Xinetica White Paper

Title: **Message Oriented Middleware and MQSeries**

Content: **A discussion of middleware technology in general, and IBM MQSeries in particular.**

www.xinetica.com

Introduction

Document Scope

This document is intended to position message oriented middleware within the framework of middleware technology in general, and to provide a technical overview of IBM's MQSeries in particular. No assumption is made about the reader's prior exposure to middleware technology.

The first part of the document is largely non-technical and can be understood by people not necessarily from an IT background. The second part concentrates on IBM MQSeries in particular and includes a larger amount of technical detail. The third and final part briefly lists some relevant products currently in the marketplace, along with some useful links to other sources of information.

Whenever relevant, there are links like this one that point to external web sites containing further information on the subject currently being discussed.

Part 1: Middleware Background

What is Middleware?

Many definitions exist, many of them are more or less correct. The problem is that the word "Middleware" is somewhat overused and as a result has been applied to just about every piece of systems software ever written. Here's one definition:

"Middleware is a general term for any piece of software that serves to 'glue together', mediate between, or enhance separate existing programs."

As you can see, even that definition covers a huge range of applications. Here's another definition:

"Middleware is software that is used to move data from one program to another, shielding the developer from

dependencies on communications protocols, operating systems and hardware platforms."

For a while, attempts have been made to categorise the different types of middleware. It's not always clear what category a product fits in to as vendors are always introducing new functionality, or include support for more than one category of middleware within a product. BEA's Tuxedo for example, includes transaction processing and message queuing services via the product's API (Application Programming Interface).

Having said that, middleware products tend to fall in to one of the following categories:

Middleware Category	Product Example
Message Oriented Middleware (MOM)	IBM MQSeries, Microsoft MSMQ, BEA MessageQ, Talarian SmartSockets Momentum's X-IPC, Level 8's Falcon MQ (a Unix port of MSMQ)
Data Connectivity	ODBC, Vendor specific products
Remote Procedure Calls (RPC's)	The OSF Distributed Computing Environment (DCE), Sun Microsystems NFS (Network File System)
Object Request Brokers (ORB's)	Traditionally associated with the CORBA (Common Object Request Broker Architecture) standard. Iona's Orbix, Visigenic's Visibroker, BEA Objectbroker, Java RMI (Remote Method Invocation)
Transaction Monitors	Microsoft Transaction Server (MTS), IBM CICS, IBM Encina, BEA Tuxedo.

The important thing to note is that each type of middleware has been designed to accomplish a specific task. The choice of middleware therefore depends on the business requirement that is being satisfied by deploying the technology. Many organisations employ more than one middleware technology. There is no "one size fits all" solution.

Synchronous versus Asynchronous Communication

Whilst there are different types of middleware as detailed above, they can all support one, or sometimes two, modes of operation. The modes are: synchronous or time dependent; and asynchronous, or time independent. Put simply, in a synchronous environment, at least two parties have to be present at the same time in order for communication to take place, rather like a telephone call. In an asynchronous environment, only one party has to be present, rather like an email. Synchronous and asynchronous modes are sometimes referred to as "connection oriented" and "connectionless" modes respectively.

Just to put this into context, it's necessary to jump ahead a little. MOM works primarily in an asynchronous (connectionless) mode, which means that for an application to send a message, the consumer of that message does not *have* to be available. Similarly when the consumer eventually reads the message (at an indeterminate point in the future) the producer of the message does not have to be available. Of course the designer of an application can "force" a pseudo-synchronous mode by insisting that the consumer sends an acknowledgement message back to the producer. The producer will then wait for the acknowledgement message before continuing processing. There's a fuller description of MOM in the next section.

