Using UML in WebSphere Business Integration Message Broker Solution Architecture

Arunava (Ron) Majumdar
Sr. IT Specialist
Software Services for WebSphere (SE Region)
arunava@us.ibm.com
IBM

Guy Hochstetler
Consulting I/T Specialist
WebSphere Business Integration National Practice
guyh@us.ibm.com
IBM

Contents

Modification History

Date	Version	Author(s)	Description
12/08/2002	1.0.0	David Grainger	MD08 - WebSphere MQ - Network Design Notation
03/23/2005	2.0.0	Arunava Majumdar Guy Hochstetler	Design Notations for Websphere MQ and Websphere Business Integration Message Broker

Legal Disclaimer:

COPYRIGHT LICENSE:

Acknowledgement:

We would like to acknowledge the help and support that we received from all the IBMers for putting this paper together. We would like to take this opportunity to specially mention Kyle Brown, Grant Larsen, James Rumbough, James Conallen, James Amsden, Branislav Selic, Paul Verschueren, Keith Watson, Anthony O'Dowd**,** Andrew Hickson, Mark Taylor, Alan Powell, Vicente Suarez, Sandra Raleigh and Jon Shoemaker for their contributions and support leading to the publishing of this paper.

We would also like to acknowledge the contributions by David Grainger for his publication of the WMQ Support Pack md08 – WebSphere MQ - Network Design Notation Version 1.0 – that introduced the WMQ Notations in the former version of the paper.

Scope of the Document:

The intent of this document is to provide a standard way of representing WebSphere MQ (WMQ) and WebSphere Business Integration Message Broker (WBIMB) based objects for design purposes. Using UML 2.0 standards where applicable in the diagrams, the authors have attempted build a standard for representing WMQ objects. This guide establishes the WMQ and WBIMB notation foundation upon which future Rational tools may be capable of generating actual WMQ and WBIMB objects.

Introduction:

Architecting WMQ and related products requires special symbols that describe the architecture in more meaningful ways. The majority of WMQ and WBIMB practitioners typically devise their own notations when designing and documenting their WMQ-based solutions. Acceptance of the notations described within this document will hopefully provide a much needed standard. Additionally, considering that UML was used as the guide in defining the WMQ-based notations positions them for inclusion in future IBM Rational development tools.

Please note that the notations presented in this document are originally based on the Support Pack md08 for Websphere MQ Design Notations but are extended to provide more WMQ design granularity and to account for Message Broker specific requirements as well as Enterprise Messaging Bus needs based on a Message Oriented Middleware (MoM).

The remainder of this document is divided into the following sections:

- **Guiding Principles**
  The Guiding Principles further describe the reasons for creating this document. They also provide the approach used in selecting the notation types for the WMQ objects.

- **WMQ Notations**
  Included here are the notation and their detailed descriptions for WMQ objects.

- **WBIMB Notations**
  Similar to the WMQ Notations, this section also provides the notation and detailed description for WBIMB objects.

- **Diagrams**
  This section brings it all together. It provides examples of using the previously described notations in design diagrams. It also introduces UML notation into messaging design with sample sequence and activity diagrams. Additionally, this section introduces a way to document message structures.

- **Conclusions**
  This section simply and briefly summarizes the main points of the document.

1. Guiding Principles:

The following are the principles that should guide the development of architectural and design models for representing computer systems.

1.1. Aesthetics

An architectural or design diagram or model is a pictorial representation of the underlying system. It is to provide a comprehensive view of the system that it describes. Aesthetics of the model should be given high priority while developing notations for these diagrams as well as when the Lego pieces are put together to construct the diagram. Neatness in the depiction of systems in a model helps in a better understanding of the system and quicker learning curve.

1.2. True Representation

The representation of the system in the diagram should be accurate. Diagrams should only show system details to the extent that correctly illustrates the functioning of the system. All system parameters should not be listed in a diagram. That would only make the diagram more complex and not very effective. Models should be able to drill down to other models for further information regarding the system. Since all aspects of the system are not shown in a particular model, the accuracy of representation becomes very important to portray the idea.

1.3. Shorten Developmental Effort and Understanding

Architectural and design models are build to help in the developmental and maintenance efforts as well as provide a better understanding of the system. Higher level models help capture business ideas or system overviews that are further illustrated with the help of lower level models. Thus handing a model over to another developer to maintain the system or integrate to the system is achieved in a better structured manner than having to go through the details in every level, every time maintenance or integration needs to be done. The building of models should not, however, get in the way of software development or integration. Too much documentation or modeling takes away the purpose of the document or model itself. It is to facilitate and augment development, not to restrain.

1.4. Standardization

One of the issues regarding textual model of documentation is that there is often no standard way of representing system design / architecture. A visual model often helps in representing the system in a standard way across different projects enhancing reusability of the design and better comprehension. The design notations in the model, therefore needs to be standardized for representing various scenarios and help in the modulation of patterns.

1.5. IDE Tooling

Integrated Development Environment (or IDE) is the basis for developing and managing software code faster and easier. Models should provide facilities for IDE tools to generate more detailed lower level models. E.g. Architectural Models should be able to generate deployment scripts, etc. The process of developing notations for models should consider the underlying fact that they will be in future be used to generate lower level models with the help of IDE tools.

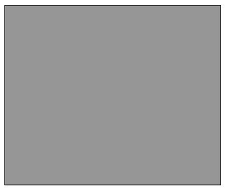
2. WMQ Notations

This chapter defines the notations for all the WMQ architectural and design components. The notations used in this chapter may be used in conjunction with the WBIMB notations defined in the next chapter (Chapter 3) and other related notations in several diagrams as illustrated in Chapter 4.

2.1. *System Components*

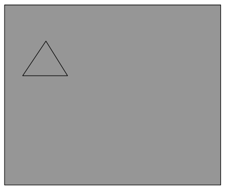
This section relates to system component objects specifically used in conjunction with WMQ. Notations for all other system components are outside the scope of this document.

2.1.1. Node



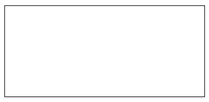
The notation for a node is represented as a gray rectangle with the stereotype **<<node>>**. A Node represents either a machine or a Logical Partition (LPAR). The hostname can be represented by either the name of the machine or LPAR in the DNS or an IP address. The IP address may also be a virtual IP as in the case of a High Availability Cluster like HACMP in the case of IBM AIX operating system HA cluster.

2.1.2. Z/OS Coupling Facility



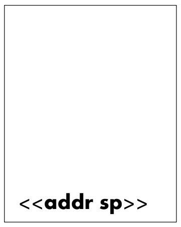
The coupling facility attached to a Z/OS system for shared memory access between different systems. It is represented as a gray rectangle with the stereotype **<<cf>>**. The notation triangle may be used to represent the coupling facility.

2.1.3. Z/OS Channel Initiator



The channel initiator address space is represented by a rectangle and the stereotype **<<chinit>>**.

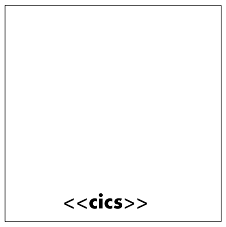
2.1.4. Z/OS Address Space



The address space is represented by a rectangle and the stereotype **<<addr sp>>**.

<<**node**>> hostname

2.1.5. Z/OS CICS Region

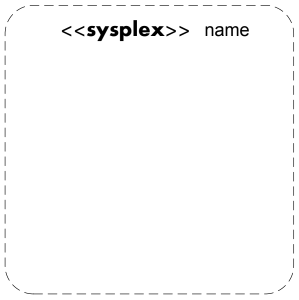


The CICS region is represented by a rectangle and the stereotype **<<cics>>**.

2.1.6. Z/OS CICS Transaction

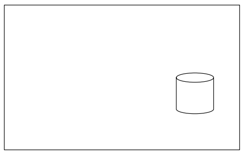
The CICS transaction is represented by a light gray rectangle and the stereotype **<<trn>>** and the name of the transaction.

2.1.7. Z/OS Sysplex



The Z/OS sysplex is represented by a rectangle with dashed outline and rounded edges with the stereotype **<<sysplex>>** and the name of the sysplex. The nodes that are part of the sysplex need to inside the outline of the sysplex notation.

2.1.8. Queue Manager

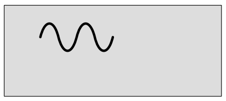


Queue Manager is represented by a rectangle with the stereotype **<<qmgr>>**. For all distributed systems the queue manager name is as QM.*. For Z/OS the name of the queue manager can only be four characters long as it is an address space. Thus the naming convention for does not apply to Z/OS. If the queue manager is a full cluster repository, it is represented with the cylinder with a qualifier **R**. The cluster that the queue manager belongs to or is hosting the repository for is represented by the cluster outline illustrated below. For cluster name lists, the queue manager may reside in two or more clusters. The listener port of the listener associated with the queue manager may be represented within parenthesis after the name of the queue manager, e.g. QM1(1515). Port 1414 is the default port for WMQ and may not be explicitly stated. Multiple listeners associated with the queue manager may be listed separated by commas (,) within the parenthesis. Listeners may also be represented as daemon processes associated with the queue manager as explained later.

2.1.9. Application

Any Application in the system is represented as in the diagram, by a rectangle and the stereotype **<<app>>**.

2.1.10. Thread



Application threads are represented by a light gray rectangle and the stereotype **<<thread>>**. The sinusoid within the diagram represents the thread as well. Either the stereotype or the sinusoid notation or both may be used to represent the thread. The thread runs in the process defined by the application and hence needs to be represented inside the application rectangle. The name of the thread run function or the thread class may be shown in the thread notation. If there are multiple threads spawned by the application of the same type, the thread instance name may be shown next to the thread function name or the thread class name separated by a colon (:).

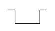
R

<<qmgr>> QM.name

2.2. Queues

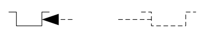
All queues are represented by a similar outline ( ). However, different types of queues have different characteristics. Message hosting queues are represented by a solid outline and non-message hosting queues are represented by the dashed outline. The detailed notations for all types of queues are illustrated in this section.

2.2.1. Local Queue



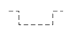
A Local Queue is represented as in the diagram with solid lines. A local queue has a physical existence on the file system where messages are stored and thus represented by a solid outline. The wildcard character * may be used to represent the names of a set of queues and a note listing all the application queues referred to. This convention is used to represent a large number of application queues without making the diagram difficult to read.

2.2.2. Alias Queue



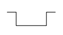
Alias Queue represented in the diagram is an alias definition for a local queue, a remote queue, a local queue shared in a cluster or another alias queue. The dashed arrow with the stereotype **<<ref>>** represents the local queue it refers to. If it refers to a different type of queue it should point accordingly. The alias queue never stores messages and represented by dashed outline with the qualifier **A** inside the outline.

2.2.3. Remote Queue



Remote Queues are definitions that forward messages to a queue in a remote Queue Manager. They do not store messages and should be represented by a dashed outline with the qualifier **R** inside the outline.

2.2.4. Transmission Queue



Transmission Queue is a special queue to enable the MCA to store and forward messages to a remote Queue Manager and represented by a solid outline with the qualifier **X** inside the outline.

Local
Queue

2.2.5. Clustered Queue

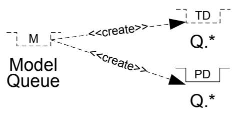
A local queue shared in the cluster is represented like a local queue with a qualifier **C** indicating that it is shared in the cluster. The queue needs to be in the correct boundary of the cluster (next section). The representation of the same queue as it appears on other queue managers in the cluster is represented similarly but with a dashed outline. This is often useful to indicate an application writing to a clustered queue not defined on the queue manager. The get operation is not possible from this queue since it is not locally defined. Even though there might be several queues in the cluster with the same name for load balancing, only one instance of the remote clustered queue may be shown and the load balancing is considered to be the assumed queue manager functionality in a cluster.

2.2.6. Shared Queue

A shared queue is a definition of a queue on the Coupling Facility that is shared in the queue sharing group (explained later). It is represented by the outline with the qualifier **S**.

2.2.7. Model Queue

C



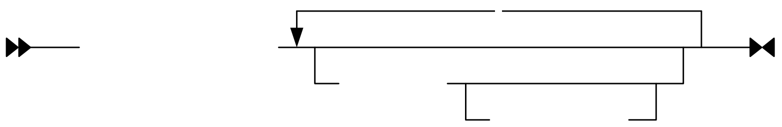
A model queue is a template for a queue that needs to be created at runtime and represented as shown with dotted lines and the letter **M** in the middle. The permanent dynamic queue created at runtime is represented with the solid outline, the qualifier **PD** and a set of names that represents the queues. The temporary dynamic queues are represented with dashed lines with a qualifier **TD** denoting a temporary dynamic queue. A dashed line from the model queue to either kind of dynamic queues with the stereotype **<<create>>** represents the dynamic creation of these queues. The life time of these queues and sequence of events leading to the destruction of these queues can be better represented by a sequence diagram of the program.

Clustered Queue

Remote Clustered Queue

2.3. Queue Access

Accessing messaging queues, i.e. putting and getting messages to and from queues, is represented by an arrow in the direction of the data flow and the qualifier syntax as below.



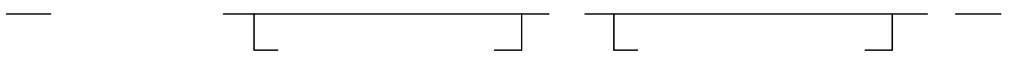
where, <Access Type> ⇒ the type of messaging object access operation
<Parm> ⇒ parameter for that type
<Value> ⇒ value of the parameter

The following table indicates the valid options for each type of messaging object access and the corresponding parameters and values. The arrow may only point to a valid object.

Access Type	Operation	Parameter	Description	Value Type	Value
<<put>>	Put message	`TX`	Transactional		
		`PER`	Persistence		
		`PRI`	Priority	Integer	[1,15]
<<get>>	Retrieve message	`TX`	Transactional		
		`PER`	Persistence		
		`PRI`	Priority	Integer	[1,15]
<<pub>>	Publish on topic	`Global`	Global scope		
		`Local`	Local scope		
<<sub>>	Subscribe on topic				
<<regsub>>	Register subscriber				
<<deregsub>>	Deregister subscriber				
<<delpub>>	Delete retained publication				
<<requpd>>	Request updated publication				

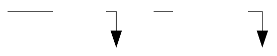
<Access Type>

Conditional statements may also be incorporated as parameters in the following syntax.



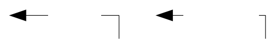
An example of a valid conditional statement parameter is **`GET(TX?PER:)`** .

