum mapped memory regions: 1 + 10 000 = 10 001

### Example: Queue

You have a queue with the following parameters:

Event count (N): 100 000

Events per spindle (S): 10

Number of spindles: 10 000

Inside the *store_uniqueID* folder:

- 1 state.mem file containing:

  1 file descriptor

  1 mapped memory region (32 byte)

- 10000 *spindle*.mem files containing:

  2 file descriptors per file or 20 000 in total

  1 mapped memory region (S * 16 byte) per file or 10 000 in total

Inside the *queue_name*_durable folder:

- *.idx file containing:

  1 file descriptor per file or 25 in total (N/4 096)

  1 mapped memory buffer per 4096 events or 25 in total (N/4 096)

- *.txn file containing:

  1 file descriptor per file or 25 in total (N/4 096)

  1 mapped memory buffer per 4,096 events or 25 in total (N/4 096)

As a result, the total for minimum file descriptors and mapped memory regions for the queue is:

Minimum file descriptors: 1 + 20 000 + 25 + 25 = 20 051

Minimum mapped memory regions: 1 + 10 000 +25 + 25 = 10 051

### Example: Channel with Durables

You have a channel with durables with the following parameters:

Event count (N): 100 000

Events per spindle (S): 10

Number of spindles: 10 000

Number of durables (D): 10

Inside the *store_uniqueID* folder:

- 1 state.mem file containing:
  - 1 file descriptor
  - 1 mapped memory region (32 byte)
- 10000 *spindle*.mem files containing:
  - 2 file descriptors per file or 20 000 in total
  - 1 mapped memory region (S * 16 byte) per file or 10 000 in total

Inside the *store_name*_durable folder:

- *.nsb file containing:
  - 1 file descriptor
  - 1 mapped memory region per durable or 10 in total
- *.idx file containing:
  - 1 file descriptor per file or 250 in total: (N/4 096) * D
  - 1 mapped memory buffer per 4 096 events or 250 in total: (N/4 096) * D
- *.txn file containing:
  - 1 file descriptor per file or 250 in total: (N/4 096) * D
  - 1 mapped memory buffer per 4 096 events or 250 in total: (N/4 096) * D

The formulas to calculate minimum file descriptors and mapped memory regions for a channel with durables are:

Minimum file descriptors: 1 + 2 * (N/S) + 1 + (N/4 096) * D + ( N/4 096) * D

Minimum mapped memory regions: 1 + N/S + D + (N/4 096) * D + ( N/4 096) * D

As a result, the total for minimum file descriptors and mapped memory regions for the channel in the example is:

Minimum file descriptors: 1 + 20 000 + 1 + 250 + 250 = 20 502

Minimum mapped memory regions: 1 + 10 000 + 10 + 250 + 250 = 10 511

## Cache Recommendation

By default, caching is not enabled for multi-file disk stores. If you want to store events from a multi-file store in the cache, set the `EnableStoreCaching` and `EnableCaching` realm configuration properties to `true` and restart the Universal Messaging server instance.

When creating a channel with a multi-file store, set the **Cache on Reload** and **Enable Caching** channel storage properties to `true`. For details about the server configuration properties, see the section "Realm Configuration" in the *Administration Guide*. For details about the channel storage properties, see "Storage Properties of Channels and Queues" on page 277.

All multi-file disk stores use a default cache of size 10 000 events.

To cater for the publish/subscribe rates, available heap size, and individual event sizes in your environment, the cache size can be changed with the system property `CACHE_SIZE`, as described in the section "Server Parameters" on page 46.

For example, when your publishers are fast and subscribers are slow, the event size is small and you have more than 10 000 events awaiting delivery, then subscribers might not be accessing cached events and are instead loading the events from disk. In this case you can increase the cache size in order to achieve a possible performance gain.

## Multi-file Disk Stores do not Use Manual or Auto-Maintenance

If you use a multi-file disk store, then the realm configuration properties for auto-maintenance have no effect, since these properties are only used if you are using a single-file disk store. Also, if you manually start the "perform maintenance" operation on a multi-file disk store, this will not have any effect on the store. (The "perform maintenance" operation is available in the Enterprise Manager when you select a channel or queue, then right-click to display the context menu.)

If you use a single-file disk store, then the auto-maintenance properties will apply.

See the description of the auto-maintenance properties in the configuration group "Event Storage" in the section *Realm Configuration* of the *Administration Guide*.

The manual "perform maintenance" operation can only be used to clean up single-file disk stores, not multi-file disk stores.

# Events

## Introduction to Events

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

Events can contain simple byte array data, or more complex data structures such as an Event Dictionary (see "Event Dictionaries" on page 295).

## 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 "Event Filtering using Selectors" on page 289).

### 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 '_' wildcard character to denote exactly one character
                   or the "%" wildcard character to denote zero or more characters.
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_'
```

```
country LIKE 'Can%'
```

```
country NOT LIKE '%nada'
```

```
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.

## Event Filtering using 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 your 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 an 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 provide a filter string of the following form if we know that at position 0, the first 2 bytes are a string that represents a value you wish to filter on:

```
EVENTDATA.AS-STRING(0, 2) = 'UK'
```

You can use the LIKE operator in the filter expression, for example:

```
EVENTDATA.AS-STRING(0, 2) LIKE 'U_'
```

### 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 a geographical reference point defined by you (a point on the earth's surface as defined by the latitude and longitude values) and the geographical location of the events contained in the event stream that you are filtering. The filter will find all events whose distance from the geographical reference point matches the given comparison criterion.