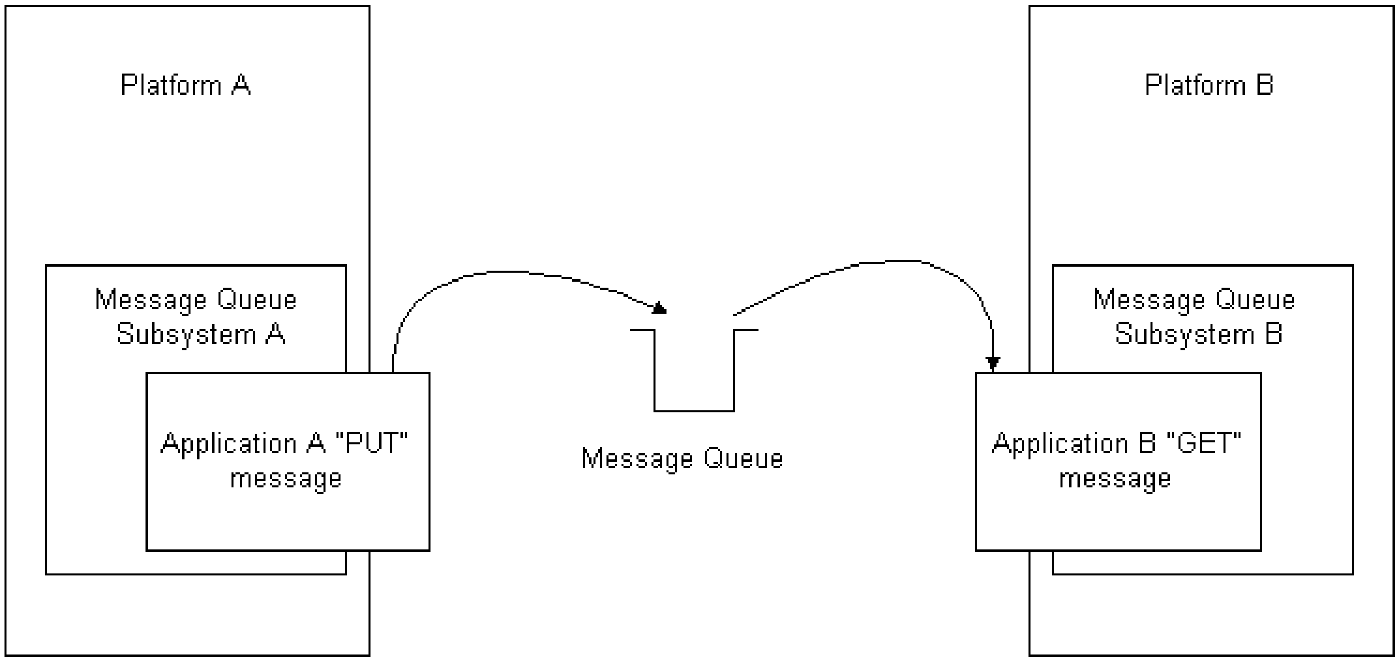
The following table shows the various combinations of modes of operation. Transaction Monitors are almost always synchronous, although as mentioned earlier, products like Tuxedo bend the rules a bit! Also note again that MOM products, whilst asynchronous in nature, can be *made* to work in a pseudo-synchronous environment.

Middleware Category	Synchronous	Asynchronous
MOM		X
Data Connectivity	X	
RPC	X	
ORB	X	
Transaction Monitors	X	

As the main aim of this document is to describe the features and benefits of MQSeries, there will be no further discussion of other types of middleware.

What is Message Oriented Middleware?

It's already been suggested that MOM is middleware that allows an application to send a message to another application without the other application necessarily being available. That's it in a nutshell, however a picture showing the relevant components may give a better perspective.


Platform A
Platform B
Message Queue Subsystem A
Message Queue Subsystem B
Application A "PUT" message
Application B "GET" message
Message Queue

This picture shows the logical world from the application's point of view. It's just one example of a common messaging scenario. Messages are placed on a queue by an application and retrieved by another application. The implication in this diagram is that the physical location of the queue is not known to either application. Similarly, the physical details of the host platform are not known. All that is required is that an application is in some way registered or connected to the message queue subsystem. The "cleverness" is contained in that subsystem and therefore shields the applications from the physical world. (This matches closely with one of the definitions of middleware suggested at the start of this document) It also provides a useful abstraction that enables physical implementations to be changed on either platform, without affecting the rest of the implementation.

Why Do I Need Middleware?

There are benefits to utilising middleware technology that are realised on many different levels. It depends who you are within an organisation. Bearing in mind what we know so far about middleware in general, and MOM in particular, lets try to imagine how different people in an organisation would see those benefits.

The second part of the document will then concentrate on how MQSeries works in particular.

Board Level Director

Your primary interests are business-related. You probably have significant investment in existing systems. Those systems provide business-critical functions, and will probably continue to do so for the foreseeable future. Unfortunately your competitors are all embracing the Internet, and have managed to "expose" their systems to the Internet and are now transacting business over it. Understandably, you want to do the same. Your IT manager informs you that this is possible, using the right technology. However, you are more interested in what is possible (and at what cost) rather than how it is achieved.