2.3.1. Putting Message to a Queue



Putting a message into a queue is represented by an arrow with the stereotype **<<put>>**. If the put is under a transaction (syncpoint) it is represented as **<<put>> TX**. Similarly any other parameter can be represented as per the syntax above.

2.3.2. Getting Message from a Queue



Getting a message from a queue is represented by an arrow with the stereotype **<<get>>**. If the get operation is under a transaction (syncpoint) it is represented as **<<get>> TX**. Similarly any other parameter can be represented as per the syntax above.

2.3.3. Publishing on a Topic

Publishing a message on a topic is represented by an arrow with the stereotype **<<pub>>**. The stereotype may be followed by the scope of the published message – global (**<<pub>> Global**) or local (**<<pub>> Local**). Also refer to section 3.2.

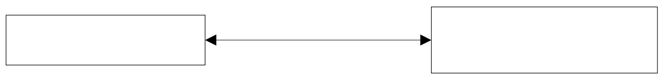
2.3.4. Subscribing on a Topic

The subscription received from the broker is represented by an arrow with the stereotype **<<sub>>** towards the subscriber application. Also refer to section 3.2.

2.4. Channels

Channels are definitions for communication between an application and a queue manager or between two queue managers and are represented by arrows. Different types of arrows and arrow heads are used to distinguish between different types of channels. For any channel in the network (TCP, SNA, etc.) the channel name should indicate the network protocol and is governed by naming standards.

2.4.1. Server Binding

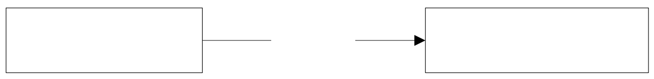


The application needs to reside on the same node as the queue manager for server binding. It is represented by a two-ended arrow indicating that the messages are sent both ways. However, it does not require any channels to be defined and hence no text is mentioned on the arrow connector. There are multiple types of server bindings and if required a note may be attached to the server binding notation with details about the particular binding used.

2.4.2. Client-Server Binding

The application may or may not reside on the same node as the queue manager and the messages are sent through the Client channel on the Client Application end and the Server Connection channel on the queue manager end. It is represented by a two-ended arrow with the name of the SVRCONN channel name specified on the arrow connector.

2.4.3. Sender-Receiver Pair

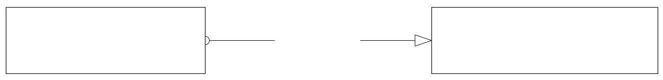


The Sender channel on the queue manager QM.A communicating to the Receiver channel on the QM.name is represented by an arrow with the name of the channel indicated. The normal beginning of the arrow represents the Sender MCA and the filled arrow head represents the Receiver MCA. The naming standards dictate the name of the channels as the same as the remote queue manager prefixed with "TO.". The direction of the arrow connector represents the direction in which the messages are flowing.

<<app>> name

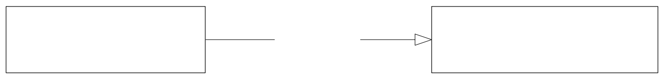
Serve

2.4.4. Server-Requester Pair



The Server channel on the queue manager QM.A communicating to the Requester channel on the QM.name is represented by an arrow with the name of the channel indicated. The circle at the beginning of the arrow represents a Server MCA and the unfilled arrow head represents a Requester MCA. The naming standards dictate the name of the channels as the same as the remote queue manager prefixed with “TO.”. The direction of the arrow connector represents the direction in which the messages are flowing.

2.4.5. Requester-Sender Pair



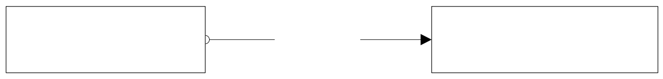
The Sender channel on the queue manager QM.A communicating to the Requester channel on the QM.name is represented by an arrow with the name of the channel indicated. The normal start of the arrow represents the Sender MCA and the unfilled arrow head represents a Requester MCA. The naming standards dictate the name of the channels as the same as the remote queue manager prefixed with “TO.”. The direction of the arrow connector represents the direction in which the messages are flowing.

2.4.6. Server-Receiver Pair

<<**qmgr**>> QM.A

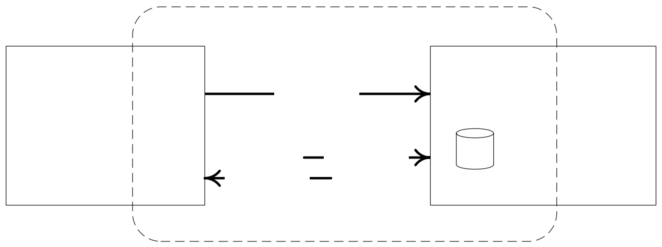
TO.Q

Server-



The Server channel on the queue manager QM.A communicating to the Receiver channel on the QM.name is represented by an arrow with the name of the channel indicated. The circle at the beginning of the arrow represents a Server MCA and a filled arrow head represents the Receiver MCA. The naming standards dictate the name of the channels as the same as the remote queue manager prefixed with “TO.”. The direction of the arrow connector represents the direction in which the messages are flowing.

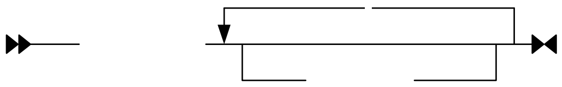
2.4.7. Cluster Channels



The cluster channels are represented by open arrows as shown in the diagram. Only manually defined channels are shown in the diagram and not the auto-defined channels, since the latter does not affect the architecture of the cluster and are maintained by the queue manger and need not be scripted. The cluster sender points to a cluster repository queue manager (QM.repos) from another queue manager (QM.name) where the CLUSSDR channel is defined. This indicates the primary source for the non-repository queue manager to retrieve cluster information not present at that period of time in its partial repository. The name indicated on the cluster sender channel is, in accordance with the naming standards, “TO.” appended to the name of the target queue manager. The cluster receiver channels, on the contrary, are defined pointing to the queue manager where the definition of the CLUSRCVR exists with similar arrows and the name of the channel indicated on the arrow. The starting end of the cluster receiver channel is not attached to a queue manager since it publishes a network definition of the queue manager to the repositories in the cluster. The boundaries of the cluster is represented as a dashed rectangular outline with rounded corners and the stereotype **<<clus>>**. All cluster components represented in the diagram should be rendered within the perimeter of the cluster boundary notation. Overlapping cluster boundaries are represented with the same notation. A parallelism of this notation can be drawn from the Venn Diagram notation in Set Theory. If a queue manager falls within the intersection set of two or more cluster boundaries, then the queue manager is considered to be in the cluster namelist containing all the clusters. A cluster namelist repository is represented in the intersection set of the cluster boundaries.

2.5. Triggering and Backout

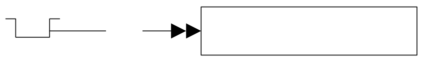
Triggering and Backout are represented as arrows with two filled arrow heads starting from the queue to the triggered application or the backout queue respectively. They all have the following syntax for the arrow label.



where, <Type> ⇒ the trigger type or backout
and <Parm> ⇒ parameters for that type.

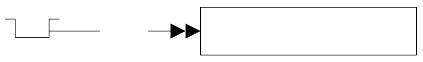
The ***Type*** and the ***Parm*** is different for different trigger types or backout operation and are described below.

2.5.1. Trigger on First



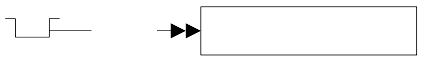
The Application is triggered when the first message is put to an empty queue. The triggering is represented by an arrow with the trigger type qualifier **`First`** indicated on it.

2.5.2. Trigger on Every



The Application is triggered for every message arriving on the queue. The triggering is represented by an arrow with the trigger type qualifier **`Every`** indicated on it.

2.5.3. Trigger on Depth



<Type> :

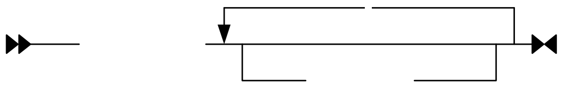
The Application is triggered when the queue depth reaches the trigger depth indicated on the queue. The triggering is represented by an arrow with the trigger type qualifier **`Depth`** and a single parameter **`n`** as the trigger depth.

2.5.4. Backout

When a message is rolled back into the queue the ***Backout Count*** is incremented by the queue manager. When the ***Backout Count*** of the message in the ***Message Descriptor*** reaches the ***Backout Threshold*** parameter of the queue, the WBIMB broker forwards the message to the ***Backout Requeue*** queue. The backout operation is represented by an arrow from the input queue to the backout queue with the type qualifier **`Backout`** and a single parameter **`n`** as the backout threshold.

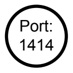
2.6. Daemon Processes related to WMQ

All daemon processes are represented with a circle and a qualifier with the following syntax.



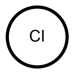
where, <Type> ⇒ the daemon type
and <Parm> ⇒ parameters for that type.

2.6.1. Listener



The queue manager listener is the component that listens for remote connections to a particular port and is represented by a circle and the qualifier type **Port** and a single parameter **n** as the port number.

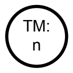
2.6.2. Channel Initiator



The channel initiator associated with the queue manager is required to trigger channels from their inactive state and is represented by the qualifier type **CI**. If all queue managers in a particular WMQ Architecture Diagram have the channel initiator running, then it is not required to show the channel initiator on each queue manager but a note with the channel initiator notation can be placed at the bottom of the diagram indicating that it is running on all the queue managers in the diagram unless otherwise stated.

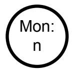
<Type> :

2.6.3. Trigger Monitor



The trigger monitor associated with the queue manager is required to trigger programs based on other parameters on the queue and is represented by the daemon circle and the qualifier type **TM** and the optional parameter **n** used to represent a non-standard trigger monitor.

2.6.4. Other Monitoring Systems



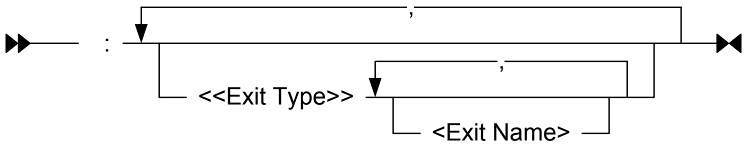
Any monitoring system other than the above mentioned standard monitors is represented as a daemon with the qualifier type as **Mon** and a single parameter **n** that indicates the particular monitoring agent. Event monitoring systems will be represented similarly.

2.7. MQ Exit, MQ Event, Expiry

This section is discusses different event options in the queue manager. An MQ Event is generated by the queue manager when certain internally monitored parameters are successful and event messages are generated by the queue manager. And MQ Exit is to trap an event while a message is in transit and the exit code is invoked. Expiry is a property of the message and the application session.

2.7.1. Exit

Different flavors of exits trap the processing of messages or communication between different queue managers. Exits are represented by a rectangle with the qualifier **X** in the font shown in the diagram following syntax below.



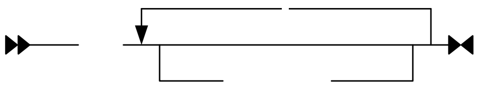
where,	<<Exit Type>>	⇒ the stereotype indicating the type of the exit
	<Exit Name>	⇒ the name of the exit (DLL)

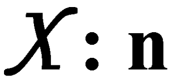
Stereotype	WMQ Exit Type
<<security>>	Security Exit for channels
<<message>>	Message Exit for the channel
<<receive>>	Receive Exit for the channel
<<send>>	Send Exit for the channel
<<API>>	API-Crossing Exit for the queue manage
<<WLM>>	Workload Management exit for workload management in a queue manager cluster
<<autodef>>	Exit is called before automatically defining the auto-defined channels in a cluster.a

Activity diagrams and other UML diagrams represent more detailed information about the internals of the exit routine.

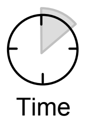
2.7.2. Event

Events are represented by a rectangle with the qualifier E in the shown font and the following syntax.





2.7.3. Expiry



Expiry of messages is represented as shown in the diagram with the expiry duration stated at the bottom. Expiry of messages is a property of the message itself and thus may be different for every message in the system. However, expiry for messages are set based on different types of messages and hence can be represented on the Message Specification Diagram (explained later). If the same message has different expiry under different conditions, then the expiry needs to be represented next to the queue access arrow in a QMAD or MID. The same representation can be applied to the session expiry in an application and to expiry for waiting period on a GET operation on a queue (a blocked call with a finite non-zero wait interval).

2.8. *Security*

WMQ security can be broadly divided into two categories – Channel Security, involving security of data in transit through the channels (SSL) and Access Security, involving access to WMQ resources in the system through the Object Authority Manager (OAM).

2.8.1. SSL Channels

SSL enabled channels are indicated by a lock sign as shown in the diagram on the arrow for the channel. The secured protocol or algorithm is indicated below the notation. Since the security protocol needs to be the same at both ends of the channel like the channel name, the notation can be placed at either end of the channel.

2.8.2. SSL Key Store

The key store for the digital certificates is to be defined for the queue manager or the client that is involved in the SSL channels. The physical location of the key store is indicated below the notation for the key as shown. Since this is a parameter for the queue manager it is necessary to show the key store within the boundaries of the queue manager.

2.8.3. WMQ Object

Any WMQ object is represented by a gray rectangle with the stereotype and the name of the object and thick border. The following table lists of stereotypes that are related to the WMQ objects.

Stereotype	**WMQ Object**
<<node>>	All queue managers on the node, complete installation
<<qmgr>>	Queue Manager
<<queue>>	Queue
<<process>>	Process
<<namelist>>	Namelist
<<authinfo>>	Authentication Information for SSL

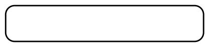
2.8.4. Authority Class

The authority class of the Object Authority Manager is represented by the grey rectangle with the stereotype <<auth>> and class indicated within it. The following are the list of valid classes: **all**, **alladm**, **allmqi**, **none**, **altusr**, **browse**, **chg**, **clr**, **connect**, **crt**, **dlt**, **dsp**, **get**, **put**, **inq**, **passall**, **passid**, **set**, **setall**, **setid**.

2.8.5. Group

The security group is represented by a rectangle with slightly rounded corners, thick border and the stereotype <<grp>> as shown in the diagram. The syntax for the text inside the rectangle is the domain (if the group mentioned is a group in a domain) followed by a colon (:) and the name of the group. The domain name is optional. Multiple groups can be represented in a single rectangle if they all refer to the same authority or a set of authorities.

2.8.6. Principal



The security principal is represented by a rectangle with slightly rounded corners and the stereotype <<usr>> as shown in the diagram. The syntax for the text inside the rectangle is the domain (if the principal mentioned is a principal in a domain) followed by a colon (:) and the name of the principal. The domain name is optional. Multiple principal can be represented in a single rectangle if they all refer to the same authority or a set of authorities.

