WebSphere MQ - Network Design Notation Version 1.0

8th August 2002

David Grainger
IBM Software Group

mailto:dgrainge@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, August 2002

This edition applies to Version 1.0 of *WebSphere MQ Network Design Notation* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

Trademarks and Service Marks

Preface

By the nature of message-oriented middleware, WebSphere MQ is often deployed into customer environments that exhibit significant technical complexity.

A basic functional requirement on any messaging infrastructure is the achievement of connectivity between a prescribed set of applications. But as we look beyond this to the non-functional requirements placed upon a real infrastructure, complexity arises from the influence of quality-of-service.

The designer has to know how to build a message-queuing network that will offer the right balance of *availability* (of messages and applications), *performance* (message throughput and timely delivery), *manageability* and *security.* Further, the design must exhibit *scalability* if all of these qualities are to be preserved in the face of growing demands on the infrastructure.

Achieving the best balance can represent a significant amount of effort on the part of the designer, but the essence of this design thinking can be easily lost in the complexity of the resulting network. It is in this sense that diagrams can play a key role, capturing concepts, decisions and details efficiently, while helping to reduce ambiguity and uncertainty.

In this document, we propose a notation for diagrams of MQ-based messaging networks, and introduce conventions for how to use this notation to communicate designs and ideas.

We hope that in this way, we can encourage better design practice throughout the WebSphere MQ technical community.

Acknowledgements

We would especially like to acknowledge the work of Benjamin Zhou (*WebSphere MQ SupportPac MD04*) where he documented a basic symbol set, on which this network design notation builds.

We also gratefully acknowledge the following people who have made provision in their own time to review and contribute to this proposal: Emma Allfree, David Arnold, Alan Beasley, Jason Edmeades, Emir Garza, Anthony O'Dowd, Tim Pickrell, Jamie Roots, Jon Stone, Mark Taylor, Graham Wallis, Ewan Withers.

Chapter 1. Introduction

Overview

In this document we propose a notation for diagrams of MQ networks. The notation consists of a set of symbols and some conventions for how they are used to represent artifacts in a messaging infrastructure.

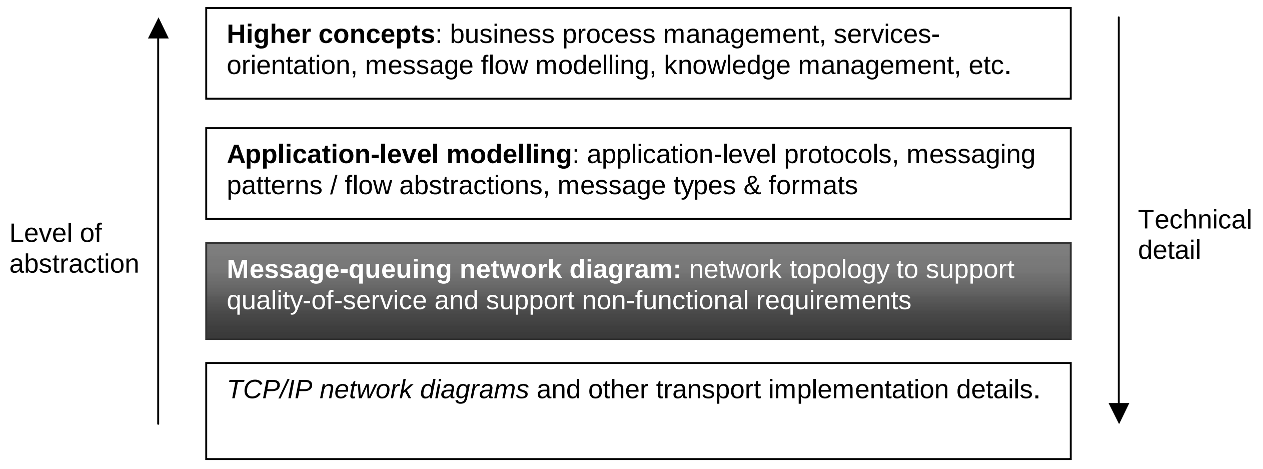
This document is intended as a guide to using the notation, rather than a manual of *how* to design an MQ network (although along the way, we do cover the odd useful design concept!)

We have attempted to design the notation so that it remains easy and flexible to use in the context of technical discussions amongst users of WebSphere MQ. That means the notation has to be fit for manual use on whiteboards and other media used in such discussion.

We have also developed a Visio stencil to accompany this proposal. The stencil contains all of the major notation symbols, which for Visio users should considerably simply the task of drawing and maintaining network diagrams for use in technical documentation.

Scope of Modelling

Considering the stack layers in a conceptual infrastructure, we envisage this notation being used at a specific level of design as represented by the shaded layer below:


Level of abstraction
Higher concepts: business process management, services-orientation, message flow modelling, knowledge management, etc.
Application-level modelling: application-level protocols, messaging patterns / flow abstractions, message types & formats
Message-queuing network diagram: network topology to support quality-of-service and support non-functional requirements
TCP/IP network diagrams and other transport implementation details.
Technical detail

When thinking about messaging infrastructures, it is often helpful to imagine how a high-level design that achieves application connectivity can be transformed into a more detailed network design that also satisfies the non-functional requirements.

One of the uses of this notation is to capture the state of the design before and after these non-functional requirements have been injected. In this way, we hope to isolate those decisions made in pursuit of quality-of-service, while retaining traceability of the design back to the basic connectivity requirements it satisfies.

Use of the notation is not restricted to solution design, but extends to any stage of a project where design information needs to be communicated.

Object Naming in the Notation versus MQ Naming Conventions

Before explaining the notation itself, a brief word on naming.

Experienced MQ users will appreciate the importance of well planned naming conventions when deploying and operating large messaging networks*.

The chief motivation is the need to protect users of the infrastructure from changes but another reason for the emergence of naming conventions stems from the constraints imposed by MQ itself. An example is the 4-character wide namespace for z/OS queue managers (arising from the fact that each is implemented as a z/OS subsystem.)

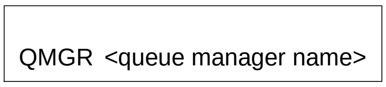
In this notation, however, we do not rigidly follow the naming conventions imposed by implementation concerns, but instead use more expressive naming to improve the readability of network design diagrams.

However, we recognize that in many cases it will be necessary to observe deployment naming conventions, especially when diagrams are targeted at systems administrators and operations specialists.