IT Manager

The IT Manager has a difficult job these days. Never before have organisations had so many different pieces of hardware and software that were never designed to work together. The vision of a single, unified set of software and hardware looks further away from reality than ever before. Now the demand to produce working systems from within a heterogeneous environment places the emphasis on re-using what you've got, rather than rewriting from scratch each time a new system is designed. The Board are increasingly nervous about spending money on new systems when the business requirements are already being serviced through existing applications that have taken many years to evolve.

There is hope though. Your team of Architects tell you that middleware technology can be used to promote re-use by enabling the definition of interfaces between systems, or between individual business processes across systems. And you don't have to throw things away.

Technical Architect

The message from senior management these days is "re-use". This makes good sense to you. There are sound architectural reasons for pursuing software re-use. The main reasons are that re-using bits of systems reduces overall complexity, delivery times and project costs.

To achieve a decent level of re-use, you should pursue component based development (CBD). CDB relies *heavily* on middleware technology. How do you do CBD? Well, CBD is a subject unto itself, but here's a fly-by.

- Identify components. In order for applications to be re-used, the useful bits of them have to be identified and turned into components. A component is something that services a business request, like “Get me this customers account history”, or “send this customer a reminder letter”, and so on. This component identification, which will be influenced by the level of re-use you want to achieve, is the main challenge for the Architect.
- Define and publish the component interfaces. A component-based architecture relies upon each component having a known interface. This interface is then useable by any other “consumer” (which may in turn be another component). In this way systems can be assembled from components rather than being duplicated.
- Use the interface, the whole interface, and nothing but the interface. That is, a user (consumer) of a component service only has to know how to invoke a component through it’s interface. The implementation details of that component are never known to the consumer. The corollary of this is that you can change a component’s implementation *without* having to change any of the consumer applications. That saves time and money.

The whole concept of CDB is very sound, but it hinges on the fact that components need interface to each other in a platform-independent way. That where the choice of middleware(s) come in, a represents a very useful tool in the architectural armoury. A component can be now be used by any other system in the enterprise.

Applications Developer

Architects have big ideas, but they seldom have to turn them into reality. That’s the job of the Developer. Now to build your system, your Systems Analyst tells you that you need data from a legacy database that lives on a platform you know nothing about, on a network running a protocol you’ve never heard of. Normally, this would be a big problem. However, a member of the legacy system team in question has already “middleware enabled” the parts of that application that provide that vital data. You can thank the Architect for spotting that, and for the choice of (in this case) message oriented middleware. All you have to do is ask for the data. To do this, you write some code that puts a message on a queue. Then you wait for a reply (or have some other process that handles replies). That’s it. You don’t care how it works, it just does. And you can carry on building your system without duplicating the effort that went in to producing the legacy system in the first place. In short, you really are re-using software components. If you are curious and you want to see a map of all the components you can use for other things, speak to your Architect.

Part 2: MQSeries Message Oriented Middleware

Background

Now that it is clearer where MOM fits in to the overall middleware picture, the remainder of this document will concentrate on IBM’s MQSeries (MQ) specifically. MQ Provides a multi-platform, robust, assured delivery application to application messaging infrastructure. It comes with a vast array of functionality but can be extended further by the addition of user-written “exits”. An exit is a piece of code that is executed by the queue manager upon certain events. Those events include; the placing of a message on a queue, the starting of a communications channel, the exchange of security information with a remote queue manager. Many exits are provided as add-on “SupportPacs”. IBM’s MQSeries SupportPac web site URL is given in the “links” table at the end of this document.

MQ forms the basis of a family of products that include:

- MQSeries Integrator – a message broker with data transformation and mapping functions
- MQSeries Workflow - Model-driven e-business process automation and tracking
- MQSeries Everyplace – Allow mobile workers with laptops, phones and PDA’s to participate in MQ networks.

Platform Support

MQ runs on every hardware platform and O/S that IBM produces, as well as NT, HP/UX, Solaris, Linux, Tandem NSK and more. IBM maintain a full list on their web site.

Network protocol support includes TCP/IP, IPX and SNA (LU6.2). This means that MQ applications can exchange data over disparate networks. Building on the idea of abstraction again, it’s worth restating that the configuration of network details is all managed by the administrator of the MQ network and is contained within MQ, *not* within the applications that want to participate in the MQ network.