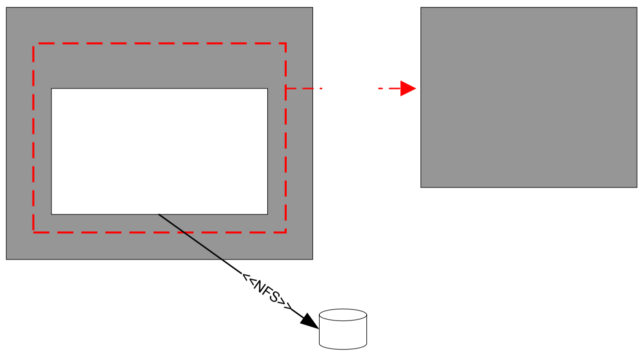
<<**auth**>> class

<<**grp**>> group

2.9. High Availability

High Availability environments essentially monitor system processes running in the same resource group with their persistent working data on a shared drive. These processes are installed on both nodes that are added to the HA environment or on a shared drive. When a node goes down, the processes are restarted on the failover node ensuring higher up-time for the application. HA failover mechanism varies from system to system and a detailed analysis of these systems is beyond the scope of the paper.

2.9.1. High Availability Failover



The diagram shows a very generic failover mechanism for the queue manager process. The resource group for the failover is represented by the dashed red rectangle. Processes within this resource group would be monitored by the HA system and failed over to the other node if the process (in this example the queue manager ***QM.name***) fails on one node. This is indicated by a cylinder with the qualifier **FS** (indicating file system). The mounting mechanism is also indicated on the arrow that points from the process over to the mounted file system. The mounting mechanism can, for example, be AFS, NFS, etc. The mount points are also indicated next to the cylinder. The mounting mechanism is indicated on the arrow as a stereotype as in the following table.

Stereotype	Failover Mechanism
<<AFS>>	AFS mounting
<<NFS>>	NFS mounting

The HA failover mechanism is represented by a dashed red arrow from the resource group to the node that it fails over to. The failover mechanism and the corresponding stereotype to be indicated on the arrow are listed in the following table.

Stereotype	Failover Mechanism
<<A-A>>	Active-Active failover configuration
<<A-P>>	Active-Passive failover configuration

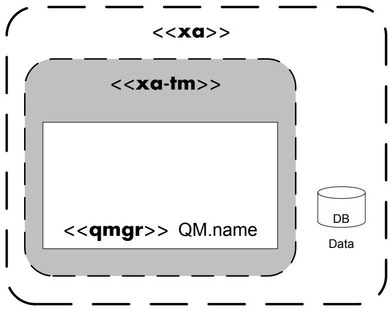
<<**node**>> node1

<<**rg**>> RG QM 1

2.10. XA Coordination

Transaction coordination is essential when multiple transaction managers are involved in a single transaction to guarantee the commit and rollback operation on the transaction. This is achieved by a two-part commit process between the different transaction managers supporting the XA protocol.

2.10.1. XA Coordination

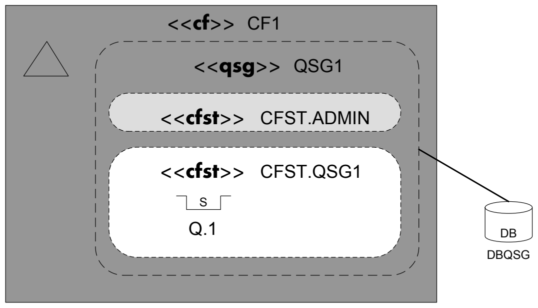


Transaction coordination between multiple resource managers in a global transaction context is represented by a dashed rectangle with rounded edges as shown in the diagram containing the transaction servers within its perimeter and the stereotype **<<xa>>**. The XA transaction manager is indicated inside a light grey rectangle with black dashed outline with rounded edges and the stereotype **<<xa-tm>>**.

2.11. Z/OS Shared Queue

Queue sharing is achieved through a coupling facility in a Z/OS environment. This feature of WMQ is limited to the Z/OS.

2.11.1. Queue Sharing Group



The Z/OS coupling facility is represented by a gray rectangle and stereotype **<<cf>>**. The queue sharing group is represented by a rectangle with dashed border, rounded edges and the stereotype **<<qsg>>**. The database that stores the information for the queues is represented as a cylinder with the qualifier **DB** and the name of the database below the cylinder. The coupling facility structures are shown as dashed rectangles with rounded edges with the stereotype **<<csft>>**. The queue manager uses an administrative coupling facility structure for its internal use and does not contain any user data. This is represented by a light gray rectangle for coupling facility structure as shown. The other structures may contain shared queue information. The shared queue is represented by a dashed outline with the qualifier **S**.

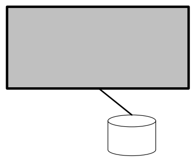
3. WBIMB Notations

This chapter defines the notations for all the WBIMB architectural and design components. The notations used in this chapter may be used in conjunction with the WMQ notations defined in the previous chapter (Chapter 2) and other related notations in several diagrams as illustrated in Chapter 4.

3.1. Broker Architecture

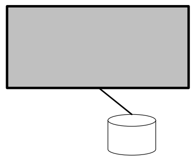
This section discusses the notions for the elements of the message broker as design components. These can be used in the QMAD and MID diagrams.

3.1.1. Configuration Manager



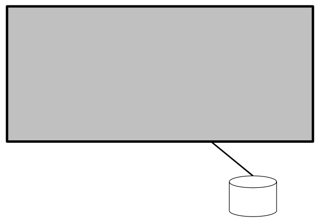
The configuration manager is represented by a light gray rectangle with the stereotype **<<cmgr>>**. The configuration manager database is represented by a cylinder with the qualifier DB and the name of the database below and connected to the configuration manager with a line.

3.1.2. User Name Server



The username server is represented by a light gray rectangle with the stereotype **<<unsvr>>**. The username server database is represented by a cylinder with the qualifier DB and the name of the database below and connected to the configuration manager with a line.

3.1.3. Message Broker

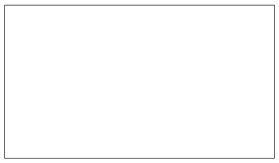


The message broker is represented by a light gray rectangle with the stereotype **<<broker>>**. The message broker database is represented by a cylinder with the qualifier DB and the name of the database below and connected to the configuration manager with a line.

<<cmgr>>

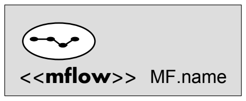
DB

3.1.4. Execution Group



The execution group is represented by a rectangle with the stereotype **<<egrp>>**. The execution group needs to be represented inside the broker outline if used along with the broker notation.

3.1.5. Message Flow



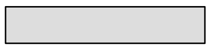
The message flow is represented by a light gray rectangle with the stereotype **<<mflow>>**. The message flow needs to be represented inside the execution group or the broker outline if used along with these notations.

<<**egrp**>> EG.name

3.2. *Publish/Subscribe*

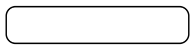
Publish/Subscribe is a many-to-may model for message distribution based on topics. This section discusses the notions for publish/subscribe related to the message broker as design components. These can be used in the QMAD and MID diagrams.

3.2.1. Topic



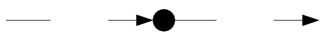
A topic in a topic tree is represented by a light gray rectangle with the stereotype **<<topic>>**.

3.2.2. Subscription Point



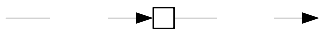
A subscription point is represented by a white rectangle with rounded edges. The subscription point may be attached to a topic in a topic tree.

3.2.3. Event Publication



Publication of events that are sent to the registered subscribers synchronously is represented by a solid black circle.

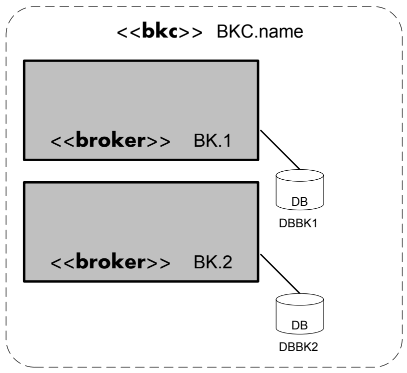
3.2.4. Retained Publication



Retained publication that stores the current state of the publication that may be delivered to the subscriber synchronously or asynchronously, is represented by a white square with black border.

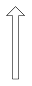
<<**topic**>> topic

3.2.5. Broker Collective



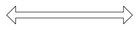
The broker collective is represented by a rectangle with dashed border and rounded edges with the stereotype **<<bkc>>**. The brokers that belong to a broker collective should be represented within the outline of the broker collective notation as shown.

3.2.6. Broker Parent-Child Relation



Two or more brokers may be in a parent-child relation in the form of a broker tree structure. The parent-child relation between the brokers is represented by the thick white arrow with a black border as shown in the diagram with the arrow starting from the child broker and pointing towards the parent broker.

3.2.7. Broker Association Relation



Two or more brokers may be in an association relation with each other in a broker tree structure. The association relation between the brokers is represented by the thick white two-ended arrow with a black border as shown in the diagram connecting the two brokers.

3.3. Message Specification

Message specification elements are represented in different colors to represent different kinds of elements to achieve a quick visual understanding of a complex message structure. The colors are chosen so that even a grayscale representation would provide distinguishing shades to signify the different element types. Moreover, the content and position of the elements provide distinction in what they represent.

3.3.1. Complex Type Element

A complex type element is represented by a gray rectangle with a black border and the name of the element within in. The message itself is a complex type and any Message Specification Diagram (described later) must have this element at the top of the message tree. The type definition is represented in bold and separated from the element instance with a bold colon (**:**).

3.3.2. Element Hierarchy in Message Tree

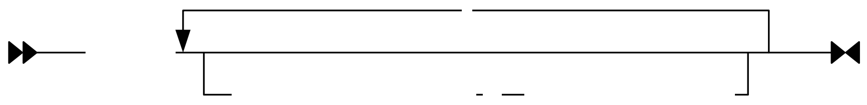
Elements in the message tree is represented a line at right angles to another starting from the complex type or list type element at the top to the decomposed element at the end of the horizontal line. This represents the hierarchical structure of the message tree.

3.3.3. Data Element

Data element is represented by a yellow rectangle with a black border. The data element models the physical data present in the incoming message.

3.3.4. Element Attributes

The attribute of the data element is represented by a white rectangle with black border. The data type is represented as a stereotype. The attributes of the data element is represented following the data type if necessary. The syntax for the representation is:

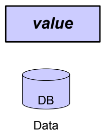


Typ: elem

E.g. for fixed length string of length 10 the representation is '<<string>> Len=10'. The following table indicates the different data types and the attributes related to them.

Stereotype	Attribute Notation	Attribute Description and Values
<<string>>	Len	Length (integer)
	Pad	Padding
	Align	Alignment (l ⇒ left, r ⇒ right)
	Def	Default value
<<int>>	Bin	Number of bytes of binary data
	Len	Length (integer)
	Pad	Padding
	Align	Alignment (l ⇒ left, r ⇒ right)
	Def	Default value
<<decimal>>	Bin	Number of bytes of binary data
	Len	Length (integer) and Precision(integer) separated by comma(,)
	Pad	Padding
	Align	Alignment (l ⇒ left, r ⇒ right)
	Def	Default value
<<float>>	Bin	Number of bytes of binary data
	Len	Length (integer)
	Pad	Padding
	Align	Alignment (l ⇒ left, r ⇒ right)
	Def	Default value
<<blob>>	Len	Length (integer)
<<list>>		The name of the list is indicated

3.3.5. Element Value



The value of the data is represented by a light blue rectangle with a black border. If the value is to be determined from a database query the value is represented as a light blue cylinder with the qualifier DB inside the cylinder as shown and the name of the database below the cylinder. The element value notation is used in case of a static value set for that data element or in a list data type for alternate values that can be present for the data element. The database representation is always associated to a list data type since it is a dynamic selection of a set of values from the database at any given point. The SQL for the data selection criteria may be shown as an annotation next to the cylinder. The static value set is represented using common set theory notations. Some examples are shown below.

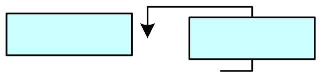
[01, 06] $\cup$ **[99]** $\Rightarrow$ the accepted values are 01, 02, 03, 04, 05, 06, 99 (for an element with length = 2 and padding = 0, the 0 in the front of the numbers is significant from a data validation perspective).

[00, 09] $\cup$ **[a, z]** $\cup$ **[A, Z]** $\Rightarrow$ the accepted values are 01 to 09 and all lower case and upper case alphabets.

3.3.6. List Data Type

List data type is represented by dark gray rectangle with the name of the list indicated in white font. This type may contain a fixed or variable set of values for validity checking. In the former case the element values are listed and in the latter case the values are derived from the database and represented by the database element value notation and an annotation provided for the SQL query statement.

3.3.7. Element Occurance

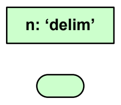


Element occurrence can be represented in either of the two ways shown. In both the cases the element occurrence is represented by a light turquoise rectangle with black borders. The syntax of the element occurrence qualifier is as follows.

$<n_{min}, n_{max}>$ where, $n_{min} \Rightarrow$ Minimum occurrence value
$n_{max} \Rightarrow$ Maximum occurrence value

In the first notation the element occurrence is either <0,1> or <1,1>, i.e. the element is either absent or occurs no more than one time. In the second notation the element can occur more than one time and is indicated by the arrow representing a recursive condition. The two representations are made distinctive since it is easier to recognize recursive elements in the diagram and often handled differently in the code (multiple occurrences are usually treated as an array and looped through in the program). A mandatory element has n_{min} value of greater than 0.

3.3.8. Delimiter



The delimiter is represented by the light green rectangle with a black border.

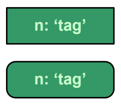
The larger rectangle shown is the type representation of the delimiter. If the delimiter is a long sequence of characters then it is recommended to use an alpha-numeric identifier to represent the long sequence for the delimiter separated by a colon (:).

The delimiter instance is represented by a rectangle with rounded corners. This notation is used between elements that are separated by the delimiter in the message tree where the delimiter instance is present. This is done for visual clarification. The qualifier may either be the delimiter or the delimiter identifier for long sequence of delimiters.

Non-printable characters used as delimiters can be represented either with the hexadecimal sequence with the hexadecimal qualifier **x** or with the mnemonics as in the following table.

Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex
<ACK>	x'06'	<BEL>	x'07'	<BS>	x'08'	<CAN>	x'18'
<CR>	x'0D'	<DC1>	x'11'	<DC2>	x'12'	<DC3>	x'13'
<DC4>	x'14'	<DLE>	x'10'	<EM>	x'19'	<ENQ>	x'05'
<EOT>	x'04'	<ESC>	x'1B'	<ETB>	x'17'	<ETX>	x'03'
<FF>	x'0C'	<FS>	x'1C'	<GS>	x'1D'	<GT>	x'3E'
<HT>	x'09'	<LF>	x'0A'	<LT>	x'3C'	<NAK>	x'15'
<NUL>	x'00'	<RS>	x'1E'	<SI>	x'0F'	<SO>	x'0E'
<SOH>	x'01'	<SP>	x'20'	<STX>	x'02'	<SUB>	x'1A'
<SYN>	x'16'	<US>	x'1F'	<VT>	x'0B'		

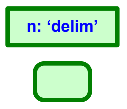
3.3.9. Tag



Tags passed to the message need to be distinguished from the actual data. Tags are represented by a sea green rectangle with black border and light green text. The tags are indicated with a tag identifier to indicate the instances of related tags. There can be two possible ways to indicate tags alongside data, either as fixed length or with a tag delimiter (next article). For fixed length tags the length is indicated as ‘Len = <length>’. Fixed length tags and tag indicators are mutually exclusive. The tag definition name is to represent the type of tags that are to be expected for the complex structure.

The tag instance is represented by the rectangle with rounded edges. The tag identifier in the tag definition and the tag instance need to match. The tag instance name may be a wildcard character (*) or a specific literal that needs to be matched in the data. Using the wildcard implies creation of the tag names at runtime based on the tag names that are passed in the data. This is the case where the data has self-defined tags that need not be validated as long as the data contained in the tags is valid.

3.3.10. Tag Data Separator

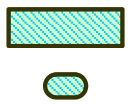


Tag Data Separator is used to distinguish between the tag and the data parts for variable length tags. It is also known as the tag delimiter since it acts as a delimiter for the tag. It is represented by a light green rectangle with a thick dark green border and blue text color for the qualifier.

Like the delimiter, the tag data separator may have an optional identifier for representing a long sequence of characters. For non-printable character values please refer to the mnemonic table (vide 3.2.8).

The tag data separator instance is represented by the rectangle with rounded edges. The qualifier for the tag separator instance may be the identifier for the tag separator for long sequences or the tag separator itself.

3.3.11. Group Indicator

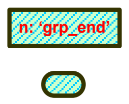


Group Indicator is represented by the light green rectangle with turquoise NW-SE stripes with a thick olive green border and red text color for the qualifier. The group indicator is used in conjunction with the group terminator – always as a pair – to represent grouping of data. It represents the start of the group.

Like the delimiter, the group indicator may have an optional identifier for representing a long sequence of characters. For non-printable character values please refer to the mnemonic table (vide 3.2.8).

The group indicator instance is represented by the rectangle with rounded edges. The qualifier for the group indicator instance may be the identifier for the group indicator for long sequences or the group indicator itself.

3.3.12. Group Terminator

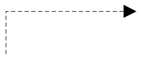


Group Terminator is represented by the light green rectangle with turquoise NE-SW stripes with a thick olive green border and red text color for the qualifier. The group terminator is used in conjunction with the group indicator – always as a pair – to represent grouping of data. It represents the end of the group.

Like the delimiter, the group terminator may have an optional identifier for representing a long sequence of characters. For non-printable character values please refer to the mnemonic table (vide 3.2.8).

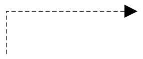
The group terminator instance is represented by the rectangle with rounded edges. The qualifier for the group terminator instance may be the identifier for the group terminator for long sequences or the group terminator itself.

3.3.13. Attribute Reference



The attribute reference is represented by a dashed arrow with the stereotype <<refers>>. This is used when the attribute length of the element in the tree is dependant on the value of another element. The referred to value is used in the length attribute for a fixed length element. This gives more flexibility in defining the fixed length field if it is only determined at runtime.

3.3.14. Element Correlation

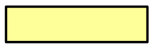


The element correlation is represented by a dashed arrow with the stereotype <<correlation>>. This notation is used when the value of one list type element depends on the value of another list type element e.g. if the value of element A is x then the value of element B needs to be 1, if the value of element A is y then the value of element B needs to be 2, etc. then there exists a correlation between the elements A and B.

<<refers>>

<<correlation>>

3.3.15. CAPP Element

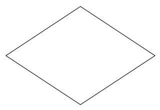


A Conditional and Progressive Parsing (CAPP) Pattern element is represented by a yellow rectangle with a thick black border. A CAPP element is a non-atomic element that may be broken down into its component parts in subsequent iterations in the flow or at the same point conditionally based on the value in the data, as defined by the related CAPP message definition. The element name is represented similar to a simple element but may refer to a CAPP message with an optional CAPP condition and CAPP staging indicator.

3.3.16. CAPP Message

A Conditional and Progressive Parsing (CAPP) Pattern message is represented by a gray rectangle with a thick black border. A CAPP message is a decomposition definition for the CAPP element. The message is represented similar to a complex type element and may be a separate message in the implementation. The CAPP message refers to a CAPP element for the conditional parsing of the message.

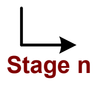
3.3.17. CAPP Condition



A Conditional and Progressive Parsing (CAPP) Pattern condition is represented by a white rhombus with a gray shadow. The condition is represented by a number based on the conventions for the CAPP staging and the branching condition itself.

CAPP elem

3.3.18. CAPP Staging



A Conditional and Progressive Parsing (CAPP) Pattern staging is represented by a thick-lined black arrow and the CAPP stage number in bold brown font.

CAPP ref msg

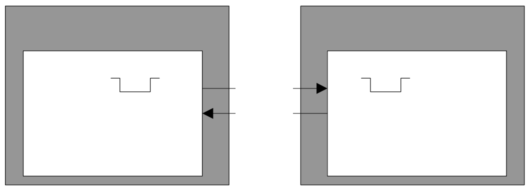
4. Diagrams

This section will provide architecture diagram samples that demonstrate the above described WMQ and MB notations. It also introduces examples of UML sequence and activity diagrams representing messaging solutions. Finally, it describes a useful way for representing message structures.

4.1. Queue Manager Architecture Diagram

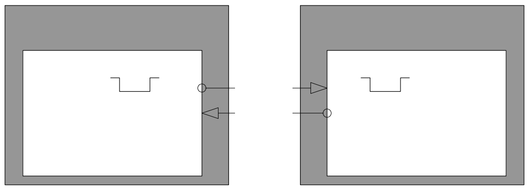
This section illustrates different examples in which the notations may be used in a Queue Manager Architecture Diagram (QMAD). A QMAD represents the architectural diagram for a queue manager and its configuration in different nodes, the intercommunication between the queue managers, the configuration for High Availability clusters, etc. It also includes application details like putting and getting messages from a queue and the binding of the application with the queue manager. The idea behind the diagram is to have a clear picture of the underlying messaging infrastructure. Thus there may be multiple diagrams to draw details of different aspects to the infrastructure and should not be limited to the use of one diagram in the case of complex architectures.

4.1.1. QMAD Sender-Receiver



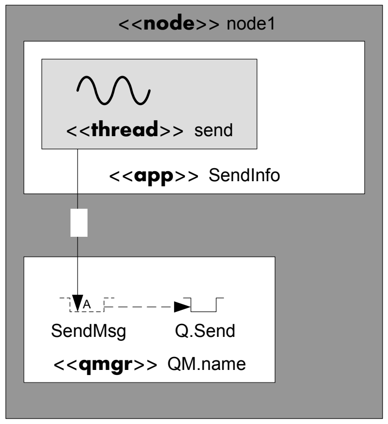
This diagram shows two sender-receiver channel pairs between the queue managers ***QM.1*** and ***QM.2*** and corresponding transmit queues. The queue managers are configured on the nodes ***node1*** and ***node2*** respectively.

4.1.2. QMAD Server-Requester



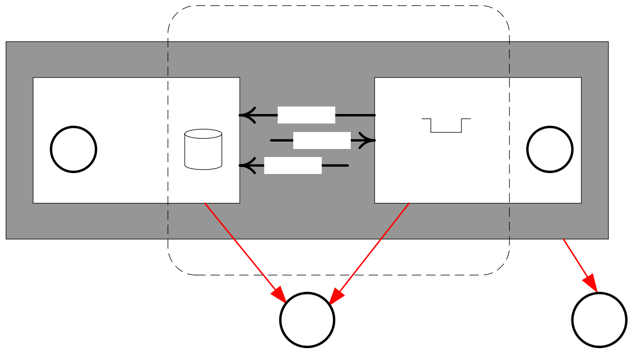
This diagram shows two server-requester channel pairs between the queue managers ***QM.1*** and ***QM.2*** and corresponding transmit queues. The queue managers are configured on the nodes ***node1*** and ***node2*** respectively.

4.1.3. QMAD Application Putting Message



The diagram illustrates an application ***SendInfo*** running on ***node1*** spawns a thread send that puts messages to an alias queue ***SendMsg*** that points to the ***Q.Send*** local queue on the queue manager ***QM.1***.

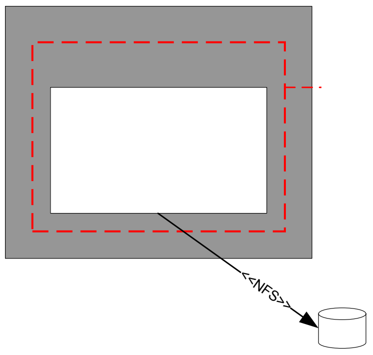
4.1.4. QMAD Cluster and Monitors



The diagram illustrates two queue managers ***QM.1*** and ***QM.2*** on a single node ***node1*** running listeners on ports ***1414*** and ***1515*** respectively. ***QM.1*** is the repository queue manager of the cluster ***CLUS.1***. The queue managers are being monitored by the Candle WMQ Agent and the operating system is being monitored by the Candle OS Agent.

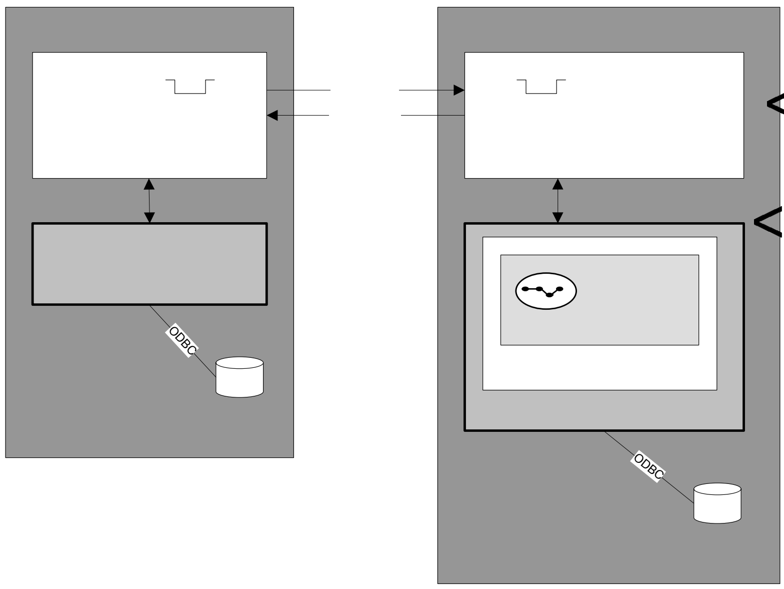
PUT

4.1.5. QMAD Fail-over



The diagram illustrates the queue manager ***QM.1*** defined on a NFS mounted shared file system on ***node2***. The HA resource group RG.QM.1 monitors the queue manager and fails over to ***node1*** in case any of the processes in the resource group goes down. The same queue manager ***QM.1*** is restarted on ***node1*** with the same definition from the NFS mounted file system from the mount point ***/var/mqm/qmgrs/QM!1***.

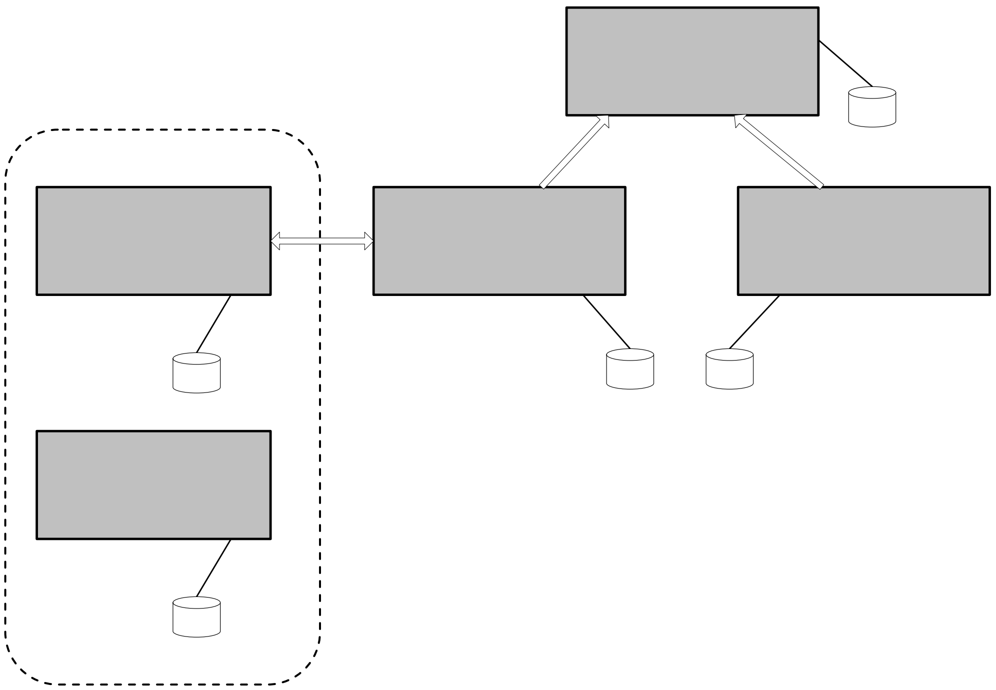
4.1.6. QMAD Broker and Configuration Manager



The configuration manager queue manager ***QM.CM*** is configured on ***nodeCM*** with the configuration manager running on the same node. The configuration manager database is ***DBCM***. The broker queue manager ***QM.BK*** is configured on ***nodeBK*** along with the broker ***BK.1*** with the broker database ***DBBK***. The communication between the queue managers is established by setting up the sender-receiver pairs for each end. The proper transmit queues are set up as shown.