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(Latitude, Longitude, Units)  Operator  Value
```

where :

- Latitude : the floating point value representing the latitude of your geographical reference point
- Longitude : the floating point value representing the longitude of your geographical reference point
- Units : an optional string indicating the unit of measurement to be used in the calculation
  - K: Kilometers < Default >
  - M : Miles

- N : Nautical Miles

- Operator: a numerical comparison operator, such as <, >, =
- Value: The required distance of the event's location from the geographical reference point.

For example :

```
DISTANCE ( 51.50, 0.16, M ) < 100
```

This will deliver events that correspond to tornadoes that are less than 100 miles away from the geographical reference point given by the latitude and longitude values (51.50, 0.16).

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.

## Event Filtering using Subjects and Principals

Universal Messaging provides several mechanisms for directing events to specific consumers:

- *Subject-based filtering*: This enables publishers to direct events to specific consumers based on the 'subject' of their connection.
- *Principal-based filtering*: This allows one or more string attributes called principals to be associated with your nSession at initialization. Publishers can also publish events with one or more principals, and your nSession will receive events that have at least one principal that matches one of your nSession's principals.

### Subject-based Filtering

Universal Messaging's subject-based filtering enables publishers to direct events to specific consumers based on the 'subject' of their connection. For example, if you initiate a Universal Messaging nSession with the subject / session name of "mrsmith", and your nSession is then used to subscribe to a channel called '/deals', any publisher on the '/deals' channel can publish an nConsumeEvent onto the '/deals' channel directly to "mrsmith" by calling the following:

```
nConsumeEvent newdeal = new nConsumeEvent("New Deal",
   "Free Holiday In Barbados".getBytes();
newdeal.setSubscriberName("mrsmith".getBytes());
channel.publish(newdeal);
```

This is a powerful concept and one which is used heavily in request/response scenarios, as well as in scenarios for directly streaming events to individuals as required.

### Limitations of subject-based filtering

Subject-based filtering has some limitations. If you consider large deployments of Universal Messaging where potentially there are multiple instances of 'services' which connect to Universal Messaging, and where the services would typically use the same UID for the nSession subject, it is difficult to distinguish between specific instances of a service, or have any kind of logical grouping of these services, as they will all appear as the same connected UID to the Universal Messaging server. One example of this would be the use of the Universal Messaging API by webMethods

Integration Server (IS), where there could be multiple instances of IS, and within each instance of IS, multiple triggers that consume events from the same channel or queue. In such a case, there would be no possibility to distinguish between specific instances of IS, or specific triggers, or all triggers on an IS instance, since they are all running using the same UID.

For this purpose, Universal Messaging provides a client API feature called *Principals* to extend the subject-based filtering mechanism. This feature is described below.

### Principal-based Filtering

All Universal Messaging Enterprise client APIs support the concept of an nSession object, which is the container for the physical connection to a Universal Messaging server. During initialisation, a handshake mechanism establishes a subject, which defaults to the local user name on the host machine (typically using the format "user@host"), but which can also be overridden to contain any arbitrary string as the UID portion of the subject.

Universal Messaging allows you to treat the subject not just as a single string but as a set of substrings called principals. These in turn can be used as filterable attributes for the nSession.

The Universal Messaging server maintains the subject in its own server-side connection that corresponds to the client nSession, and so any subject and principal list is available at the Universal Messaging server.

The nSession includes a method that enables multiple string principals to be passed during authentication. In addition, the subject filtering mechanism in the Universal Messaging server also checks the list of provided principals during the filtering based on the subscriber name(s) passed in any published events.

So you can do the following:

```
String[] group1 = {"trigger1","IS1"};
nSessionAttributes sAttr1 = new nSessionAttributes(rname);
nSession subSession1 = nSessionFactory.create(sAttr1, "subscriber1");
subSession1.setAdditionalPrincipals(group1);
subSession1.init();
String[] group2 = {"trigger2","IS1"};
nSessionAttributes sAttr2 = new nSessionAttributes(rname);
nSession subSession2 = nSessionFactory.create(sAttr1, "subscriber2");
subSession2.setAdditionalPrincipals(group2);
subSession2.init();
```

The nSessions created above now contain several principals. Similarly, a publisher can publish events in the following way:

```
nConsumeEvent event = new nConsumeEvent(group1[0].getBytes(),
   group1[0].getBytes());
event.setSubscriberName(group1[0]);
pubchan.publish(event);
nConsumeEvent event2 = new nConsumeEvent(group1[1].getBytes(),
   group1[1].getBytes());
event2.setSubscriberName(group1[1]);
pubchan.publish(event2);
```

You can see that the first event should go only to subSession1 / subscriber1, since the subscriberName equals the principal "trigger1", and the second will go to both subSession1 and subSession2, since they both have been created with a principal list that contains "IS1".

One extension to this is the ability to send an event to multiple subscriber names, which means that published events can also be delivered to multiple sessions with a matching principal in their list. For example, consider the following additional code:

```
String[] group3 = {"trigger3","IS2"};
String[] group4 = {"trigger4","IS2"};
String[] group5 = {"trigger1","trigger2","trigger3","trigger4"};
nSessionAttributes sAttr3 = new nSessionAttributes(rname);
nSession subSession3 = nSessionFactory.create(sAttr3, "subscriber3");
subSession3.setAdditionalPrincipals(group3);
subSession3.init();nSessionAttributes sAttr4 =
    new nSessionAttributes(rname2);
nSession subSession4 = nSessionFactory.create(sAttr4, "subscriber4");
subSession4.setAdditionalPrincipals(group4);
subSession4.init();
```

So a publisher can now publish the following events:

```
nConsumeEvent event3 = new nConsumeEvent("All", "All".getBytes());
event3.setSubscriberNames(group5);
pubchan.publish(event3);
nConsumeEvent event4 = new nConsumeEvent("IS1 and IS2",
   "IS1 and IS2".getBytes());
event4.setSubscriberNames(new String[]{group1[1], group3[1]});
pubchan.publish(event4);
```

Here, event3 will be delivered to all 4 sessions, since all 4 'triggerX' principal names have been added to the event, and event4 will also be delivered to all 4 sessions, since both 'IS1' and 'IS2' are added to the subscriber names. Therefore, all sessions have a matching principal for both events.

This very simple concept greatly enhances the ability to direct events to single connections or to a logical grouping of connections.

## Event Dictionaries

Universal Messaging Event Dictionaries (nEventProperties) provide an accessible and flexible way to store any number of message properties for delivery within an event (for related information, see "Events" on page 288).

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

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

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

# Behavior of Transient Events

Events marked as transient are never stored on a channel. After they are read, Universal Messaging attempts to deliver them to any consumer that is present. If there are no consumers the events are discarded.

Due to their transient nature, such events are not supported by all subscription types.

### Behavior on channels

- Non-durable subscriptions on a channel support transient events.
- Exclusive durable subscriptions on a channel support transient events.
- Shared and Serial durable subscriptions on a channel do not support transient events because transient events are not stored. Consumers connected to durable subscriptions of these types will never receive transient events.

### Behavior on queues

The transient flag on an event is ignored, and the event will be processed as a non-transient event.

### API support

Events can be marked as transient through the enterprise APIs for Java, C# and C++ .

# Event Lifecycle Logging

### Overview

*Event lifecycle logging*, also called *trace logging*, is used for creating highly detailed log files for describing the flow of events.

The generated log files are stored by default in the following locations:

- On the server: in the server's `data` directory, in a folder called `traceLogging`
- On the client : in the current directory where the client application was started from, in a folder called `clientTraceLogging`

Each store (i.e. channel or queue) for which the trace logger is enabled will have its own folder under the trace logger folder, with the same name as the store. In the store's folder, a file `trace.log` will be created, containing information for only the events that are relevant to this particular store.

If you wish to change the default folder for trace logging, set the value of the system property `com.softwareag.um.server.log.TraceLoggerPath` (on the server) or `com.softwareag.um.client.log.TraceLoggerPath` (on the client) as required. The value of the property is read only once during server/client initialization.

Log messages are not directly flushed into the trace log files for performance optimization. There is a buffer of 32KB and a scheduled task responsible for flushing the messages. The interval for flushing the messages can be configured via the system property `com.softwareag.um.log.TraceFlushInterval`. The default value is 15 seconds. This means that the trace log messages will be flushed to the trace log file when one of the following is true:

- the buffer of 32KB is full
- the trace flush interval has been reached

To enable the trace logger, some log4j dependencies must be added to the classpath of the Java application that will be traced. This is already done for the Universal Messaging server in the server's `Server_Common.conf` file. The log4j dependencies are:

- `log4j-api.jar`
- `log4j-core.jar`

### Performance Impact

Due to the nature of the feature, a drop in the performance of the server is expected when trace logging is enabled. The reason behind this is that some frequently used paths of the event flow are accessed and enriched with log messages. Below you can find the expected drop in the message throughput in different cases:

- If event lifecycle logging is disabled - no impact on performance.
- If event lifecycle logging is enabled with INFO log level:
  - Channels - up to 60% degradation
  - Channels with durables - up to 50% degradation
  - Queues - up to 50% degradation
- If event lifecycle logging is enabled with TRACE log level:
  - Channels - up to 70% degradation
  - Channels with durables - up to 60% degradation
  - Queues - up to 60% degradation

### Configuration Properties on the Server

A configuration group called **Trace Logging Config** is available on the server.

For more information on this configuration group see the section *Realm Configuration* in the Enterprise Manager documentation in the *Administration Guide*.

The trace logger can be enabled by any of the following means:

- Using the AdminAPI.

This can be done by the `nConfigGroup` object as in this example:

```
nRealmNode realmNode =
  new nRealmNode(new nSessionAttributes("nsp://localhost:9000"));

// get trace logging config group
nConfigGroup logConfGrp = realmNode.getConfigGroup( "Trace Logging Config");

// configure log level to info
logConfGrp.find("TraceStoreLogLevel").setValue(
   String.valueOf(StoreLogConfigLevel.INFO.ordinal()));
// configure the maximum size trace.log file can reach in MB
logConfGrp.find("TraceStoreLogSize").setValue(Integer.toString(10));
// configure the maximum size the trace folder can reach in MB
logConfGrp.find("TraceFolderLogSize").setValue(Integer.toString(1024));
// enable trace logger for all stores
logConfGrp.find("TraceStores").setValue("*");