MQSeries Objects

In order to understand how MQ works, it is necessary to appreciate some of the objects that go to make up a working MQ installation. The word “object” in this context is not used in an Object Oriented sense. It merely refers to the “things” that have to be set up after the product is installed. After a brief description of each object, a picture will bring together all of the objects based upon the earlier conceptual diagram.

Queue Manager

The queue manager represents a physical instance of the messaging subsystem on a server. For an application to use MQ, it must first connect to a queue manager. The queue manager is created on the command line using the “crtmqm” command and is given a name by it’s creator. During the creation process, a number of default objects are created. These default objects are used by MQ for it’s own purposes during the life of the queue manager instance, as well as “model” objects than can be used as templates for creating further objects later on.

Once a queue manager is created, it can be started via the “strmqm” command, and stopped using the “endmqm” command. It is possible to run more than one queue manager on the same machine, although this often causes confusion when testing applications. Once it is started, administrative commands can be issued using the “runmqsc” command. MQ employs a scripting language to describe other objects that can be manipulated. The following objects are all created using runmqsc script.

Queue

The most frequently used object type. A queue can be defined using a rich set of attributes, far too many to describe within the scope of this document. Suffice to say queue attributes include message length, maximum queue depth, and default message persistence (decides if a message on a queue can survive a queue manager restarting)

Queues can be defined as *local* in which case they reside on the local machine; or as *remote*, in which case they exist on a remote machine with the definition of the remote queue “pointing” to the actual location on another queue manager. If an application puts a message on a remote queue, it is immediately placed on a “transmit queue” ready for transmission over a channel to a remote queue manager.

Finally, an application may create a queue dynamically. Typically this is used to accommodate replies to request messages, and once the reply message is read, the queue is closed and destroyed.

Channel

In order for messages to be transmitted to remote queues managed by remote queue managers, a channel between two queue managers must be defined. This usually involves some scripting on both queue managers although it is also possible to dynamically create a sender channel on request.

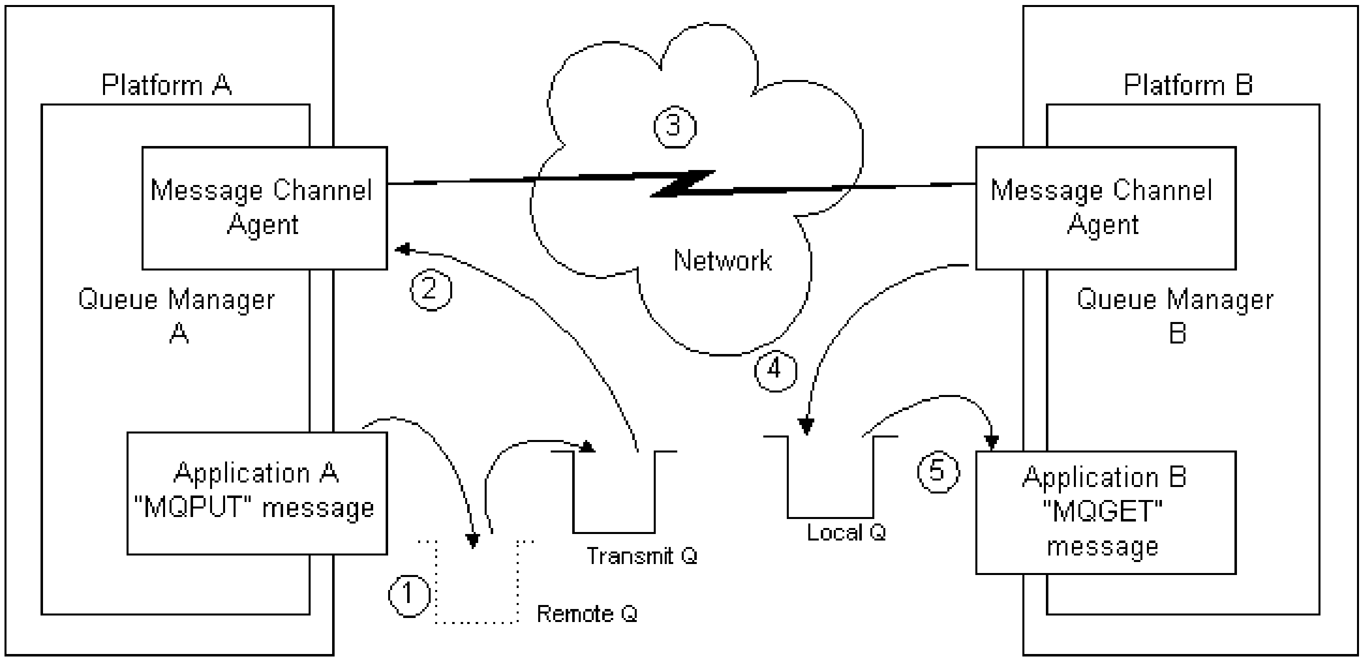
Channels are uni-directional. In other words a channel can be used for sending or receiving, but not both. It’s therefore common to have a pair of channels defined between two queue managers – one for sending and one for receiving.