/var/mqm/q

4.1.7. Broker Topology Tree



The above diagram illustrates a broker topology tree with the ***BK.Parent*** broker (with database ***DBBKP***) at the top. This broker has two other brokers configured as its children – ***BK.Child1*** and ***BK.Child2*** (with corresponding databases ***DBBKC1*** and ***DBBKC2***). The broker ***BK.Child1*** is associated with the broker ***BK.1*** (with database ***DBBK1***) at the same level. Brokers ***BK.1*** and ***BK.2*** (with database ***DBBK2***) are in a broker collective ***BKC.Local***.

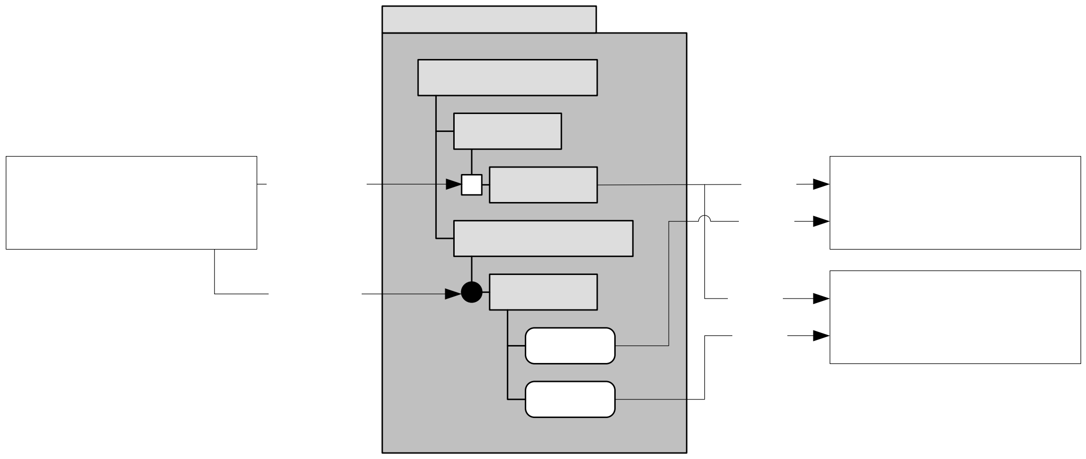
<<**bkc**>> BKC.Lo

<<**broker**>> BK.

DB

DBBK1

4.1.8. Publisher and Subscriber to a Topic Tree



The diagram illustrates a topic tree in the package ***TGame***. The stereotype **<<topic>>** is used to represent a topic package. The root topic in the tree is ***Game***. It has two children – ***Score*** and ***GameUpdate***. The ***Score*** represents a topic where retained publications are sent since the subscribers are interested in only the latest score. The ***GameUpdate*** topic represents the proceedings of the game (commentary) and is sent as event publications to the subscribers immediately. Both the topics have a subtopic ***AvsB*** representing the game where team A is playing team B. The sub-topic ***Game/GameUpdate/AvsB*** has two subscription points – ***english*** and ***spanish***, to publish information in English and Spanish respectively. The publisher application, ***publisher***, publishes local publications for the commentary both in English and Spanish as even publications. It also publishes the score globally as retained publication. There are two subscriber applications – ***subscriberEng*** and ***subscriberSp***. The former subscriber subscribes to the score and the updates in English for the game AvsB and the latter subscriber subscribes to the score and the updates in Spanish for the same game AvsB.

There may be other publishers on the same topics and other subscribers on the same topics who join in temporarily or permanently as well as subscribers on remote (but connected) broker for only the scores of the game AvsB.

<<pub

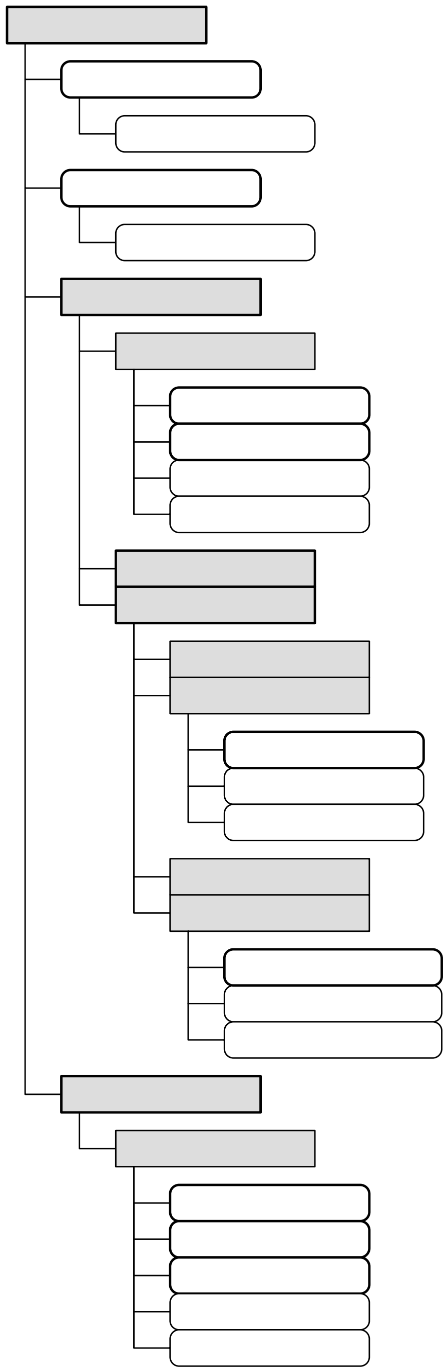
<<**app**>> publisher

<<pub

4.2. Security Profile Diagram

This section illustrates examples for the Security Profile Diagram (SPD) for WMQ objects and WBIMB publish/subscribe topic security components.

4.2.1. WMQ Security



The following SPD shows the WMQ security profile for a particular node ***node1***.

Every security profile diagram starts with the node object indicating access to the complete installation on that node. In the example presented here, ***mqm*** group refers to the WMQ administrator group and the user ***wmqiadm*** is a part of this group. Similarly another system control group is ***mqbrkrs*** for all administrative control of the message broker on the system. The user ***wmqiadm*** is also a part of this group.

Next the security tree indicates the profile for queue manager ***QM.1***. The connect authority for connection to the queue manager is given to the group ***grpA*** in domain ***DA*** and ***grpB*** in domain ***DB***. Also the access is given to the principles ***ron*** and ***guy***. Inside the queue manager ***QM.1***, open and put accesses for the queues ***Q.1*** and ***Q.2*** is given to the group ***grpA*** in domain ***DA*** as well as to the principles ***ron*** and ***guy***; while the open and get accesses for the same queues are given to group ***grpB*** in domain ***DB*** and also to the principles ***arunava*** and ***popi*** both in domain ***DX***.

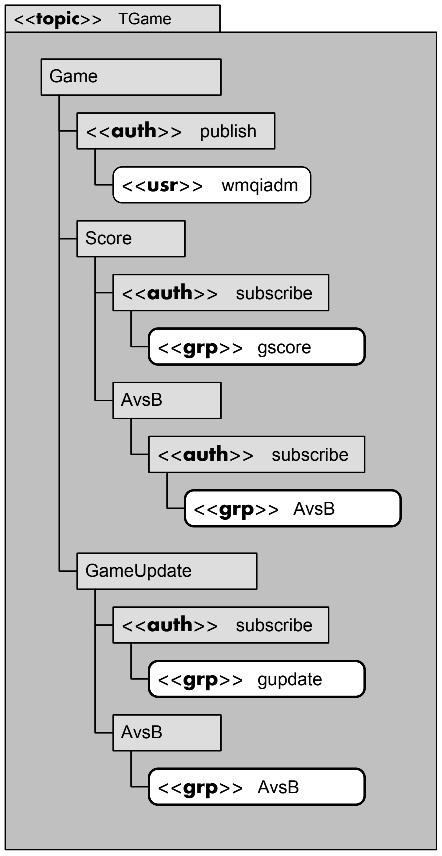
Next ***all*** access to queue manager ***QM.2*** is given to the groups ***grpA*** in domain ***DA***, ***grpB*** in domain ***DB*** and the local group ***grpX*** and to the principles ***ron*** and ***guy***.

<<**node**>> node1

<<**grp**>> mqm

<<**usr**>> wmqiadm

4.2.2. Topic Security

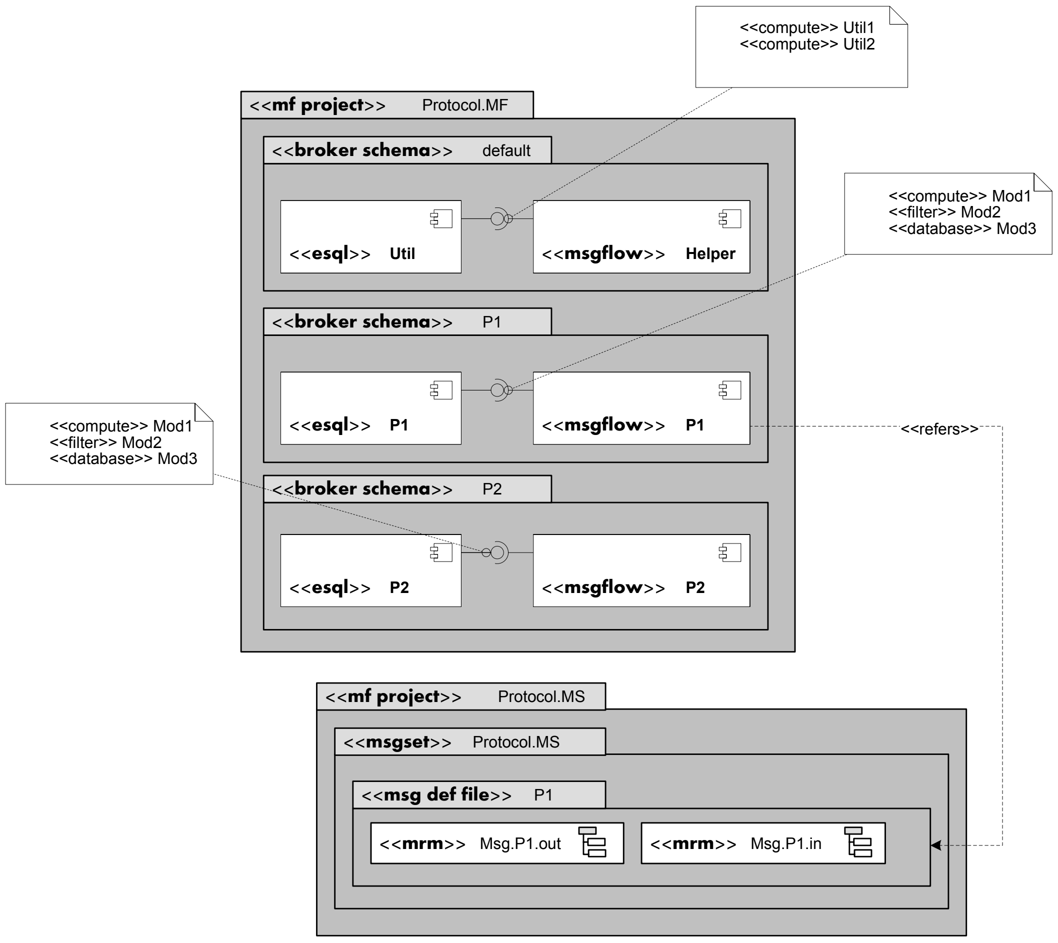


This diagram illustrates the security aspects related to the topics in the topic tree for the game ***AvsB*** (vide 4.1.8) in the topic package ***TGame***.

The user id ***wmqiadm*** has ***publish*** access to the root topic ***Game***. Thus he may publish to any sub-topic of ***Game*** as well. The group ***gscore*** has ***subscribe*** access to the topic ***Game/Score***. The group ***gupdate*** has ***subscribe*** access to the topic ***Game/GameUpdate***. The group ***AvsB*** has ***subscribe*** access to the topics ***Game/Score/AvsB*** and ***Game/GameUpdate/AvsB***.

4.3. Broker Component Diagram

A Broker Component Diagram (BCD) represents a schematic component diagram of the broker packages and components. The artifacts are derived from UML 2.0 component diagram model and extended to represent the broker components.

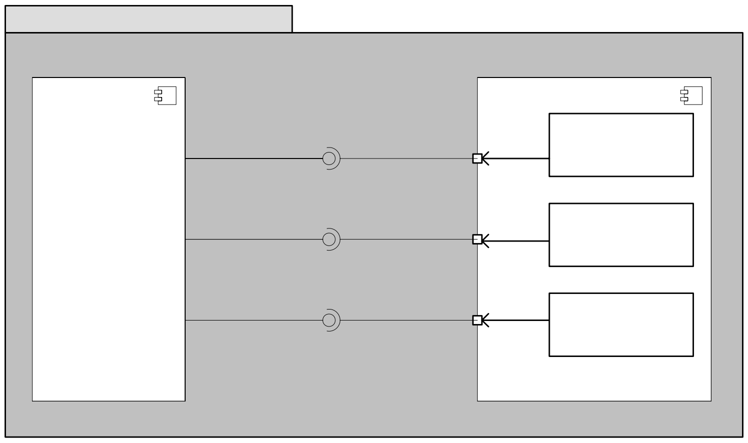


A Broker Component Diagram (BCD) like any other component diagram is a schematic representation of the components of a broker development environment. The BCD example illustrated above shows a basic message flow project ***Protocol.MF*** and its components along with a related message set project ***Protocol.MS*** and its components.

Inside the message flow project different components are aggregated under broker schemas. The stereotype **<<mf project>>** is used for the package representation of the message flow project. The stereotype **<<broker schema>>** is used for the package representation of the broker schema inside a message flow project. Every message flow project has a default package which maps to the base project folder. All other broker schemas are packaged under their own folders. Inside the broker schema the ESQL code is represented as a component with a set of interfaces or modules. The stereotype **<<compute>>** represents a compute module, the stereotype **<<filter>>** represents a filter module and stereotype **<<database>>** represents a database module. Interfaces can also be global functions and procedures with the stereotypes **<<function>>** and **<<procedure>>** respectively. There are two different types of components in a message flow project – ESQL components comprising of ESQL code and Message Flow components comprising of the flows and are represented by the **<<esql>>** and **<<msgflow>>** stereotypes respectively. The components are distinguished by the UML component symbol.

The modules may be represented inside annotations with corresponding labels as shown. This helps in better organization in the diagram. If the number of interfaces is relatively small they can be represented simply near the interface notation.