// save configuration
realmNode.commitConfig(logConfGrp);
realmNode.close();
```

- Using the Enterprise Manager. There are settings in the **Realm Configuration** tab that you can set up as required.
- Using Command Central. In the Command Central UI, the **Trace Logging** configuration can be found in **General Properties** after you click the **Configuration** tab.

### Client APIs and Client System Properties

Client APIs and system properties allow the configuration of event lifecycle logging on the client.

The client APIs are located in `nConstants`:

- `setTraceFolderLogSize(int)`
- `setTraceStoreLogSize(int)`
- `setTraceStoreLogLevel(StoreLogConfigLevel)`
- `setTraceStores(String)`
- `setTraceLoggerPath(String)`

**Important:**
When this API is used to change the trace logger folder while the event lifecycle logging feature is enabled and log messages are being written to the log files, all log files present in the previous trace logger path will be zipped and new trace log files will be created in the specified folder for the stores which are enabled for trace logging.

Alternatively, you can use the following system properties on the client. They have the same effect as the methods in the `nConstants` class:

- `com.softwareag.um.client.log.TraceFolderLogSize`
- `com.softwareag.um.client.log.TraceStoreLogSize`

- com.softwareag.um.client.log.TraceStoreLogLevel
- com.softwareag.um.client.log.TraceStores
- com.softwareag.um.client.log.TraceLoggerPath

All system properties are read only once at client initialization.

**Generating a unique ID**

The additional property com.softwareag.um.client.GenerateEventUID can be used for generating a unique ID when an event is created on the client side.

When this property is set to "true", a unique ID will be generated on the client side and will be put into the nConsumeEvent tag in order to allow easier tracing of the event on the client and the server. This ID will be generated only when ALL of the following three conditions are met:

- The tag of the generated nConsumeEvent is empty, and
- The uuid property is not generated in the fEventDictionary of the nConsumeEvent, and
- The messageID is not populated in the nHeader of the nConsumeEvent.

**Example of a setup on the client**

```
nConstants.setTraceStoreLogLevel(StoreLogConfigLevel.TRACE);
nConstants.setTraceStores("*");
nConstants.setTraceLoggerPath("./clientTraceLog");
try {
     nConstants.setTraceStoreLogSize(10);
     nConstants.setTraceFolderLogSize(1024);
} catch (nIllegalArgumentException e) {
     e.printStackTrace();
}
```

## Disabling Log4J Logging in an OSGi Environment

When Universal Messaging connects to applications in an OSGi environment, you can disable the Log4J2 logging using the following property:

```
Dcom.softwareag.um.log.TraceLoggerProviderClass=com.pcbsys.foundation.logger.storelogger.NoopLoggerProvider
```

## Behavior of the Trace Log Files

A trace log is created per store when the log level of the trace logger is set to INFO or TRACE and there are stores enabled for tracing via the TraceStores configuration property. When the specified TraceStoreLogSize value is reached, the trace log file will be zipped and a new one will be created. When the size of the trace logging folder reaches the value set in the TraceFolderLogSize property, the oldest zip files will be deleted.

**Important:**

**Important**

The value of `TraceFolderLogSize` can sometimes be ignored. This will happen if the following equation is true :

(number of stores to be traced) * (`TraceStoreLogSize` value) > (`TraceFolderLogSize` value)

Let's look at the following example :

1. `TraceStoreLogSize` is set to 100MB and `TraceFolderLogSize` is set to 1024MB (1GB)
2. There are 15 stores on the server/client and trace logging is enabled for all of them

Then the value of `TraceFolderLogSize` will be ignored because 15 * 100 > 1024.

## Basic Examples

Here are some basic examples of event lifecycle trace logs presenting the format and the information that can be found in them.

**Example 1: A channel is created and one event is published to it**

On the client:

```
[2020-03-13 16:45:42,801][INFO][Created channel successfully.
   attr={type=Reliable, JMSEngine=false, TTL=0, capacity=0, ID=-1,
   spindleSize=50000, isClusterWide=false}, conn=,
   Conn={SSID:ec2200000000,Local:127.0.0.1:58153,URL:nsp://127.0.0.1:26000}]
   [nSession][Time-limited test]
[2020-03-13 16:45:43,184][INFO][Sending publish request. nPublished={EID=0,
   tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8}, conn=, Conn={SSID:ec2200000000,
  Local:127.0.0.1:58153,URL:nsp://127.0.0.1:26000}][nChannelImpl][Time-limited test]
```

On the server:

```
[2020-03-13 16:45:42,737][INFO][Store created successfully.
   attr={, type=Reliable, JMSEngine=false, TTL=0, capacity=0, ID=47045869343854696,
   spindleSize=50000, isClustered=false}, connId=127.0.0.1:58153]
   [nUserStoreCreationHandler][ReadPool:0]
[2020-03-13 16:45:43,188][TRACE][Publish request received.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8},
   conn=127.0.0.1:58153][nUserPublishHandler][ReadPool:1]
[2020-03-13 16:45:43,220][TRACE][Sending event to fanout.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8},
   conn=127.0.0.1:58153][nUserPublishHandler][ReadPool:1]
[2020-03-13 16:45:43,221][INFO][Processing in fanout.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8}]
   [nChannelEngine][ReadPool:1]
[2020-03-13 16:45:43,221][TRACE][Storing event in channel.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8}]
   [cChannelList][ReadPool:1]
[2020-03-13 16:45:43,223][INFO][Returning from fanout.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8}, count=0]
   [nChannelEngine][ReadPool:1]
```

**Example 2: A channel is obtained and an async subscriber which receives one event is added to it**

On the client:

```
[2020-03-14 15:02:40,148][TRACE][Adding asynchronous subscriber.
   isInfiniteWindow=false, windowSize=0, selector=null, maintainPriority=false,
   eid=0, messagingValidator=No,
   Conn={SSID:1d1200000000,Local:127.0.0.1:60408,URL:nsp://127.0.0.1:26000}]
   [nChannelImpl][Time-limited test]
[2020-03-14 15:02:40,149][INFO][Successful addition of asynchronous subscriber.
   requestId=54, Conn={SSID:1d1200000000,Local:127.0.0.1:60408,
   URL:nsp://127.0.0.1:26000}][nChannelImpl][Time-limited test]
[2020-03-14 15:02:40,228][TRACE][Publishing response to client.
   nConsumeEvent={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8, redelivery=0}]
   [nClientChannelList][UM-Connection-Reader:3 Active]
[2020-03-14 15:02:40,228][TRACE][Received event on client.
   nConsumeEvent={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8, redelivery=0}]
   [nClientChannelList][UM-Connection-Reader:3 Active]
[2020-03-14 15:02:40,231][INFO][Received event for asynchronous subscriber.
   nConsumeEvent={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8, redelivery=0}]
   [nClientChannelList][UM-Event-Processing-Pool: 3:0]
```

On the server:

```
[2020-03-14 15:02:40,148][INFO][Adding asynchronous subscriber.
   EID=0, selector=, requestId=54, connId=127.0.0.1:60408, sessionId=1d1200000000]
   [nUserSubscribeHandler][ReadPool:2]
[2020-03-14 15:02:40,194][INFO][Channel recovery attempting to write event to client.