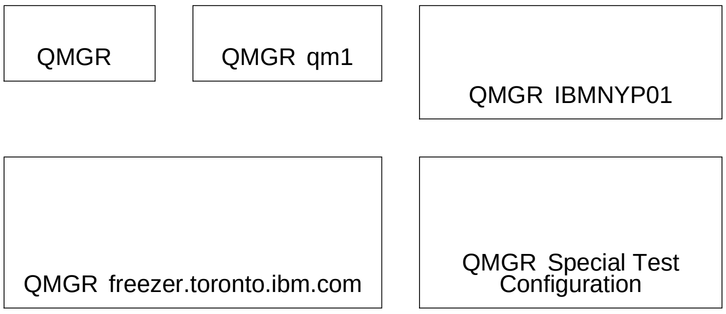
Naming Examples

For the majority of the symbols in this notation, the general form of an object name is:

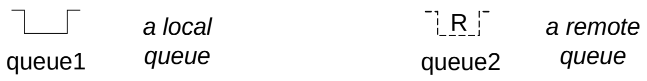
<**OBJECT TYPE**> [<**object name**>]



Taking the **queue manager** symbol, here are some examples with varying degrees of expressiveness:



The same naming form applies to many of the symbols in this notation. The main exceptions are the queue symbols, where the type is indicated by the symbol itself, e.g.:



For message PUT or GET operations, it is sometimes necessary to be explicit about the queue (and perhaps queue manager) that is the target of the operation. For these cases, we have adopted the following convention:

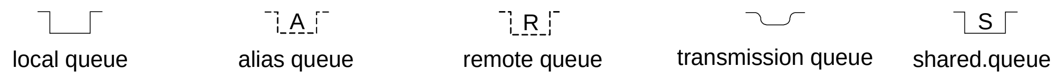
<**OPERATION**> <**queue name**> or <**OPERATION**> <**queue name**>@<**queue manager**>

```
PUT my.queue                           // “my.queue” is a clustered queue
GET requests                           // “requests” is z/OS shared queue
PUT somequeue@my.queue.manager         // app specifies queue manager
PUT somequeue@my.queue.manager.alias   // or more likely, alias qmgr
```

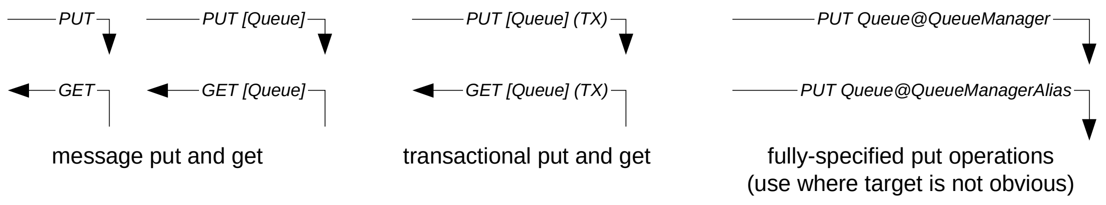
*note: Information on MQ naming practices can be found in the product documentation, and in resources such as *SupportPac MD01 - MQSeries Standards and Conventions*.

Chapter 2. Network Design Notation Legend

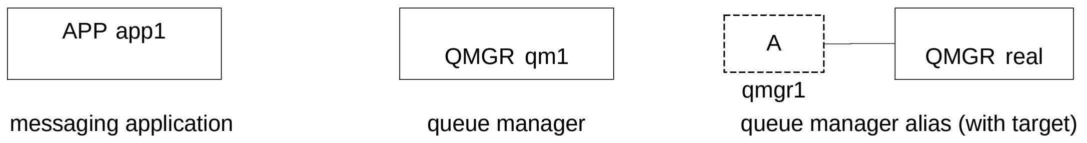
queues



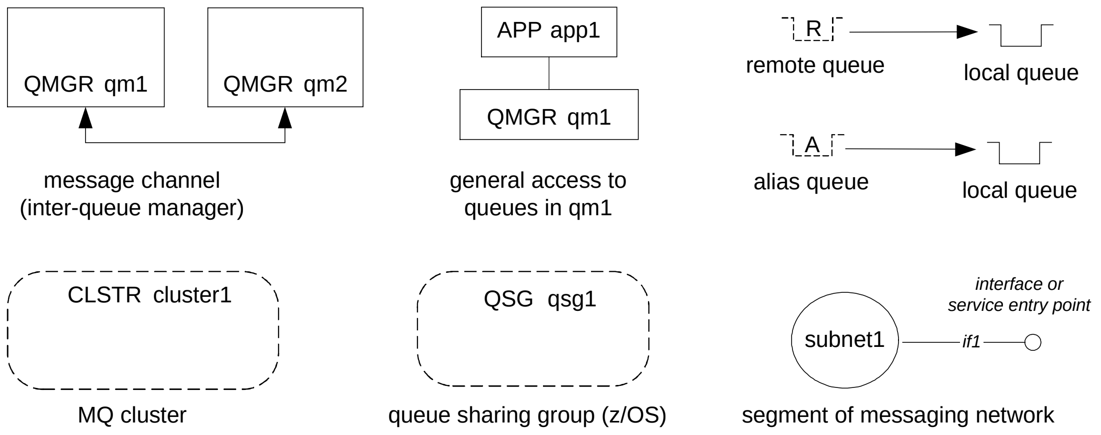
messaging operations


PUT
PUT [Queue]
PUT [Queue] (TX)
PUT Queue@QueueManager
GET
GET [Queue]
GET [Queue] (TX)
PUT Queue@QueueManagerAlias
message put and get
transactional put and get
fully-specified put operations
(use where target is not obvious)