The message set project is represented with the package notation and the stereotype **<<ms project>>**. The message set is represented with the stereotype **<<msgset>>**. Only one message set is permitted in a message set project. There can be multiple message definition files inside the message set where the message definitions are stored. The message definition file is represented with the stereotype **<<msg def file>>**. Inside each message definition file multiple messages are defined. The messages inside a message definition file are represented as shown in the diagram with a rectangle and a notation of a message tree inside the rectangle. The stereotype associated with the message definitions is **<<mrm>>** for the broker which may be shown optionally. Only MRM type of messages may be represented in this manner since the message set is an integral part of the MRM model for the message broker. It is to be noted that messages are not components since they are templates and do not provide interfaces or call an interfaces.

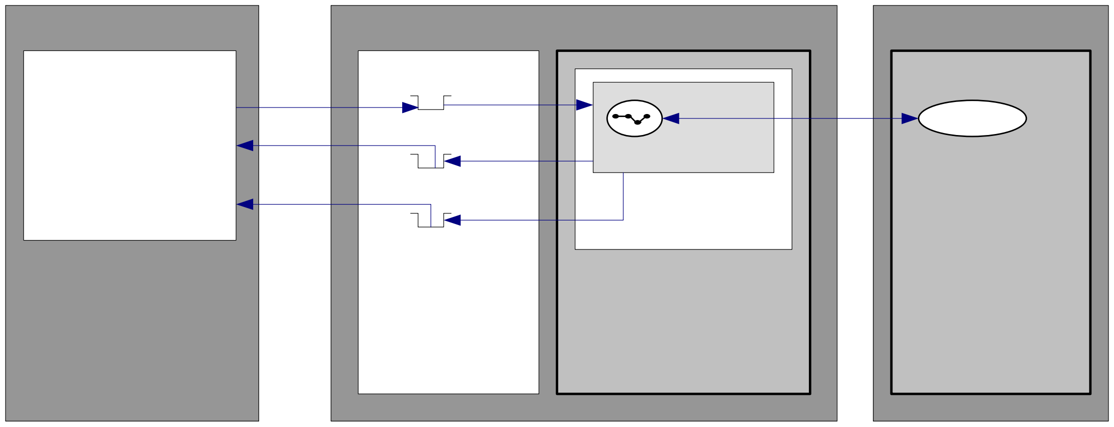


The diagram on the left illustrates the details of the broker schema ***P2***. The ESQL component ***P2*** provides three interfaces – ***Mod1*** of type compute module, ***Mod2*** of type filter module and ***Mod3*** of type database module. The diagram shows the delegation of the interfaces inside the message flow component ***P2***. The instance of Compute Node ***Mod1*** uses the interface Mod1 provided by the ESQL component ***P2***. Similarly, the Mod2 interface is used by the Filter Node instance Mod2 and the Mod3 interface is used by the Database Node instance Mod3. It is to be noted that a compute module interface may only referenced from a Compute Node, a filter module interface can only be referenced from a Filter Node and a database node module can only be referenced from a Filter Node.

4.4. Message Interaction Diagram

A Message Interaction Diagram (MID) is a simple schematic representation of the messages, message types and protocols in the entire or parts of a subsystem. The diagram introduces two notations – a single ended arrow representing asynchronous communication and a two-ended arrow representing a synchronous communication. The diagram may be used in conjunction with the Queue Manager Architecture Diagram (QMAD), thus, making it a powerful tool to even illustrate communication with queue managers and applications. However, the MID should be kept simple and not to represent the architectural complexity of the subsystem.

An example of a message interaction diagram is shown in the diagram below.

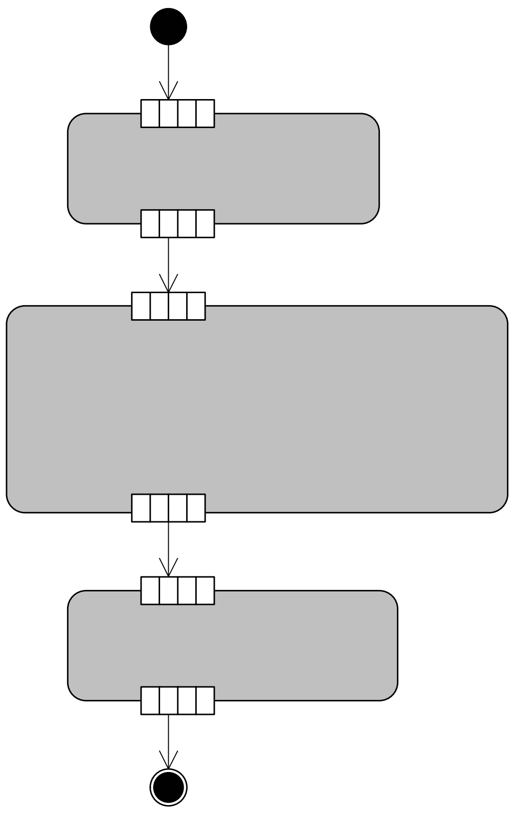


The following is a list of the parameters that can be represented on the arrow showing message flow:
Message Format: XML, MRM (including the message name), SOAP, String, IDOC, etc.
Transport Protocol: MQ, HTTP, X.25, Socket, FTP, TCP, SNA, etc.
Protocol Parameters: Port, Packet Size, Keep Alive, etc.

<<node>> host0

4.5. Flow Activity Diagram

A Flow Activity Diagram (FAD) represents a sequence trace of a message flow. The artifacts are derived from UML 2.0 sequence diagram and extended to represent a message flow.



The above diagram shows an Activity Diagram for a simple message flow the message is picked up by the message flow from the ***Q.IN*** queue and parsed according to the MRM message format ***Msg.in***. The MQ Input Node then calls the ***Transform Msg*** Compute Module associated with the Compute Node where the message is re-parsed and transformed into the ***Msg.out*** format as defined by the XML schema in the XML domain. The message is then put into the ***Q.OUT*** queue. The table below represents the different stereotypes and the nodes and modules it is associated with. As a general rule all nodes inside the message flow are represented as an expansion region. The stereotype determines the type of the node or module and the expansion node models the Input and Output terminals with the label representing the name of the terminal. The expansion region may be decomposed into its activities if required.

The loop notation is not defined in UML 2.0 specifications and hence we use the notation similar to the Flowchart loop notation as demonstrated in the next section.

Stereotype	**Node : Module**
<<compute>>	Compute Node : Compute Module
<<filter>>	Filter Node : Filter Module
<<database>>	Database Node : Database Module
<<MQ in>>	MQ Input Node
<<MQ out>>	MQ Output Node
<<MQ reply>>	MQ Reply Node
<<publish>>	Publication Node
<<MQe in>>	MQ Everyplace Input Node
<<MQe out>>	MQ Everyplace Output Node
<<SCADA in>>	SCADA Input Node
<<SCADA out>>	SCADA Output Node
<<HTTP in>>	HTTP Input Node
<<HTTP req>>	HTTP Request Node
<<HTTP reply>>	HTTP Reply Node
<<RT in>>	Real Time Node
<<RT opt flow>>	Real Time Optimized Flow Node
<<in term>>	Input Terminal Node
<<out term>>	Output Terminal Node

<<MQ in

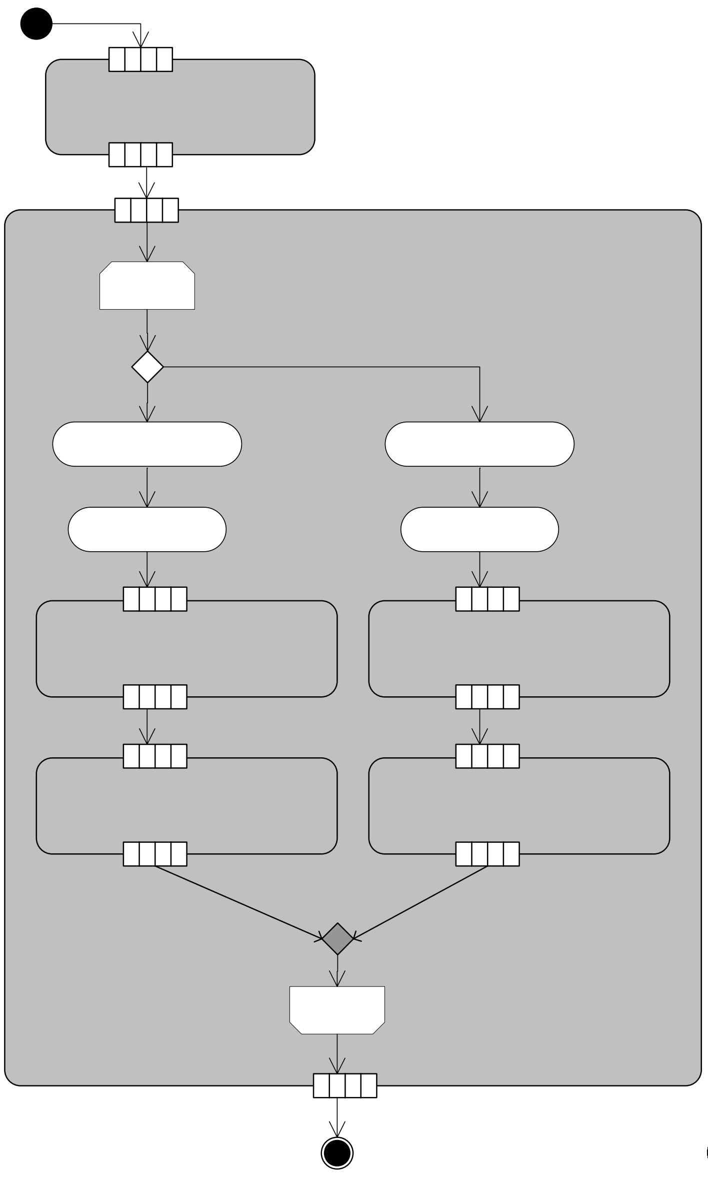
Parse message
Msg.in from Q.IN

Out

In

<<data del>>	Data Delete Node : Data Delete Map
<<data insert>>	Data Insert Node : Data Insert Map
<<data update>>	Data Update Node : Data Update Map
<<extract>>	Extract Node : Extract Map
<<mapping>>	Mapping Node : Data Map
<<warehouse>>	Warehouse Node : Warehouse Map
<<aggr control>>	Aggregate Control Node
<<aggr req>>	Aggregate Request Node
<<aggr reply>>	Aggregate Reply Node
<<check>>	Check Node
<<flow order>>	Flow Order Node
<<rcd>>	Reset Content Descriptor Node
<<route>>	Route To Label Node
<<label>>	Label Node
<<passthrough>>	Passthrough Node
<<trace>>	Trace Node
<<throw>>	Throw Node
<<try catch>>	Try Catch Node

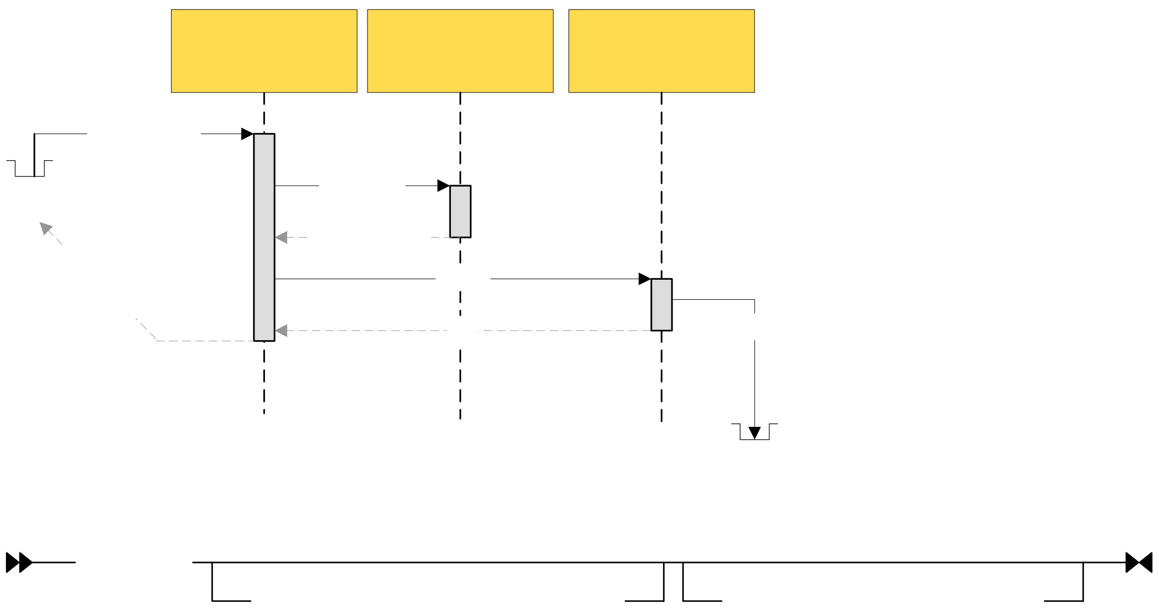
4.5.1. FAD Branching



The above diagram shows a message flow that accepts a message in the ***Msg.in*** format and sends it to the Compute Module ***Process Msg***. The Compute Module checks whether each of the rows passed in the message is either of types A in a loop. If the row is of type A the row is parsed into ***Msg.RowA*** format, set the destination of the label to ***RowA*** and propagated to the ***Out*** terminal of the Compute Node that is wired to a Route To Label Node. Based on the label information in the destination the message is routed to the Label Node with label ***RowA***. The Route To Label Node is omitted in the diagram as the Compute Module loops through the rows and propagates messages in the corresponding format. If the row is not of type A, then the row is parsed into the ***Msg.RowB*** format and in the same way passed on to the Label Node with label ***RowB***. The Label Nodes ***RowA*** and ***RowB*** are wired to the MQ Output Nodes ***Q.RowA*** and ***Q.RowB*** that put the individual row messages in ***Q.RowA*** and ***Q.RowB*** queues in the ***Msg.RowA*** and ***Msg.RowB*** formats respectively.

4.6. Flow Sequence Diagram

A Flow Sequence Diagram (FSD) represents a sequence trace of a message flow. The artifacts are derived from UML 2.0 sequence diagram and extended to represent a message flow.