   connId=127.0.0.1:60408, event={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8}]
   [nChannelRecoveryTask][Recovery Pool:1]
[2020-03-14 15:02:40,195][TRACE][Event written to subscriber.
   nPublished={EID=0, tag=dedef12f-967a-44c5-bfd2-95d4ffaf3cf8 }]
   [nChannelRecoveryTask][Recovery Pool:1]
```

## Native Communication Protocols

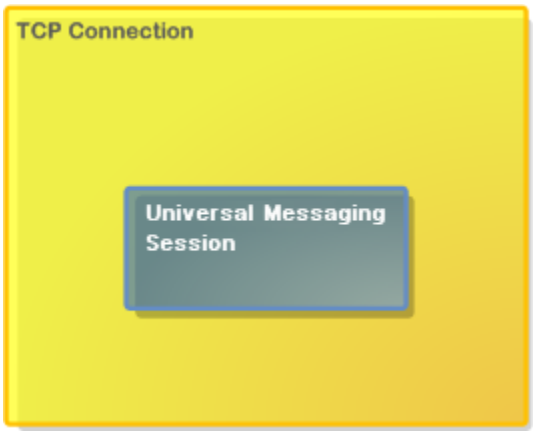
Universal Messaging supports four *Native Communication Protocols*. These TCP protocols are:

- 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)

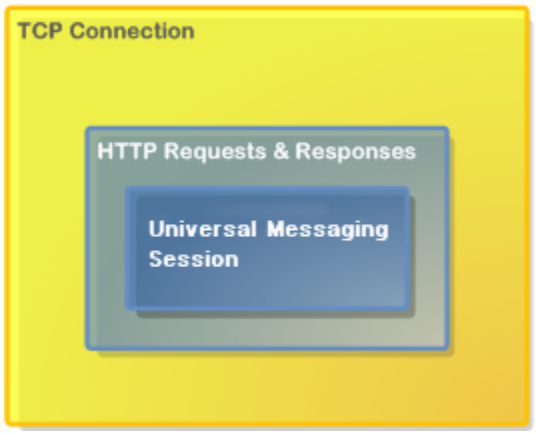
### Universal Messaging HTTP Protocol (nhp)

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.

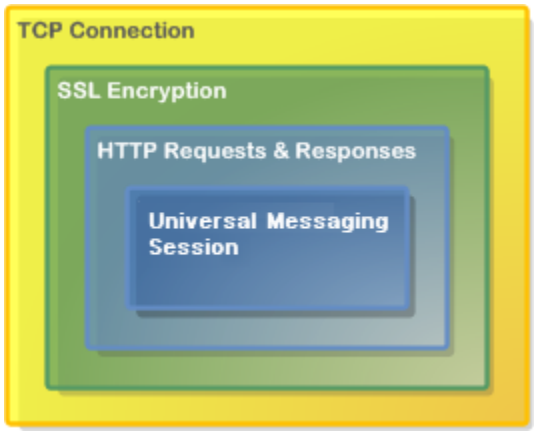



Universal Messaging HTTP Protocol (nhp)

### 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 using the *Universal Messaging HTTPS Protocol (nhps)* for its firewall traversal and security features.

When handling many inbound connections that do not require the HTTP or HTTPS protocol, we recommend using the *Universal Messaging Socket Protocol (nsp)* or the *Universal Messaging SSL Protocol (nsps)*.

### 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.

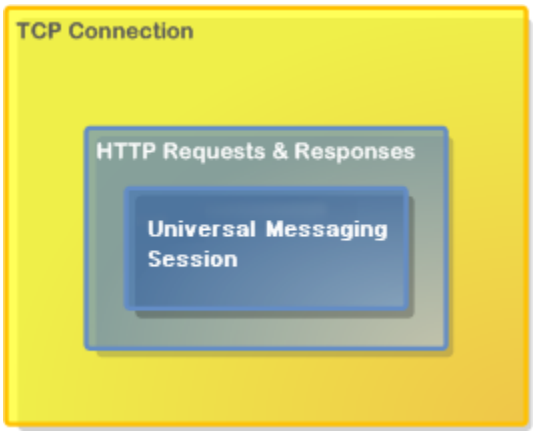
The protocol is fully SSL-encrypted and also supports the use of HTTP proxy servers, with or without proxy user authentication.

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.

## 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.

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.

Universal Messaging always uses server-side filtering of Google Protocol Buffers, and this, coupled with Google Protocol Buffer's space-efficient serialization can reduce the amount of data delivered to a client.

**Note:**
If server side filtering is not required, do not configure Google protocol buffer descriptors for the respective channel or queue.

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 protobuf descriptor file required at runtime is constructed by compiling the `.proto` file using the standard protobuf compiler and the --descriptor_set_out option.

### 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 .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 for a given channel or queue is achieved by providing the server with the protobuf descriptor file for the channel/queue. The location of the protobuf descriptor file for any given channel/queue can be configured in the attribute list of the channel/queue using the Enterprise Manager, or programmatically using the Administration API.

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.

The Enterprise Manager can be used to view, edit and republish protocol buffer events, even if the Enterprise Manager is not running on the same machine as the server.

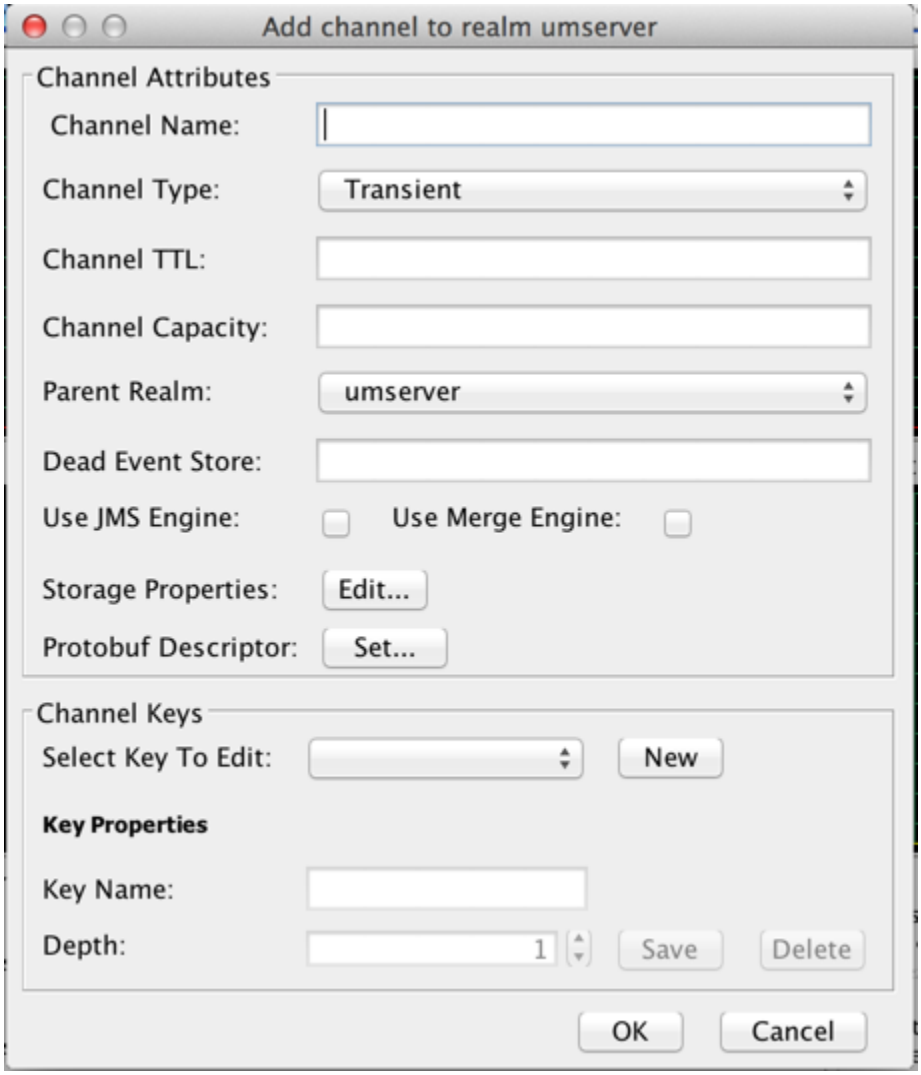
### Configuring Universal Messaging for use with Protocol Buffers

The realm configuration parameters for protocol buffers have their own section **Protobuf Config** in the Enterprise Manager **Config** panel.

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

### Protobuf with the Enterprise Manager

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




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

## 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*

## Rollback

When an event is consumed from a queue transactionally, there is a chance that the transaction will be rolled back. This could be triggered by the user or by a disconnection. When this happens, the event needs to be put onto the queue for redelivery.

Note that for performance reasons for asynchronous consumers, Universal Messaging can deliver a window of messages to the client. So if the user rolls back event 0, they will receive the window of events already on the client before they consume event 0 again. It is possible to set the client window to be only 1 event, and doing so will guarantee event ordering.

The window size is set on the `nQueueReaderContext` for queues, and on `nChannel.addSubscriber(..., windowSize)` for channels. It is only relevant for asynchronous consumers.

Using a small window size affects the performance negatively, since this increases the number of windows Universal Messaging server has to deliver. Therefore, if your application does not require rolled-back events to be ordered, we suggest leaving the default windows size unaltered.

**Note:**

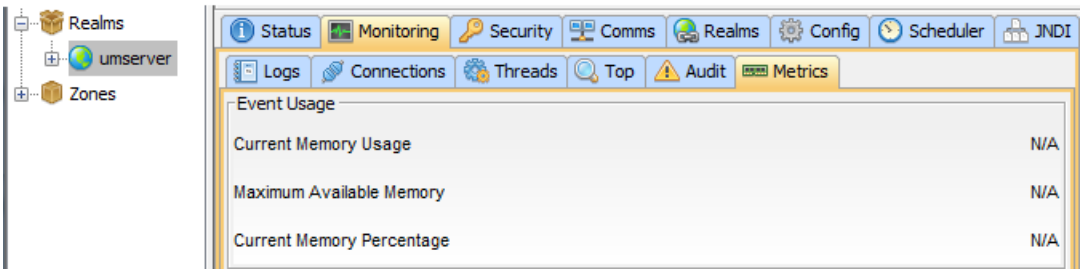
Prior to version 10.5, for queues, when rollback happens, the event is re-added to the front of the queue and is assigned a new event ID. So if an event is rolled back, that event will be added straight back at the front of the queue ready for redelivery. The API method `event.getHeader().getOriginalTransactionEID()` returns the event ID that the event had when it was first published to the queue; this value is only set on republish.

## Out-of-Memory Protection

Universal Messaging provides methods that you can use to protect from out-of-memory situations. It is important to note that a subset of these methods may impact performance; those which do so are disabled by default.

### Metric Management

Universal Messaging provides metrics pertaining to memory usage on a realm server. When you select the **Monitoring** tab in the Enterprise Manager, these metrics can be found under the **Metrics** tab, as shown below:




Metrics can be enabled/disabled on a realm server through the realm configuration. Setting the configuration option `EnableMetrics` controls the enabling/disabling of *all* metrics; each metric can also be enabled/disabled individually.

**Event Usage**

The event usage metric provides information on memory currently in use by on-heap events. The three statistics provided are as follows:

1. Current Memory Usage - the memory currently in use by on-heap events.
2. Maximum Available Memory - the maximum memory currently available to the JVM.
3. Current Memory Percentage - the percentage of the maximum memory available to the JVM currently occupied by on-heap events.

Event usage monitoring can be enabled/disabled through the configuration option `EnableEventMemoryMonitoring`. Enabling this metric may impact performance, and so it is disabled by default. Disabling this metric will cause "N/A" to be displayed for each of the three statistics described previously, as shown in the screenshot above.

## Flow Control

Universal Messaging provides the ability to hold producing connections (publishes, transactional publishes, batch publishes, etc.) before processing their events. This is to allow consumers to reduce the number of events on connections before additional events are published, thus decreasing memory usage.

Flow control can be enabled/disabled and configured on a realm server through the realm configuration. Setting the configuration option `EnableFlowControl` controls the enabling/disabling of flow control; flow control is disabled by default. There are also three other realm server configurable options:

- `FlowControlWaitTimeOne` - the first level of waiting, and shortest wait time; activated at 70% memory usage
- `FlowControlWaitTimeTwo` - the second level of waiting; activated at 80% memory usage
- `FlowControlWaitTimeThree` - the final level of waiting, and longest wait time; activated at 90% memory usage

Each of these options can be configured to suit the level of out-of-memory protection required. They represent the amount of time, in milliseconds, for which producer connections are throttled upon reaching the associated percentage of memory usage. In general, as memory usage gets

closer to 100%, the need for throttling will be greater, so you would normally set `FlowControlWaitTimeThree` to a higher value than `FlowControlWaitTimeTwo`, which in turn will be higher than `FlowControlWaitTimeOne`.

Example: Assume a current memory usage of 90%. If there is a new publish request, the connection from which the publish originates will be throttled for `FlowControlWaitTimeThree` milliseconds, ensuring the event(s) will not be published sooner than `FlowControlWaitTimeThree` milliseconds from the time of publish.

If the memory usage exceeds the value defined by the configurable option `EmergencyThreshold` (94% by default), the connection will be paused for a long duration (24 days).

**Note:**
The memory threshold values of 70%, 80% and 90% that trigger the wait times `FlowControlWaitTimeOne`, `FlowControlWaitTimeTwo` and `FlowControlWaitTimeThree` respectively are defined on the assumption that `EmergencyThreshold` is set to its default value of 94%. If the `EmergencyThreshold` value is reduced, the memory threshold values also reduce proportionally.

**Transactional Publishes**

It is important to note that when using transactional publishes, by default the timeouts are shorter than standard publishes. This means that even if a client times out, the event may still be processed if the session has not been disconnected by the keep alive. As such, when using transactional publishes, it is best practice to call publish just once before committing. Every consecutive publish will be throttled, potentially leading to a timeout during the commit, and despite this timeout, the commit may still be successful.

**Note:**
Even if flow control is enabled globally on the server, you can use Universal Messaging client API calls to bypass flow control for an individual producing connection. To do this, a boolean flag can be set on the relevant session attributes which requests that the associated connection should not be throttled. See the client API documentation on the class nSessionAttributes for more information.

# Pause Publishing

The *pause publishing* feature allows all client publishing to the server to be paused, across channels, queues and data groups. This pause does not affect the administration API, inter-cluster communication or joins.

The feature is activated by setting the server configuration property `PauseServerPublishing` to true. Then, clients trying to publish or commit will receive `nPublishPausedException`.

See the section *Realm Configuration* in the *Developer Guide* for a description of the configuration property.

The sections below describe how publish methods are affected when publishing is paused.

## JMS Publishing

**Transactional**

A commit will throw a JMSException with a message containing "Publishing paused". The root exception will be `nPublishPausedException` which inherits `nSessionPausedException`.

The publish calls before the commit will throw an asynchronous JMSException if an asynchronous exception listener is attached.

**Note:**
The first call of a publish will not throw an exception because of an optimization, whereby the first published event is in fact not sent to the server until a commit or a second publish is in place.

**Non-transactional send**

The send calls throw an asynchronous JMSException if an asynchronous exception listener is attached.

**Note:**
The first call of a publish will not throw an exception because of an optimization, whereby the first published event is in fact not sent to the server until a commit or a second publish is in place.

**Non-transactional SYNC send (syncSendPersistent = true)**

The send calls will throw a synchronous JMSException.

The root cause of the JMSException is `nPublishPausedException`, and can be checked for example with this code:

```
} catch (JMSException e) {
  Throwable rootCause = findRootCause(e);
  assertTrue(rootCause instanceof nPublishPausedException);
}
Throwable findRootCause(Throwable e){
  Throwable cause = e.getCause();
  return (cause != null ? findRootCause(cause) : e);
}
```

## Native Java Publishing

**Transactional**

`nTransaction.commit` will result in `nPublishPausedException` which inherits `nSessionPausedException`.

The publish calls before the commit will throw an asynchronous `nPublishPausedException` if an asynchronous exception listener is attached.

**Note:**
The first call of a publish will not throw an exception because of an optimization, whereby the first published event is in fact not sent to the server until a commit or a second publish is in place.

**Non transactional**

The publish calls will throw an asynchronous `nPublishPausedException` if an asynchronous exception listener is attached.

### C++/C#

The exceptions for C++ and C# are the same as for the Java native API.

### MQTT, AMQP and JavaScript

The exceptions for MQTT, AMQP and JavaScript when publishing is paused are handled as follows:

**MQTT:**

Currently, the MQTT protocol does not provide mechanisms to return proper errors to publishers. To handle publish exceptions (in particular, when publishing is paused), Universal Messaging provides two options: either ignore the exception, or disconnect the publisher's connection. This is configured using the `DisconnectClientsOnPublishFailure` MQTT configuration setting.

**AMQP:**

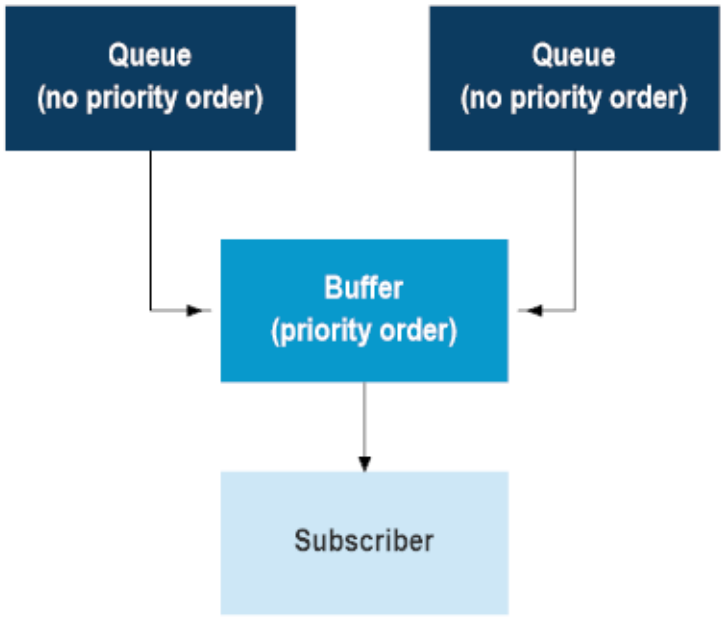
A standard AMQP error `amqp:internal-error` will be returned to the publisher client if publishing is paused. The error message starts with "Publishing paused", so adopters can use this substring to identify the actual cause

**JavaScript:**

If there is a publish or commit while publishing is paused, an error with a description starting with "Publishing paused" will be returned.

## Priority Messaging

The following diagram shows how priority messaging works in Universal Messaging.



Consider the following information:

- Message prioritization works only for asynchronous messaging.
- Message prioritization happens only in the client-side buffer of the subscriber.
- The client-side buffer of the subscriber is filled based on the window size and is filled by the server.