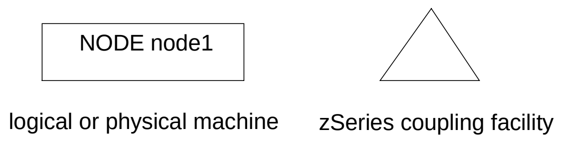
applications & queue managers



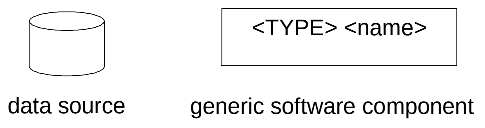
connectivity & grouping



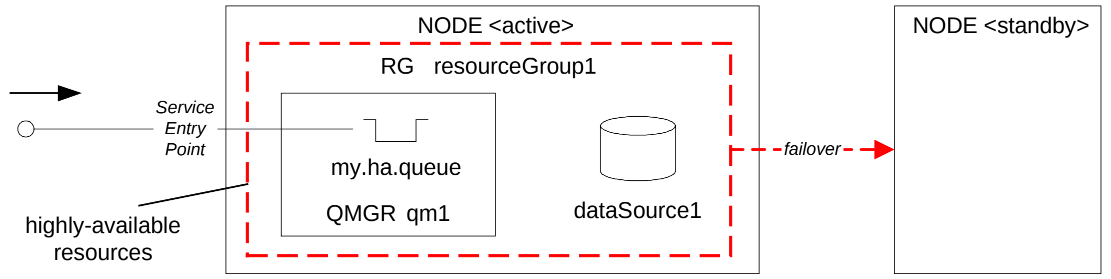
physical infrastructure



miscellaneous



high availability configurations

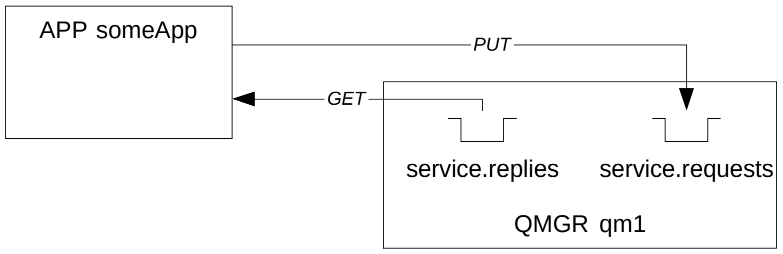


Chapter 3. Using the Network Design Notation

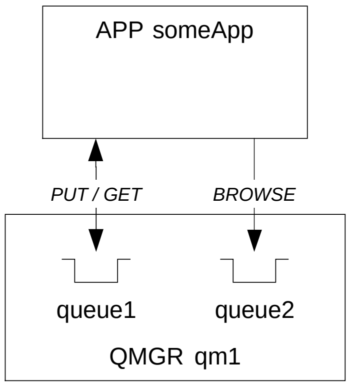
Basic Application-to-Queue Connectivity

Most of the notation examples in this document involve messaging applications. These are simply applications use asynchronous MQ messaging as part of their design.

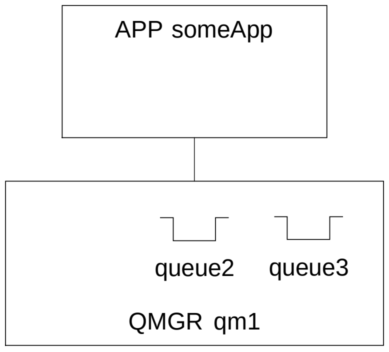
In the first example, a **messaging application** puts and gets messages using two **local queues**, hosted by a **queue manager**, qm1.



While the most common operations are puts and gets, we can easily describe other messaging operations such as “browse” on the connecting line.

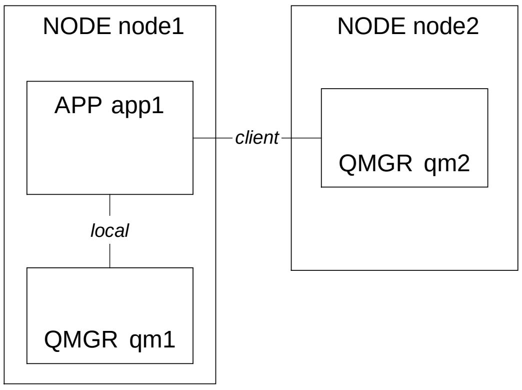


We also introduce this shorthand to indicate that application has general access to the queues **visible at that queue manager** (including shared queues and clustered queues). This can be useful to help avoid clutter on a diagram.

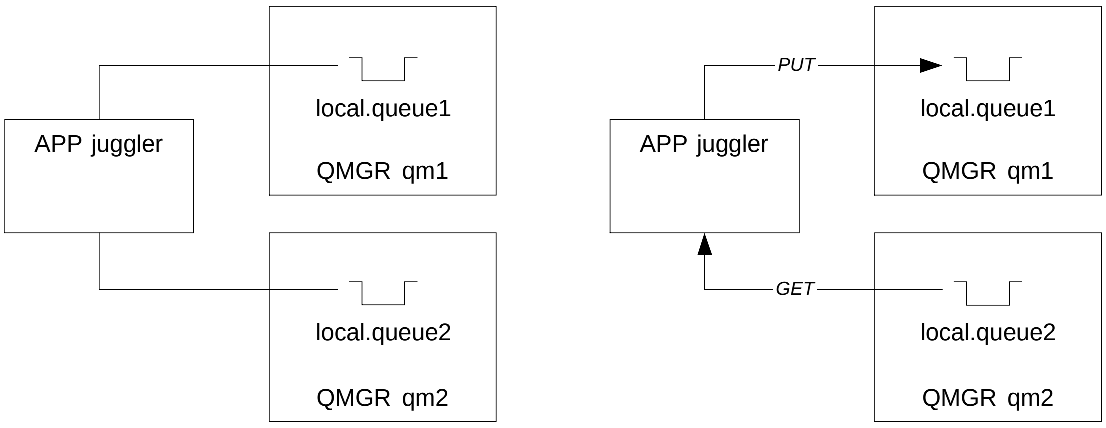


In effect, the line above represents a **connection** between application and queue manager, rather than the opening of any specific queue. When used like this, details of the actual messaging operations must be provided elsewhere.

Sometimes, it is necessary to show whether the application is connecting to a queue manager on a remote node (using the MQ client) as opposed to a local node. We consider a **node** simply as a logical operating system image, for example a single Unix machine, a virtual Linux image or a z/OS logical partition.



An application can also be shown having **connectivity** to several queue managers:



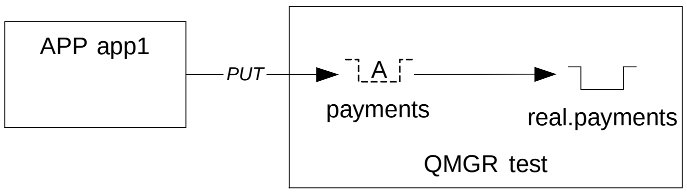
In fact, the left-hand diagram shows very little other than the fact that **juggler** needs to access queues in **qm1** and **qm2** *at some time*. There is insufficient detail to infer the pattern of access to the two queue managers and their queues.

The diagram on the right reveals a little more about the nature of the operations, but we still cannot specify whether juggler's put and get are part of one messaging operation or are two independent operations.

The principle adopted in this notation is that we show **associations** between applications and queues (a fairly static relationship), but do not show **ordering** or **sequencing** of message flow (which in any case is determined dynamically in many networks).

We believe that flow patterns are better represented in a **message flow model** using other techniques (such as UML Sequence Diagrams and Yourdon-Demarco Data Flow Diagrams).

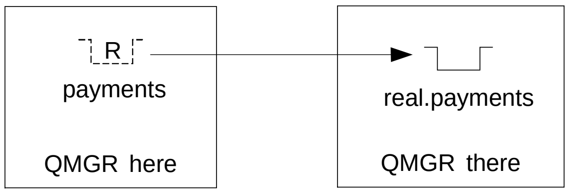
Next is an example of an **alias queue**, which is associated with ("is resolved to") a local queue.