It may be the case that a remote queue exists on a queue manager that is more than one “hop” away. It is possible to configure remote queues so that they are routed through any number of intermediate queue managers.

Here’s an example of how this would be useful. A queue manager on an SNA network manages a particular queue. Another queue manager on a TCP/IP network has a remote queue definition that points to the SNA queue manager, however because of the different network protocols in place, there is no direct route between the two networks. Enter a third machine, running SNA *and* TCP/IP, and is therefore visible to both networks. This third queue manager has two pairs of channel definitions, one pair to the TCP/IP queue manager, and the other pair to the SNA queue manager. The third queue manager can now act as a gateway for messages between the SNA and TCP/IP queue managers.

Again, all of this is in the configuration of each queue manager, not within any application that uses MQ.

Here’s a picture based upon the earlier conceptual view, fleshed out with some MQ physical detail.



An explanation is appropriate. As you can see each step in the process is numbered.

(1) An MQ application, connected to queue manager A, puts a message on a queue. It so happens that this queue is not

actually held locally, but is a definition of a remote queue managed by another queue manager, queue manager B, on a different box. The message is therefore immediately placed on a transmit queue.

(2) A component of the queue manager known as the Message Channel Agent (MCA) gets the message from the transmit queue. Note that the MCA is actually an MQ application, with built-in communications functionality. The MCA is controlled by the channel definitions given by the MQ administrator and represents the physical instance of a started channel.

(3) The two MCA's negotiate with each other, and the MQ "channel" is started. Channels can be configured to be open all the time, or to start when a message arrives on a transmit queue (as in this example).

(4) The receiving MCA inspects the message and determines where to place it. At this point, if the message details indicate that the message is destined for a different queue manager, it would be placed on another transmit queue, and another channel to that queue manager would be negotiated.

(5) The message is now available on the local queue, ready for application B to process.

Client Connections

In the example given above, it is assumed that the application physically resides on the same machine as the queue manager. The does not have to be the case, as applications can be configured to run as MQ clients instead. In this case the application is not coded any differently than before. The queue manager and client do not have to be the same platform type either. For example it is common to see a queue manager running on a Unix server, with client applications running on Windows 98 PC's.

When the client application makes a connection to the queue manager, a special kind of channel, a client channel, is negotiated. This channel stays active for as long as the client application remains connected to the queue manager.

Support for Different Messaging Models

Datagram (fire & forget)

A datagram message is a message placed on a queue, for which no reply is expected.

Request/Reply

An application places a message on a queue and expects a reply in return. This does not necessarily imply synchronous operation, as the process receiving the reply may be separate from the requester.

Publish & Subscribe

In this model messages are "broadcast" to interested participating applications in the MQ network. There are a number of elements within a publish & subscribe model.

1)The **publisher**. Publishers supply information about a number of topics. A topic can be anything the designer of the system wants, say stock market prices or seating arrangements for a theatre.

2)The **broker**. The publisher sends MQSeries messages containing topical data to a Publish & Subscribe broker (installed on a queue manager), which then forwards it to other brokers in the network.

3)The **subscriber** registers an interest in a number of topics. Brokers then send MQSeries messages to subscribers containing data on the subscribers registered topics. It is possible for many publishers to publish the same topic (for example, publishers for both NASDAQ and London Stock Exchange prices) and subscribers can register interest in any number of topics.

The publish and subscribe model can be quite sophisticated. Writing pub/sub applications can therefore be complicated, so IBM have provided another API, the AMI (Application Messaging Interface) for C, C++ and Java. This greatly simplifies the creation of publishing and subscribing applications. Much of the configuration of the pub/sub environment is held in an external repository which is referenced by the participating applications.