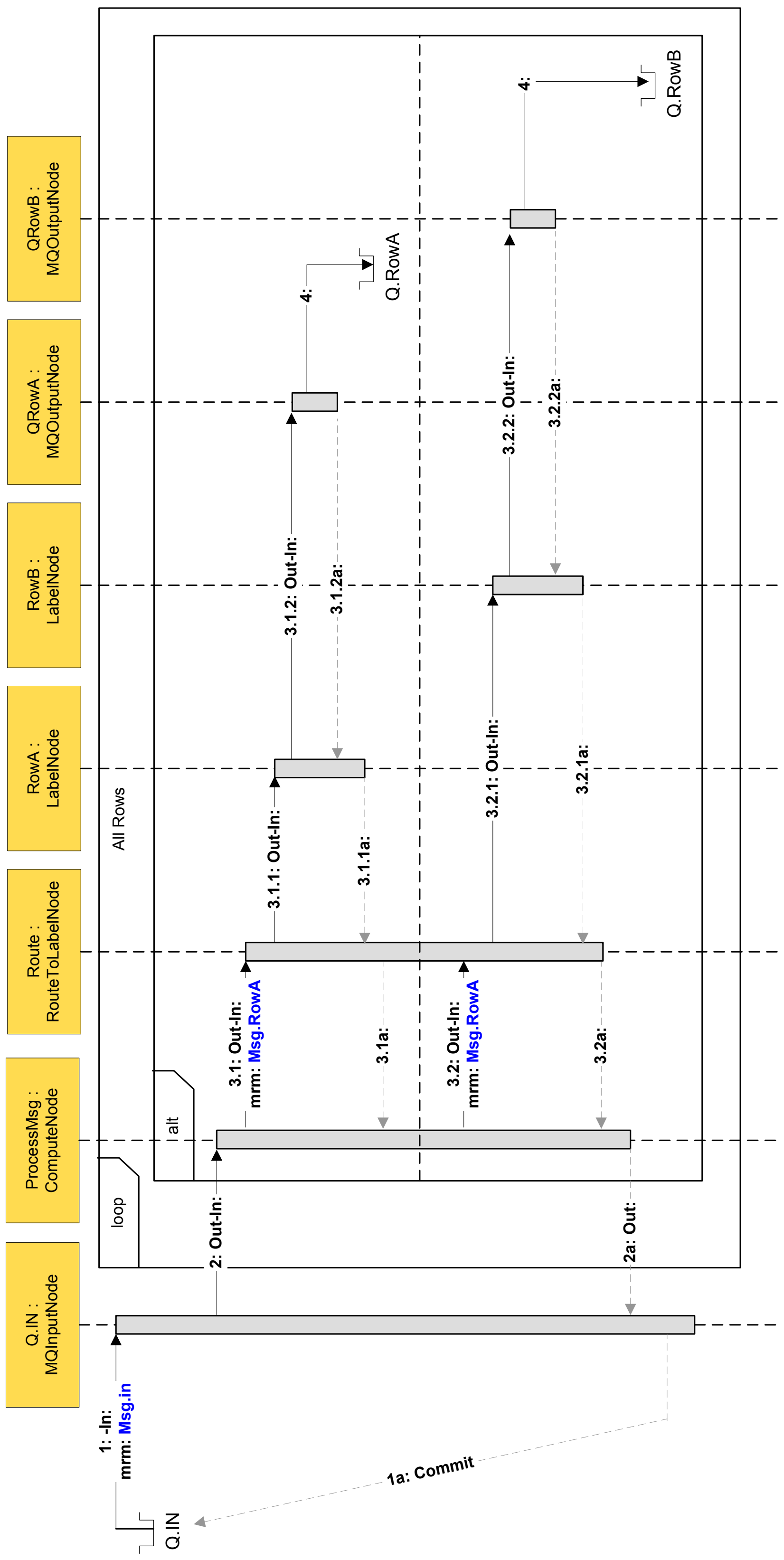
The above diagram shows a very simple flow with three nodes – the MQ Input Node picks up messages in the format ***Msg.in*** from the queue ***Q.IN***, the Compute Node ***Transform Msg*** that converts the message format into an XML with the schema described in ***Msg.out***. After that it is propagated to the MQ Output Node that puts the message in the queue ***Q.OUT***. The syntax for the messaging between nodes is given below:

Seq # ⇒ Sequence number for the FSD
Out Terminal ⇒ The output terminal of the node
In Terminal ⇒ The input terminal of the node
Msg Domain ⇒ Message Domain for the message (e.g. ***xml***, ***mrm***)
Message ⇒ Message definition

For the MQ Input Node the message is being picked up from the queue ***Q.IN*** and is propagated to its ***In*** input terminal. So there is no output terminal reference. The same message is propagated from the ***Out*** output terminal of the MQ Input Node to the ***In*** input terminal of the Compute Node. Here the message is absent as the message format remains unaltered. The Compute Node returns a message in the xml domain. The same message is propagated to the MQ Output Node and gets put to a queue ***Q.OUT***.

The naming convention of the sequence numbers is starting from '1' and incremented progressively from node to node and the corresponding returns are referred with the same number with an alphabetic character appended to it. The alphabetic reference for the returns start from 'a' and is incremented for the alternative responses sent. E.g. if the Compute Node throws an exception there would exist a separate return path ***2b***.

4.6.1. FSD Branching



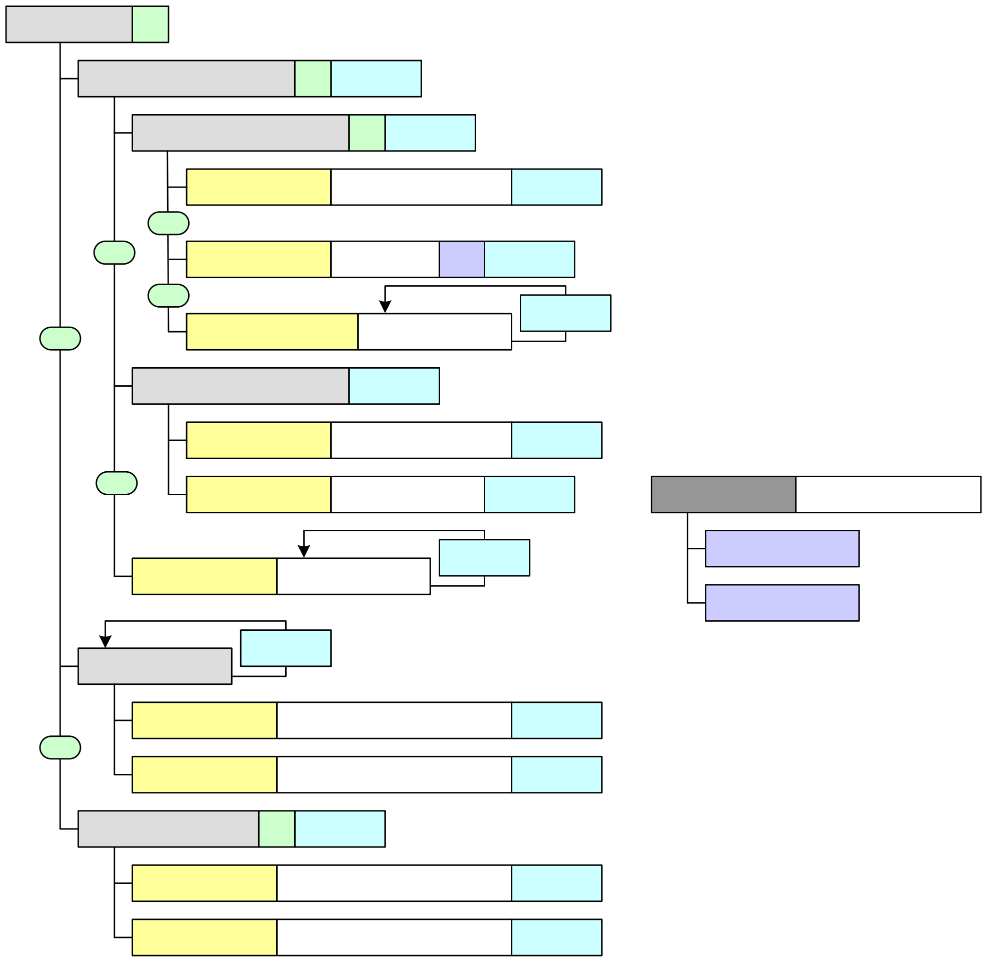
The message flow in the diagram identifies each row of the incoming data in a message containing multiple rows, parses it in two different formats as a single message per row and puts them in two different queues.

This section illustrates an example for a message flow that accepts a message in the MRM domain and of type ***Msg.in*** and passes it to a Compute Node ***ProcessMsg*** that iterates through each row in the message and propagates to the ***Out*** output terminal which is wired to the Route to Label Node ***Route***. Based on the type of the row data either of type A or type B the message is parsed into either ***Msg.RowA*** or ***Msg.RowB*** and the destination list is populated for Label Nodes ***RowA*** and ***RowB*** respectively. Thus, it has two alternate paths – one going through the ***RowA*** node and ***QRowA*** node to the queue ***Q.RowA*** and the other going through the ***RowB*** node and ***QRowB*** node to the queue ***Q.RowB***.

4.7. Message Specification Diagram

A Message Specification Diagram (MSD) lays down a detailed data structure and is independent of the type of language or parser it is implemented with. It is to provide a schematic diagram of the data structure and thus gives a visual perception of very complex data types used in enterprise applications with fixed length, delimited, tagged, grouped formats.

4.7.1. MSD Msg.in



Every message has its own data structure and is represented by a complex type. The ***Msg.in*** message is primarily divided into three parts ***Headers***, ***Data*** and ***Footer*** delimited by '***$***'.

Headers is a complex type consisting of elements delimited by '***;***'. The first element is a complex type ***HeaderA*** delimited by '***,***' consisting of the elements ***HeaderAFldA*** – a mandatory string of length 10 occurring only once, ***HeaderAFldB*** – a mandatory string of length 1 and must have a value of '***A***' occurring only once, followed by an optional set of fields ***HeaderAFldReserved*** up to 50 of data type blob. The second element in the ***Headers*** structure, ***HeaderB*** is a fixed length complex type consisting of the

Typ.in: Msg.in $

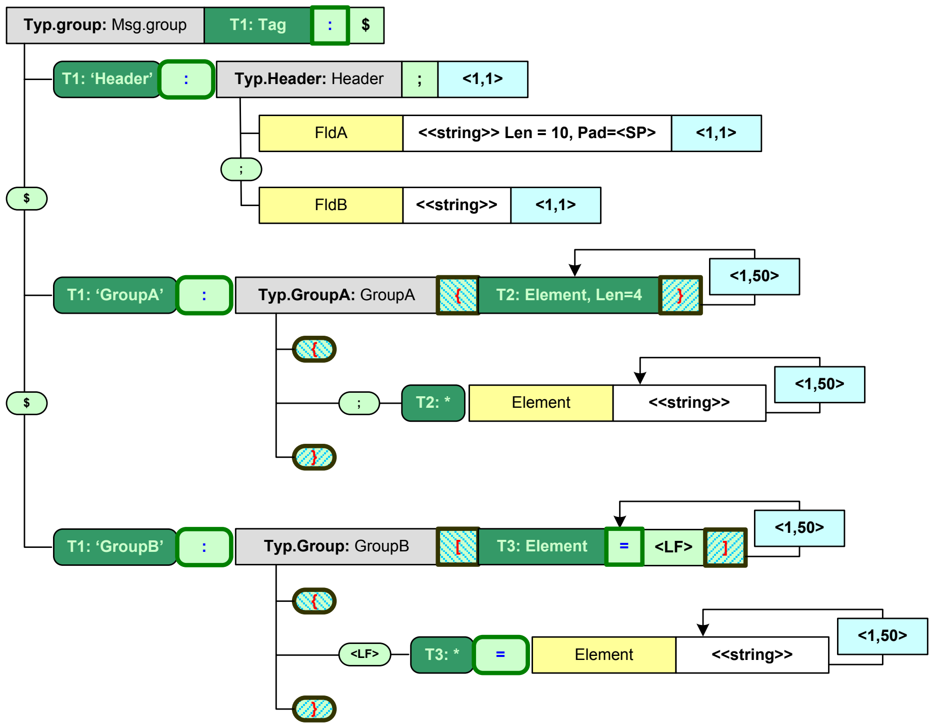
Typ Headers: Headers

elements ***HeaderBFldA*** – a mandatory string of length 10 and ***HeaderBFldB*** – a mandatory field of length 10 with values as listed by the list ***List1***. The list consists of '***Option1*** ' and '***Option2*** ' indicating that the only permissible values for field ***HeaderBFldB*** is either '***Option1*** ' or '***Option2*** '. The third element in the ***Headers*** structure, ***HeaderReserved*** is a set of optional fields up to 50 of data type blob.

Data is a mandatory complex type with a fixed length data structure consisting of an integer of length 10 and padding character '***0***', ***DataFldA***, and a decimal field ***DataFldB*** of length 10 and precision 2. The structure be repeated an infinite number of times.

Footer is a mandatory complex type element delimited by '***;***' and consists of ***FooterFldA*** of type integer and a length of 10 with padding '***0***' and ***FooterFldB*** of type string.

4.7.2. MSD Msg.group



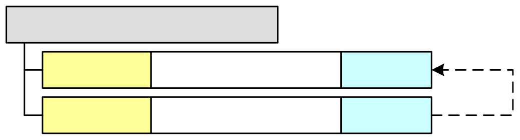
The diagram in this section illustrates a tag-delimited message containing group indicators. The message is of the complex element type ***Msg.group*** containing a tag with identifier ***T1*** (for reference purposes in the MSD) that is delimited by the tag-data separator '***:***' and each element is separated by the delimiter '***$***'.

The first element in the message is a complex structure that is identified by the instance of the tag type ***T1*** and the tag string ***Header***. The ***Header*** structure consists of two mandatory fields delimited by '***;***'. FldA is a string of length 10 and padding as spaces and ***FldB*** is a string.

The second element is a complex element that is identified by the instance of the tag type ***T1*** and tag string ***GroupA***. The complex type ***GroupA*** consists of a set of tag-delimited elements enclosed in a group identifier '***{***' and group terminator '***}***'. The tag type is identified by ***T2*** and is of fixed length of 4. Each element in the group is delimited by '***;***' and the minimum number of elements in the group is 1 and the maximum may be 50. Each of the elements is identified by the anonymous instance of the tag type ***T2*** read in from the data supplied (the first 4 characters) and the element is parsed into a string data type.

The third element is a complex element that is identified by the instance of the tag type ***T1*** and tag string ***GroupB***. The complex type ***GroupB*** consists of a set of tag-delimited elements enclosed in a group identifier '***{***' and group terminator '***}***'. The tag type is identified by ***T3*** and is delimited by the tag-data separator '='. Each element in the group is delimited by the line-feed byte ***x'0A'*** and the minimum number of elements in the group is 1 and the maximum may be 50. Each of the elements is identified by the anonymous instance of the tag type ***T3*** read in from the data supplied (until the parser encounters the tag-data separator '=') and the element is parsed into a string data type.