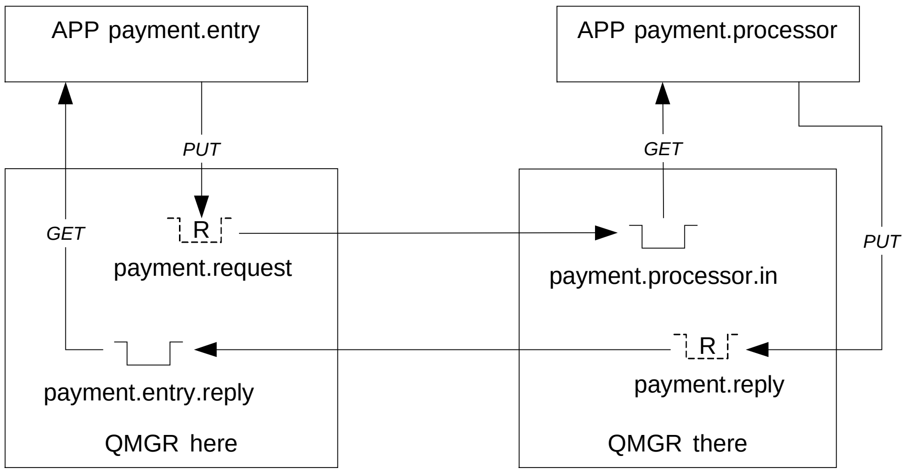
It is common practice to shield the names of local queues by designing applications to target alias queues.

Distributed Queuing

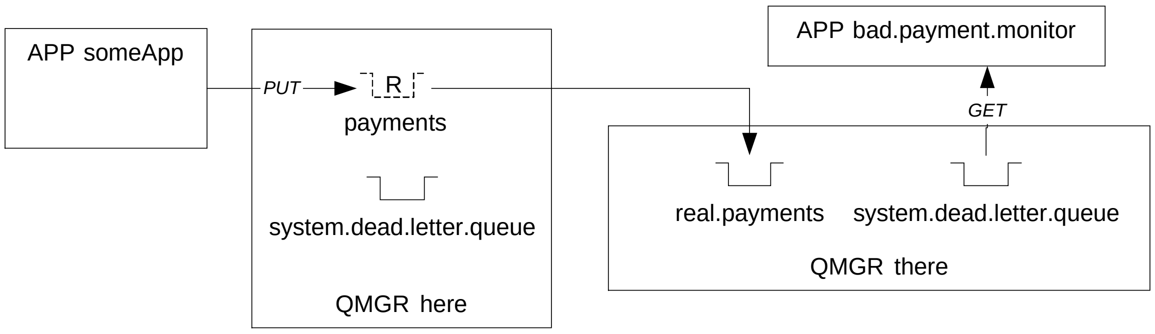
The **remote queue definition** is represented as follows (note the visual similarity with the alias queue from the previous section):



The next example shows a simple request-reply flow.



Of course, some messages reaching a queue manager might, for some reason, be un-deliverable to the target queue. It is conventional in MQ to configure a **dead letter queue** for the queue manager to store these undeliverable messages – we could show this using the normal symbol for a local queue:

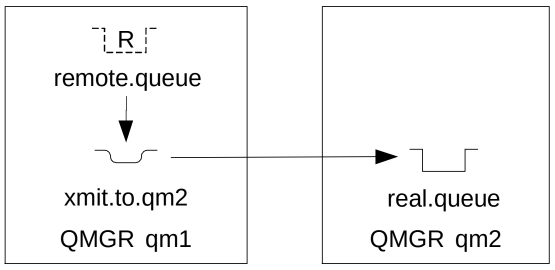


(We have not shown the actual flow of messages onto the dead letter queue – this is assumed to be per documented MQ behaviour.)

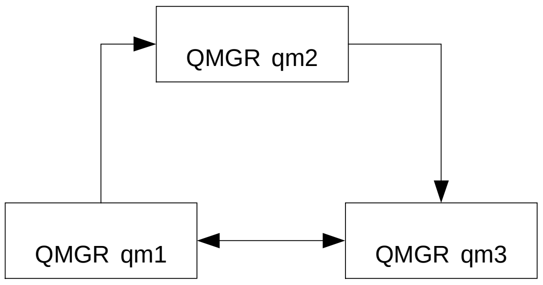
Channels

In distributed queuing, an underlying MQ channel and a transmission queue need to be present, so that messages can be transferred to a remote queue manager. Note that in the preceding networks, we have not shown any channels or transmission queues. We believe that it should be possible to omit channel information from many network diagrams without significantly reducing their usefulness.

However, where it is necessary to show transmission queues, we can do so as follows:



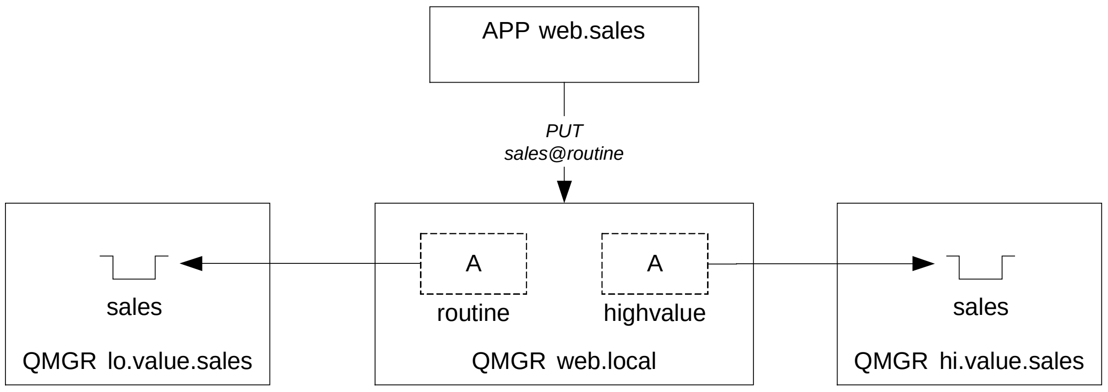
The next diagram shows channel connectivity between queue managers: although there are various types of MQ channels, we have opted for simplicity by using simple uni- or bi-directional arrows only.



Finally, in this section, we look at **queue manager aliases**.

In WebSphere MQ, an application can specify a queue manager at the time it **opens** a queue – this is called a *fully qualified open*. One of the functions performed by a queue manager alias is to map the queue manager name specified by the application to an alternate name. This re-mapping actually takes effect at the time an application puts a message.

In this network for example, we have two queue manager aliases defined in queue manager web.local:



The application performs a fully qualified open of the queue, represented here with **sales@routine**: **sales** is the target queue and **routine** is the specified queue manager alias.

When the message is put, the queue manager **web.local** will re-map **routine** to **lo.value.sales** and the message will be transferred to that queue manager.