Queue Manager Clusters

Queue managers can be logically grouped together to form clusters. Each queue manager may be on a different platform type (NT, AIX, Solaris) and may be physically remote from each other. Cluster queue managers host cluster queues, which are advertised to other queue managers in the cluster. It is possible to have the same queue logically shared by more than one queue manager, thus providing a level of high availability. There is also a benefit in terms of reduced systems administration, as clustered queues do not have to have a definition within each queue manager in the cluster.

Queue manager clusters can be quite sophisticated, and individual clusters may logically "overlap" one another. A repository of cluster information is held on at least two queue managers within the cluster.

For a detailed description, have a look at the IBM MQSeries clustering documentation.

XA Compliance

The Open Group's XA interface standardises the relationship between a *transaction manager* and a *resource manager*. This allows applications to group operations into a single unit of work that will be committed only when all operations have completed successfully. Any failure will result in *all* of the operations in the unit of work being backed-out. This enables applications to maintain a guaranteed level of integrity in the event of an application failure.

An MQSeries queue manager is XA compliant. This means it can act as a *resource manager* in a distributed unit of work managed by an external *transaction manager* such as Encina or Tuxedo. This gives the application designer the freedom to use whatever tools are appropriate for the task. Other resource managers include Oracle, Informix, Sybase, DB/2 and SQL Server.

For example, it is possible for an application to have a unit of work that does an SQL INSERT, then put a message on a queue. If any of the statements fail, the messages will be rolled back off the queue, along with the database change, thus ensuring the "all or nothing" integrity of the unit of work. The changes are only committed when all operations have been successful. This kind of distributed transaction processing is often vital in enterprise level systems.

Web Development

MQ is becoming increasingly popular as a web-enabling technology. Perhaps the simplest technique is to use the MQSeries Internet Gateway. This is essentially an MQ enabled CGI program. It can handle HTML "POST" requests, forwarding form data to a queue. A reply can be processed, with the appropriate HTML response being sent back to the client browser. Although simple, the Internet Gateway product provides a straightforward way of accessing existing applications via the web – a hot topic for many organisations at the moment.

More sophisticated applications can be developed with Java. Java is supported in a number of different ways. IBM describe this support as follows:

- MQSeries classes for Java and MQSeries classes for Java Message Service - The MQSeries classes for Java allow a program written in the Java programming language to connect to MQSeries as an MQSeries client using TCP/IP, or directly to an MQSeries server using the Java Native Interface (JNI). They allow Java applets, applications, and servlets access to the messaging and queuing services of MQSeries. If the client-style connection is used, no additional MQSeries code is required on the client machine. The MQSeries classes for Java enable a message-based approach to application integration using Java. MQSeries classes for Java Message Service is a set of Java classes that implement Sun Microsystems Java Message Service specification. A JMS application can use the classes to send MQSeries messages to either existing MQSeries or new JMS applications.
- The MQSeries Client for Java - an MQSeries client written in the Java programming language for communicating via TCP/IP. It enables Web browsers and Java applets to issue calls and queries to MQSeries giving access to legacy applications over the internet without the need for any other MQSeries code on the client machine.

Security

The "built in" security system within MQ is the Object Authority Manager (OAM). This system provides basic access control security to MQ objects, rather like Unix file and group permissions. With the OAM, it is possible to control which users or groups can connect to a queue manager, put or get messages to queues, or create objects dynamically. This system works well as far as access control is concerned, but cannot deliver the sophisticated message-level security that organisations now require.

Many organisations are looking towards PKI (Public Key Infrastructure) technologies to provide security services for their applications. A detailed description of PKI is beyond the scope of this document, but more details can be found from here.

IBM have produced an MQ SupportPac that allows MQ to participate specifically within Entrust's PKI infrastructure. The SupportPac is C source code for a channel message exit that enables the use of the PKI API (via GSS-API function calls), therefore enabling security features such as queue manager authentication, digital signing of messages, and message encryption.

Since the source code uses the GSS-API, the source code should be portable to environments that provide security services through other systems such as the Massachusetts Institute of Technology's Kerberos system. A version of Kerberos is now being used (somewhat controversially) in Microsoft's Windows 2000 operating system.