4.7.3. MSD Msg.LengthRef

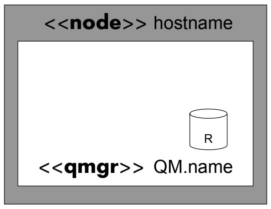


The fixed length complex structure for the message illustrated in the diagram, ***Msg.LengthRef*** consists of two mandatory fields – the first an integer field ***length*** of length 10 and the second a blob type field ***blob*** of length equal to the value in the ***length*** field. Thus the length of the ***blob*** field is determined as runtime from the value of the length and should be indicated by the stereotype **<<refers>>**.

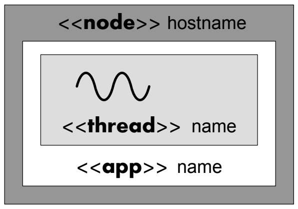
Typ LengthRef: Msg LengthRef

5. Legend

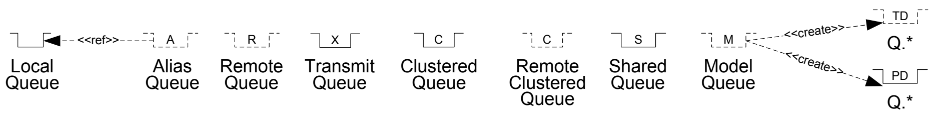
Queue Manager & Node



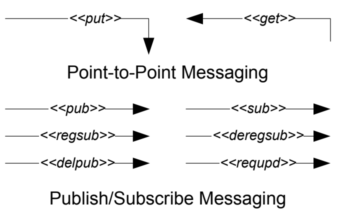
Application



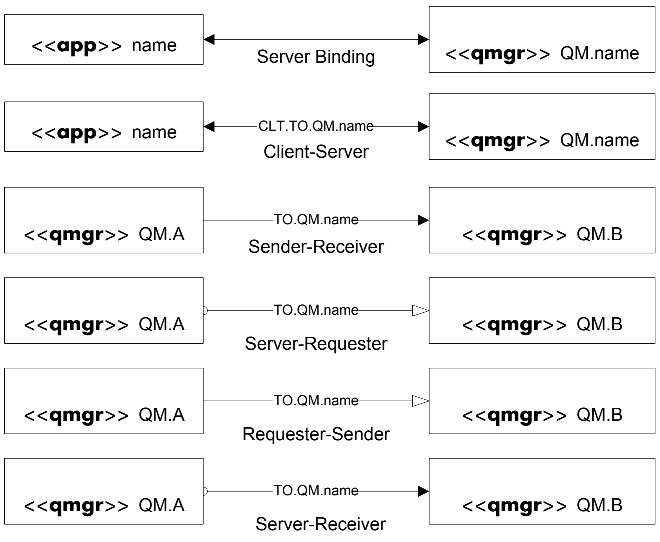
Queue



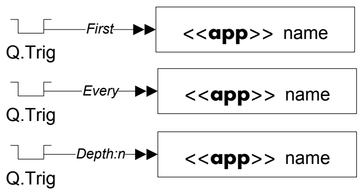
Queue Operation



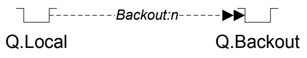
Channel



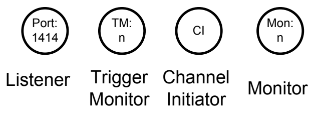
Trigger



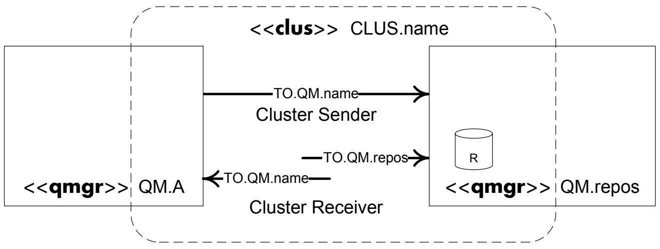
Backout



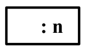
Daemon



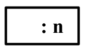
Cluster



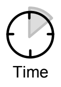
Exit



Event



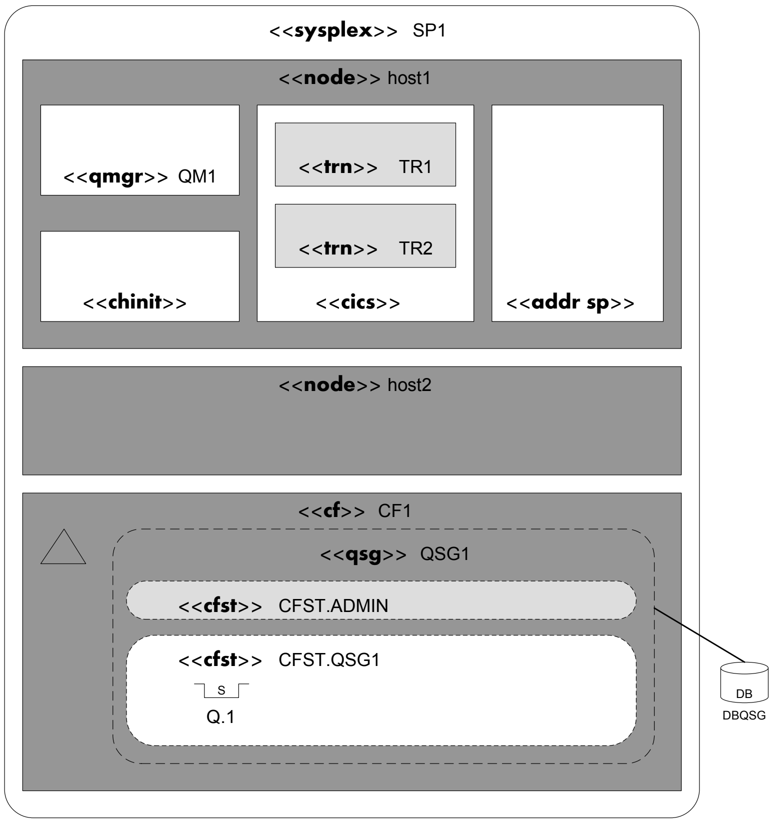
Expiry



Arunava Majumdar
arunava@us.ibm.com

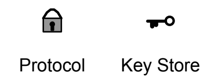
Guy Hochstetler
guyh@us.ibm.com

Z/OS Notation and Queue Sharing Group

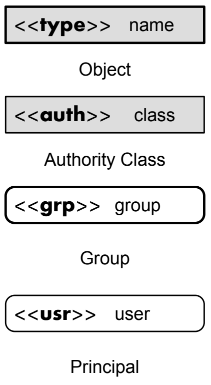


Security

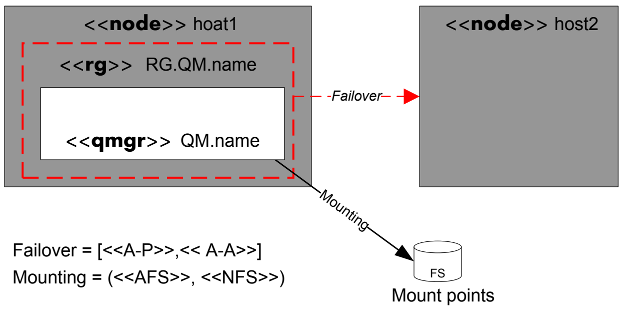
SSL



OAM

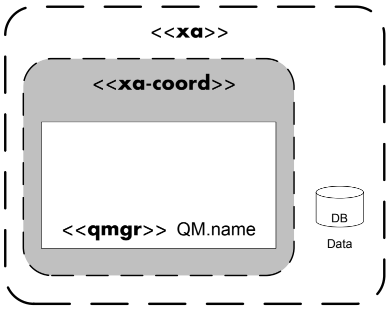


HA Failover

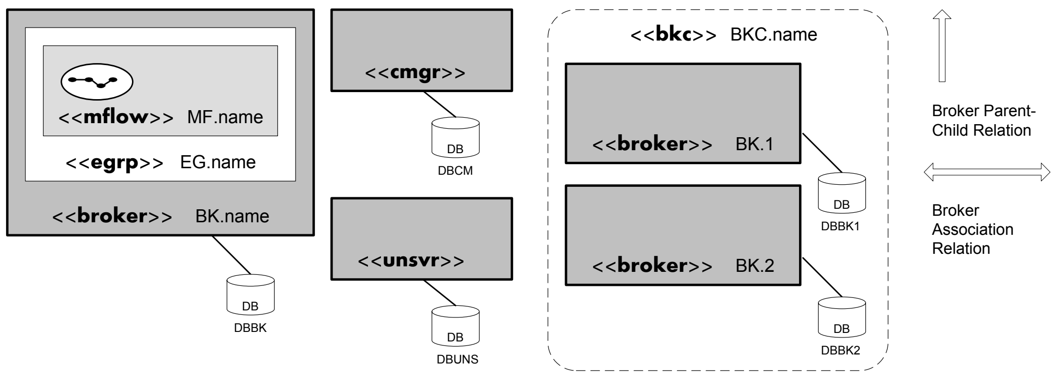


Failover = [<<A-P>>,<< A-A>>]
Mounting = (<<AFS>>, <<NFS>>)

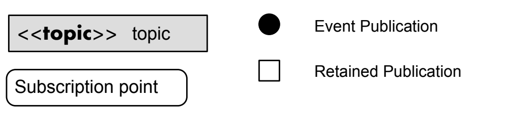
XA Coordination



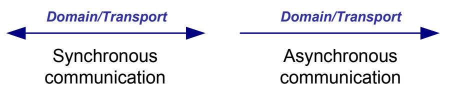
Broker



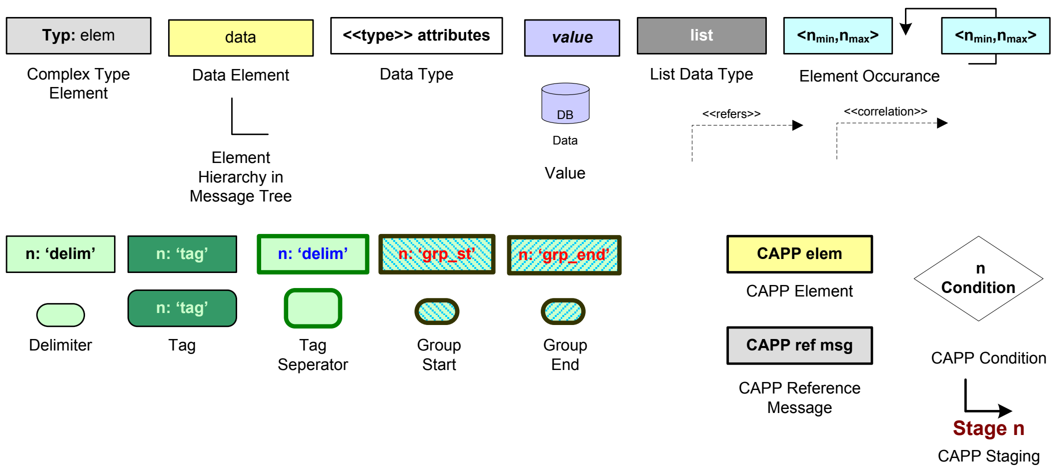
Pub/Sub



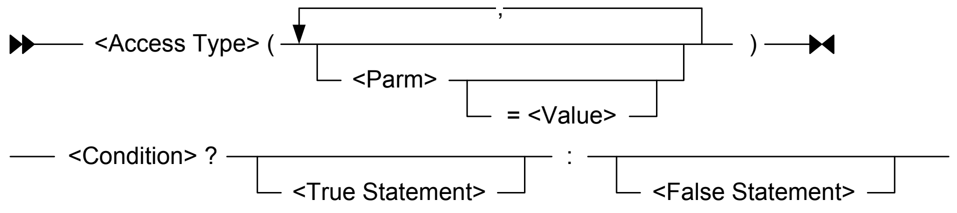
Message Interaction



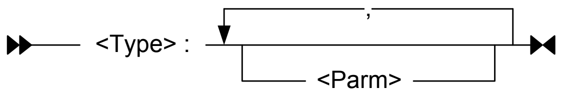
Message Specification



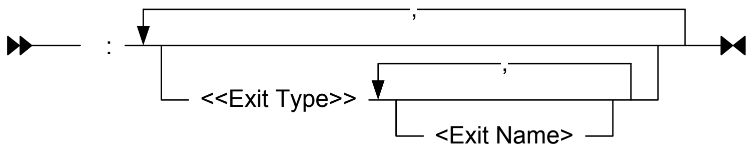
Syntax – Queue Access



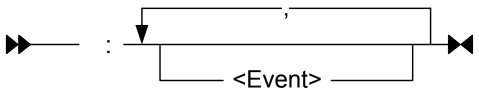
Syntax – Trigger, Backout, Daemon



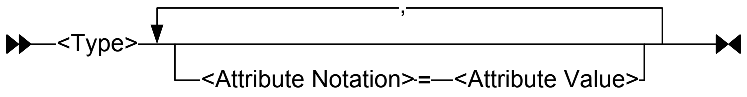
Syntax – Exit



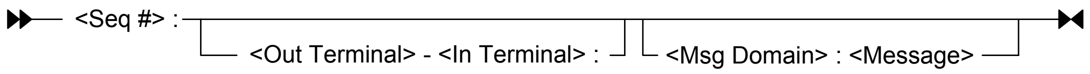
Syntax – Exit



Syntax – Element Attribute



Syntax – Message in Flow Sequence Diagram



6. Conclusion

This paper is an attempt to unify and standardize notation for messaging and workflow systems. Our first release is an extension from the Websphere MQ Network Design Notation paper by David Grainger released as a support pack for WMQ. Our next steps include creating an UML profile for the notations and IDE tools that would translate the models into deployment scripts and message flows. While further standardization and UML profile and IDE tooling development are on its way, we believe the release of this paper along with the Microsoft Visio 2003 version of the stencil would immensely assist messaging experts to move ahead in designing and architecting solutions using these standard set of notations.

7. Bibliography:

1. SC34-6059-03 – WebSphere MQ Intercommunication
2. GC34-6057-01 – WebSphere MQ Messages
3. SC34-6055-03 – WebSphere MQ Script (MQSC) Command Reference
4. SC34-6079-01 – WebSphere MQ Security
5. SC34-6068-01 – WebSphere MQ System Administration Guide
6. SC34-6061-02 – WebSphere MQ Queue Managers Clusters
7. WMQ Support pack md08 – WebSphere MQ Network Design Notation
8. OMG 03-08-02 – UML 2.0 Superstructure specification
9. The Unified Modeling Language Reference Manual by James Rambaugh, Ivar Jacobson and Grady Booch