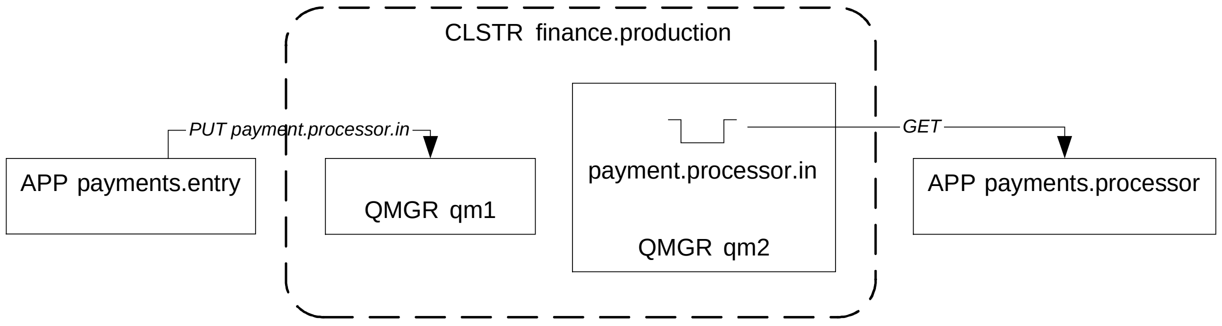
One practical use of queue manager aliases is the provision of classes of service in the messaging network. By configuring several queue manager aliases to queue managers over different underlying channels, applications can receive a particular class of messaging service, depending on which queue manager alias they choose.

MQ Clustering

Broadly speaking, **clustering** in WebSphere MQ provides:

- Improved availability of a receiving application as seen by the sender (by routing messages away from put-disabled queues or failed queue managers)
- Better message response time and throughput (by balancing load across a group of receivers)
- Simplified network administration (through auto-configuration of inter-queue manager channels)
- Extended queue visibility for put access (through clustering of queues)

The following network diagram illustrates a simple MQ cluster that we've named **finance.production**:



The notation shows:

i) Membership of a cluster for **cluster queue managers** (those inside the boundary)
ii) **Cluster queues** (those hosted in a member queue manager and inside the boundary)

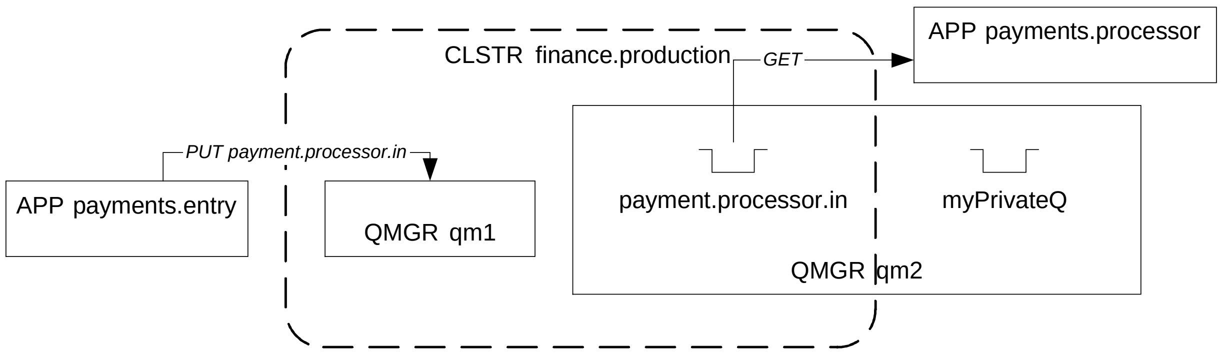
In this notation, we believe that it is normally not necessary to show the underlying cluster transmission queue or cluster sender / receiver channels. Nor do we show the message flow through the cluster itself – we assume this behavior in a cluster.

However, unlike previous diagrams, the target queue for a message put can be ambiguous when using the cluster notation.

Therefore, we must name the clustered queue on the put operation. In general, we advocate this as good practice for all diagrams where the target of a messaging operation is not obvious.

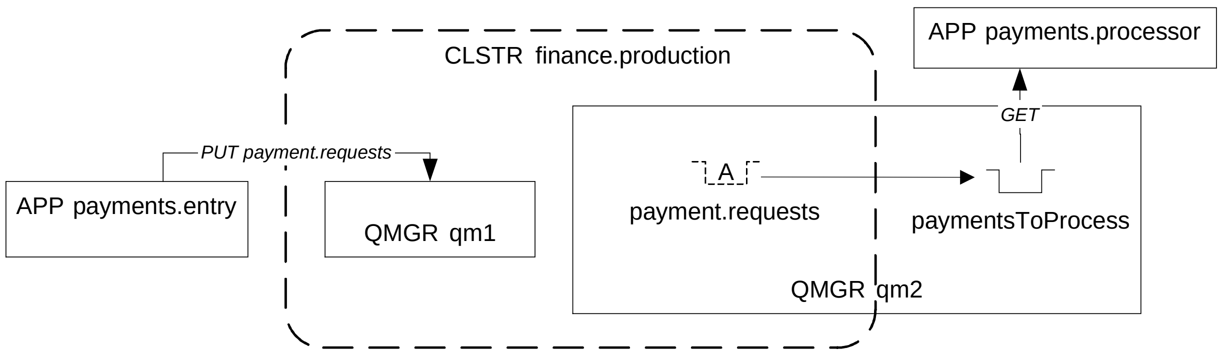
The cluster is scoped by the dotted boundary and the notation should be interpreted as: "an application connected to any member queue manager can put* to clustered queues owned by any other member queue managers in that cluster."

By allowing a queue manager to straddle the cluster boundary, we can illustrate local queues that are private and not visible to the cluster. In the following case, **myPrivateQ** is not accessible to the **payments.entry** application:

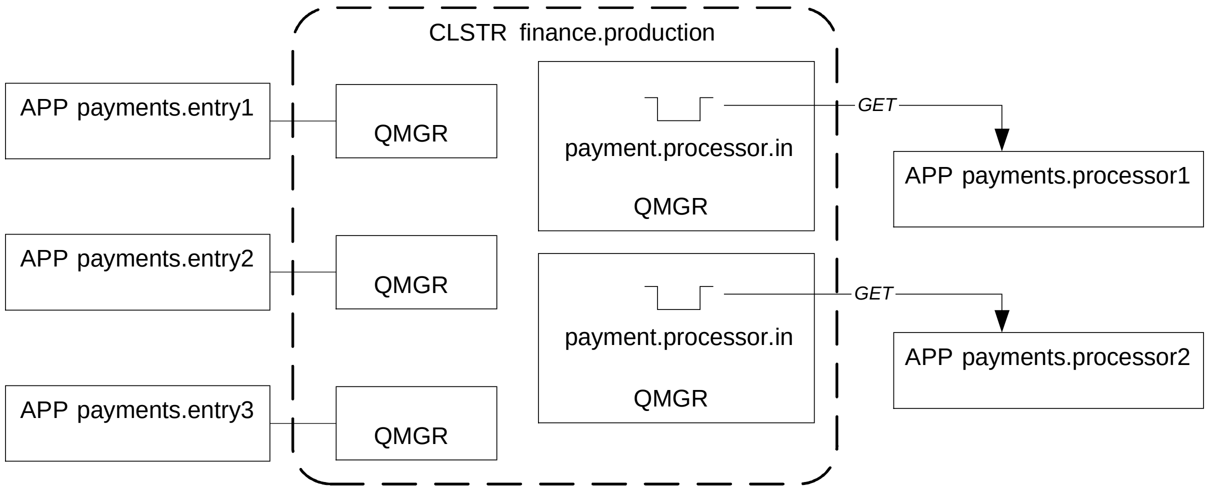


* a remote clustered queue is only accessible to an application for put purposes. MQ does not support message-get from a cluster queue at a remote queue manager.