As the web page for SupportPac MS0C says:

"The mechanism used by MQSeries to transfer messages between two adjacent queue managers, or between a client and a queue manager is called a channel. This SupportPac provides security services at the middleware level; more specifically, it integrates with the MQSeries channels as channel exit programs. These channel exit programs are based on the well-defined open standard security application interface (GSS-API). The GSS-API services are provided by the EntrustSession toolkit, which runs in the Entrust/PKI environment. Entrust is a company that provides Certificates and Key management services. More information about the company and its products can be found at Entrust Technologies."

This approach offers a convenient way to integrate MQ into a PKI. The MQ applications do not have to have any knowledge of the PKI environment, as the channel exit will be executed by the queue manager, outside of the control of the application.

The MQI (Message Queue Interface)

Background

The MQSeries MQI benefits from standardisation across all platforms. Although there are semantic differences between programming languages, the basic verbs and data structures are the same if you are programming in C on Solaris, or COBOL on OS/390. In addition, an OO interface is available for C++ and Java programmers.

The API is physically implemented as a set of run-time libraries linked in to the MQ application. The application then invokes each verb in a fashion appropriate to the programming language environment, for example a C function all, a class instance, or subroutine call.

A brief overview of each verb is given below. Verbs are grouped together to illustrate complementary functionality. The overview does not give the full prototype for each verb, just a description of the function performed by that verb.

Language Support

There is direct support for C, C++, Java, COBOL, Perl (via a SupportPac), and PL/I.

Verbs

MQCONN / MQCONNX / MQDISC

MQCONN connects the application to a named queue manager. If successful, a handle is returned that is passed to all subsequent MQ calls. MQCONNX is a variation of MQCONN, but it establishes a “trusted” connection to the queue manager. This means that the application bypasses an IPC layer that all MQI calls normally go through, and operations are processed more rapidly. The disadvantage of MQCONNX is that an errant MQ application can adversely affect the queue manager.

MQDISC disconnects an application from a queue manager.

MQOPEN / MQCLOSE

Open or close an MQSeries object. Usually associated with opening and closing queues.

MQPUT / MQGET

Get or put messages to a previously opened queue.

MQBEGIN/ MQCMIT / MQBACK

These are the MQSeries unit-of-work (syncpoint) co-ordination verbs. They can be used if the application requires MQSeries to control units of work. If an external transaction manager is being used then the application would use the calls appropriate for that environment. See the section on XA Compliance for more information.

MQBEGIN marks the start of a unit of work. This is a useful shorthand that means that every subsequent MQ operation will be processed under the same unit of work. The alternative to this is to include an option within each MQ operation that explicitly states that the operation is to included under syncpoint.

MQCMIT commits a unit of work, so all MQPUT and MQGET operations are physically, rather than logically executed. MQBACK “rolls out” a unit of work, effectively undoing the last group of operations under syncpoint.

MQINQ / MQSET

MQINQ provides the application with a way to retrieve the attributes of an MQ object. They may be inspected, altered, and then applied to the object using the MQSET verb.

expiry.

Part 3: Products in the Marketplace

Security Infrastructure

Apart from providing the ability to interface to key management systems like Entrust, IBM have many products in the PKI space. Tivoli SecureWay Public Key Infrastructure, IBM Vault Registry and IBM Payment Registry. See the link below for an array of products. PKI products are appearing now on a regular basis.

Microsoft have a component called “Windows 2000 Certificate Services”. The link below takes you to a page where you can download a document that details this component fully.

Hewlett Packard’s OpenView product suite includes a security management solution, although details of how this works are not entirely clear from their web site.

MIT's Kerberos is a freely-available (subject to US imposed restrictions on the export of cryptographic software) implementation of a client/server network authentication system.

MQSeries Configuration and Systems Management

Candle Command Centre, BMC Patrol

Links

BEA Systems
Candle Corporation
EAI Journal The Enterprise Applications Integration Journal
Entrust Technologies
Hewelett Packard's OpenView security page
IBM Developerworks. Many useful resources covering a wide range of IBM and Open Source technologies.
IBM MQSeries Home Page
IBM MQSeries SupportPacs
IBM Security Products
Kerberos Network Authentication System
Level 8 Software
MessageQ.Com Middleware news and analysis
Microsoft Windows 2000 Certificate Services
Momentum Software
The Free Online Dictionary of Computing
The Open Group