We can also show a **clustered alias queue** along with its corresponding local queue that feeds the payments.processor application:



The next network shows three Payment Entry applications. Since they each connect via a queue manager that is a member of the cluster, they all have put access to a queue called **payment.processor.in**:



*Each p**ayments.entry** application puts to the clustered queue **payment.processor.in***

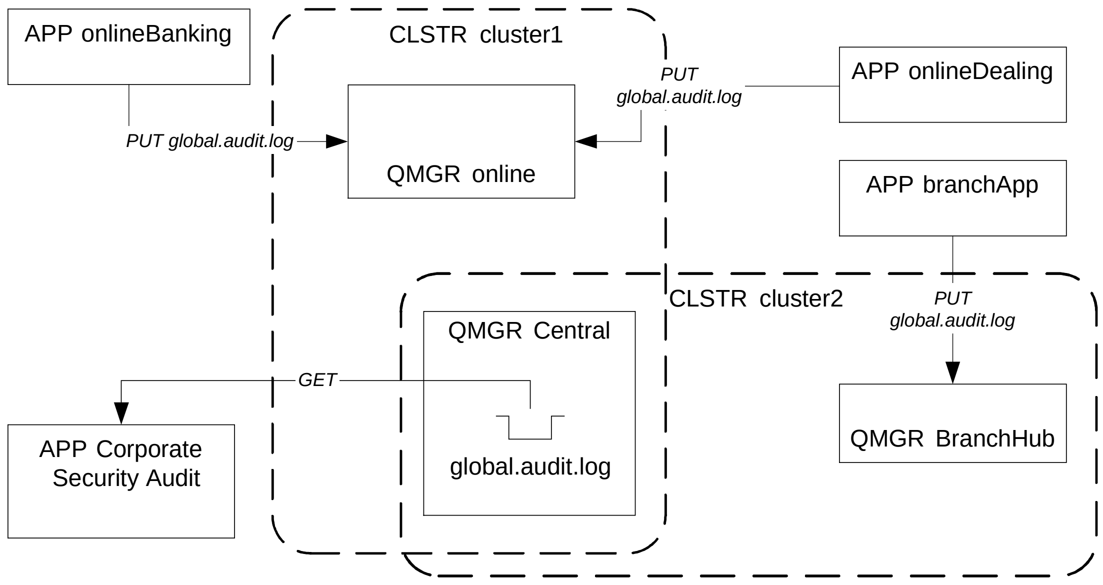
For better clarity, we have used a diagram note rather than show “PUT payment.processor.in” for each sender.

The line connecting each sender to its local queue manager indicates general access to queues visible in that queue manager, which include the cluster queues. Therefore, the note is essential to clarify the nature of the messaging operation being performed.

By arranging for a named queue to appear twice in the cluster as here, we can take advantage of the workload balancing and re-route-around-failure benefits of MQ clustering.

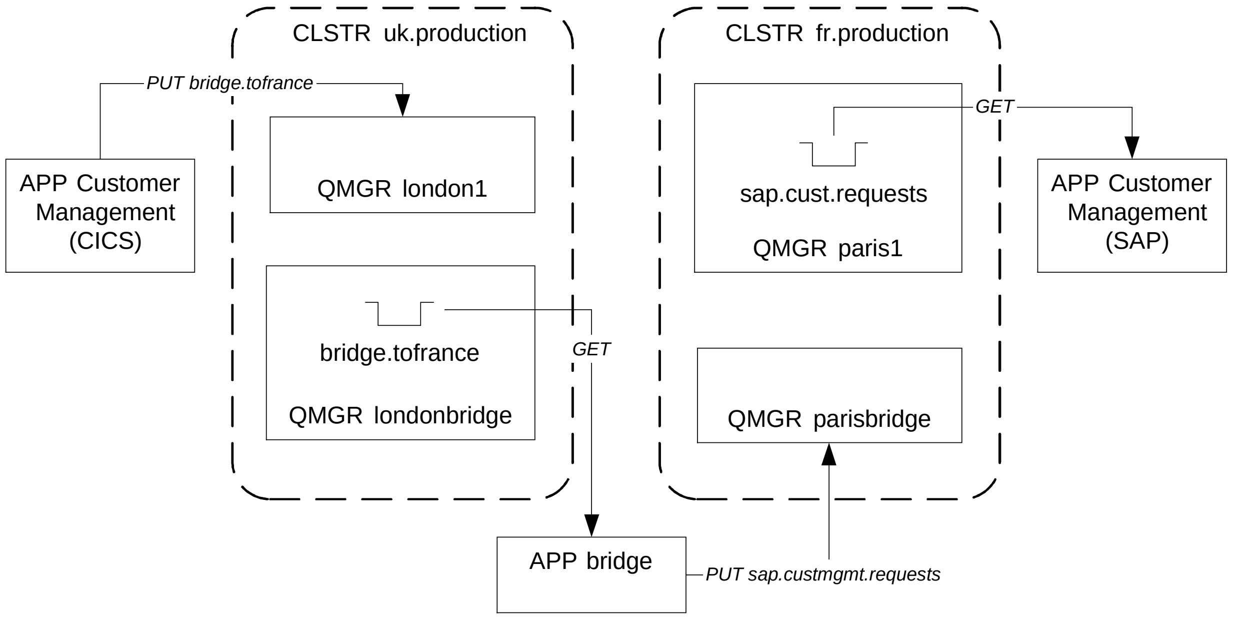
If the group of the Payments Processor applications is considered as a single “logical” application, then the main benefit brought by MQ clustering is the improvement of performance *and* availability of this application, as seen by the Payment Entry applications.

Next, we have a pair of overlapping clusters being used to create even wider queue visibility. Any application, once connected to a **cluster1** or **cluster2** queue manager, is able to put to the queue **global.audit.log**, hosted by queue manager **Central** and read by a central audit application:



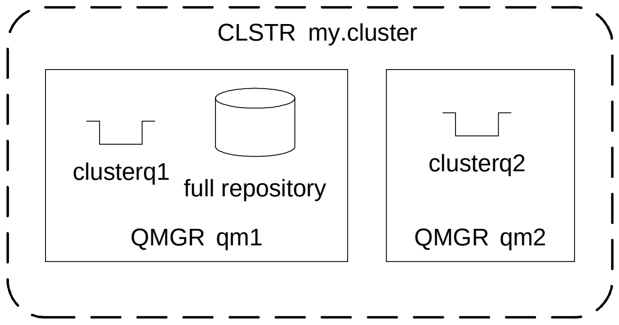
Applications can PUT audit messages to the clustered queue ***global.audit.log***

Below we have a system where messages flow from one cluster to another. In this case, the inter-cluster transfer has been implemented with an application called **bridge**. Applications on the **uk.production** cluster can put messages into the outbound queue from where they are retrieved by the bridge and sent to queues in **fr.production**, such as the SAP input queue:



The purpose of the diagram above is not to illustrate the mechanics of the Bridge application. This is a high-level diagram that illustrates the requirement to transfer messages between two clusters via an application. To describe the detailed workings of the bridge, a more specific diagram with supporting documentation on the message flow patterns would have to be supplied.

Finally in this section, when thinking about clustering implementation, it can be useful to show which of the queue managers are designated as holders of the **full repositories** of cluster information. We can do this easily with the generic data source symbol:

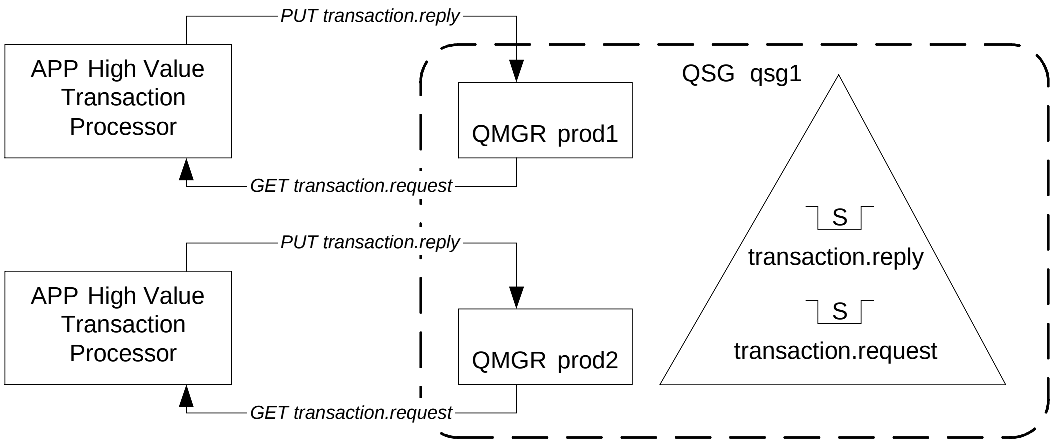


Shared Queues (z/OS)

On WebSphere MQ for z/OS, **shared queues** bring very high quality-of-service messaging infrastructure in terms of both performance (message throughput and response time) and availability (of messages in transit, and of the receiving application from the perspective of the sending application).

The benefit stems from the fact that the shared queue really is shared via the zSeries **coupling facility** - queue managers in a **queue-sharing group** (QSG) are able to cooperate to ensure that applications can put and send messages with maximum quality of service, while the dependency on any one queue manager is minimized.

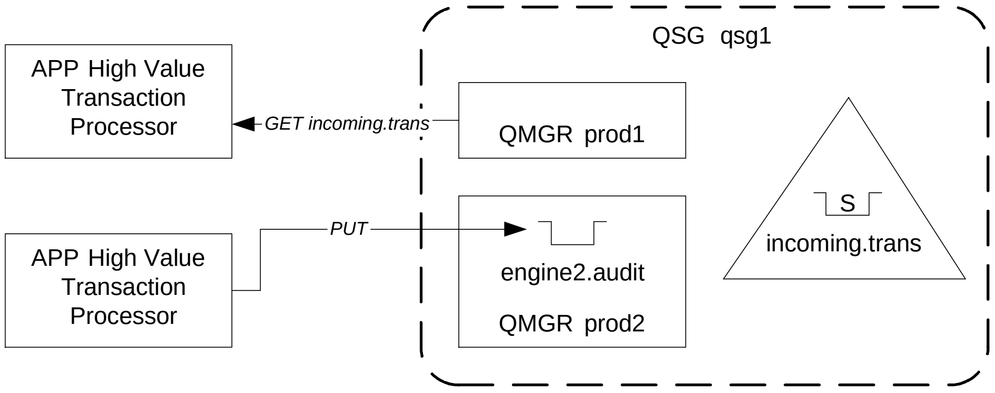
The diagram notation for shared queues is based on that in the MQ *Intercommunication* manual. For example, the two applications below can process messages for incoming transactions by retrieving them from a shared queue (which must reside in the coupling facility):



Use of a z/OS shared queue for input means that messages do not become isolated on the failure of any one application or queue manager. If a queue manager fails, another member of the queue-sharing group can resolve messaging transactions that were in-flight in the failed queue manager. Note in this case how the two queue managers do not have any queues of their own but offer a way to "plug-in" to the network.

The notation resembles that for clusters in that the visibility of **shared queues** (and membership of the queue sharing group) is denoted by visual containment. All queues inside the coupling facility are shared and accessible through all queue managers in the queue-sharing group.

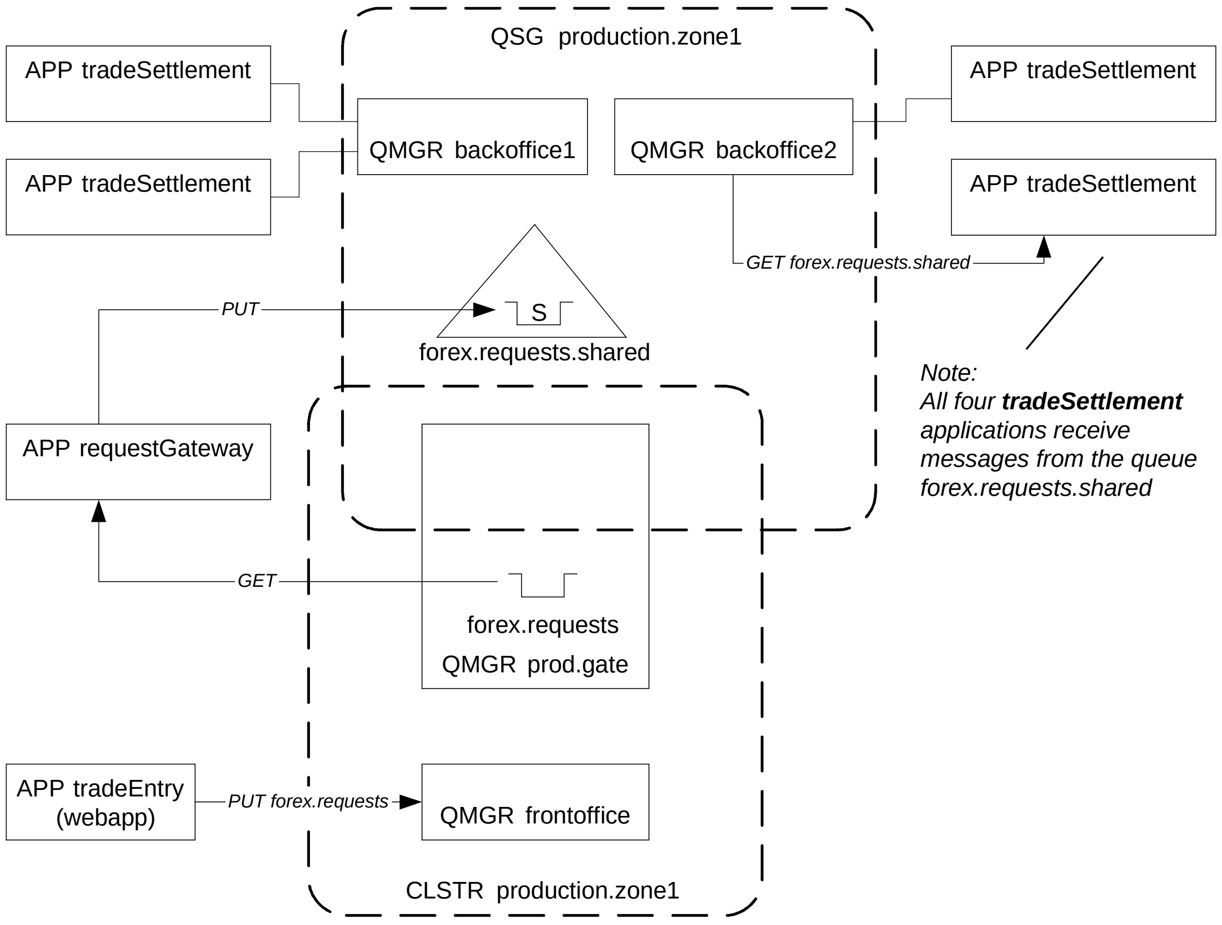
We show local* queues as usual:



Note that the visibility of a local queue is always restricted to applications connected to that queue manager, regardless of whether the queue lies inside or outside of the queue-sharing-group boundary. Unlike MQ clusters, it is not necessary to place these queues outside the boundary:

*Often referred to as private queues to distinguish from shared queues

Using the notation introduced so far, we can now illustrate an MQ cluster and shared queues working together in this fictional foreign exchange trading system:



The diagram above shows:

- A front-office trade entry application connected to a queue manager in an MQ cluster
- The queue manager, **prod.gate**, member of both the queue sharing group and the MQ cluster
- Message flow from the trade entry application to the shared queue **forex.requests.shared** via the **requestGateway** application (which might for example validate message formats).

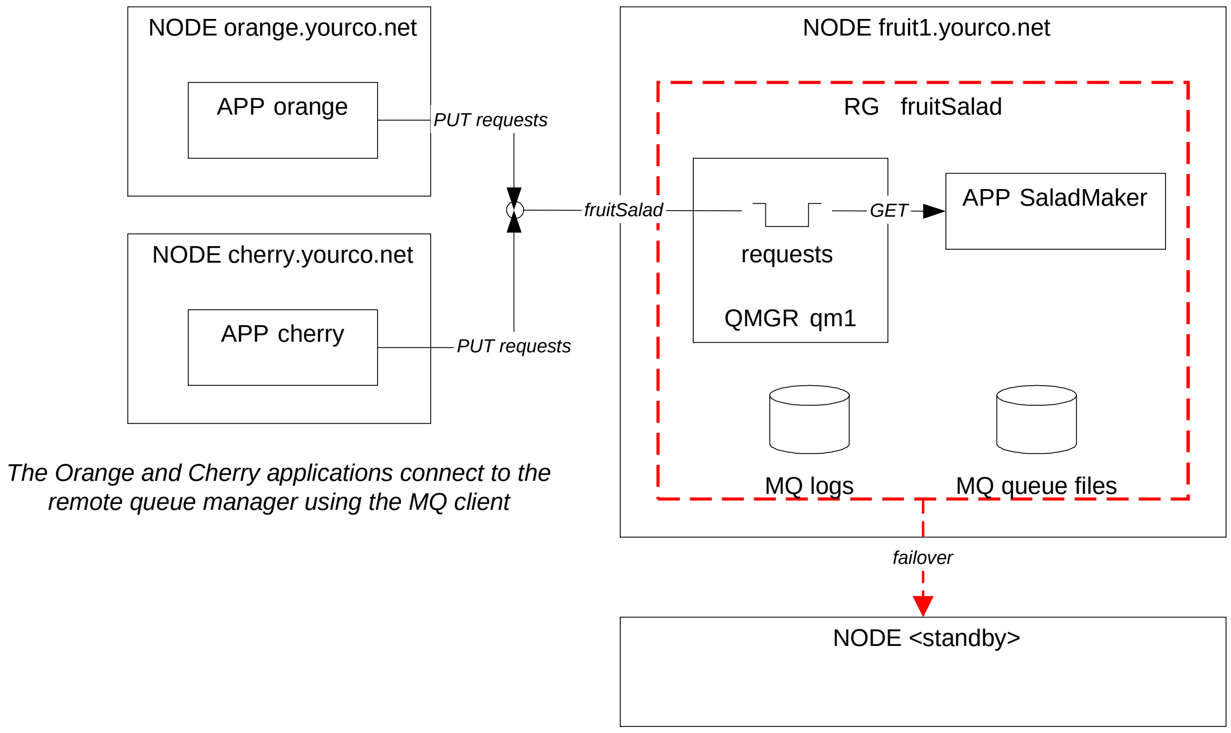
The single shared input queue allows the four trade settlement applications to compete to process the incoming trade message. The applications "pull" work as fast as they can and this promotes very efficient utilization of the backend system resources.

High Availability and Failover

The notation includes a simple means for showing high availability (HA) configurations of queue managers and applications. (We could equally include other components such as message brokers.)

Consider the first example below, which is a simple ‘standby configuration’. “SaladMaker” is an application that ‘makes’ fruit salad from orange and cherry messages. This application and its resources are considered as a **resource group** configured for high availability.

The high availability is achieved here using two nodes (corresponding to physical servers), one of which is acting as a ‘cold standby’.



The notation illustrates four general concepts, while avoiding HA vendor-specific terminology:

i) The **service entry point**: the interface symbol shows the point where clients connect to obtain normal service from the component. Advertised to clients as a **well-known address,** it could take any form, such as a TCP/IP port, an email address, or in our case, an MQ queue “requests”. The service entry point remains stable (to the client applications) during a fail-over scenario.

ii) The **resource group:** a set of sub-components that make up the highly available component (the “unit of failover”).

iii) The **node**: a physical machine or discrete operating system image where the resource group lives.

iv) The **failover arrow**: associates the resource group with its destination node in a failover scenario.

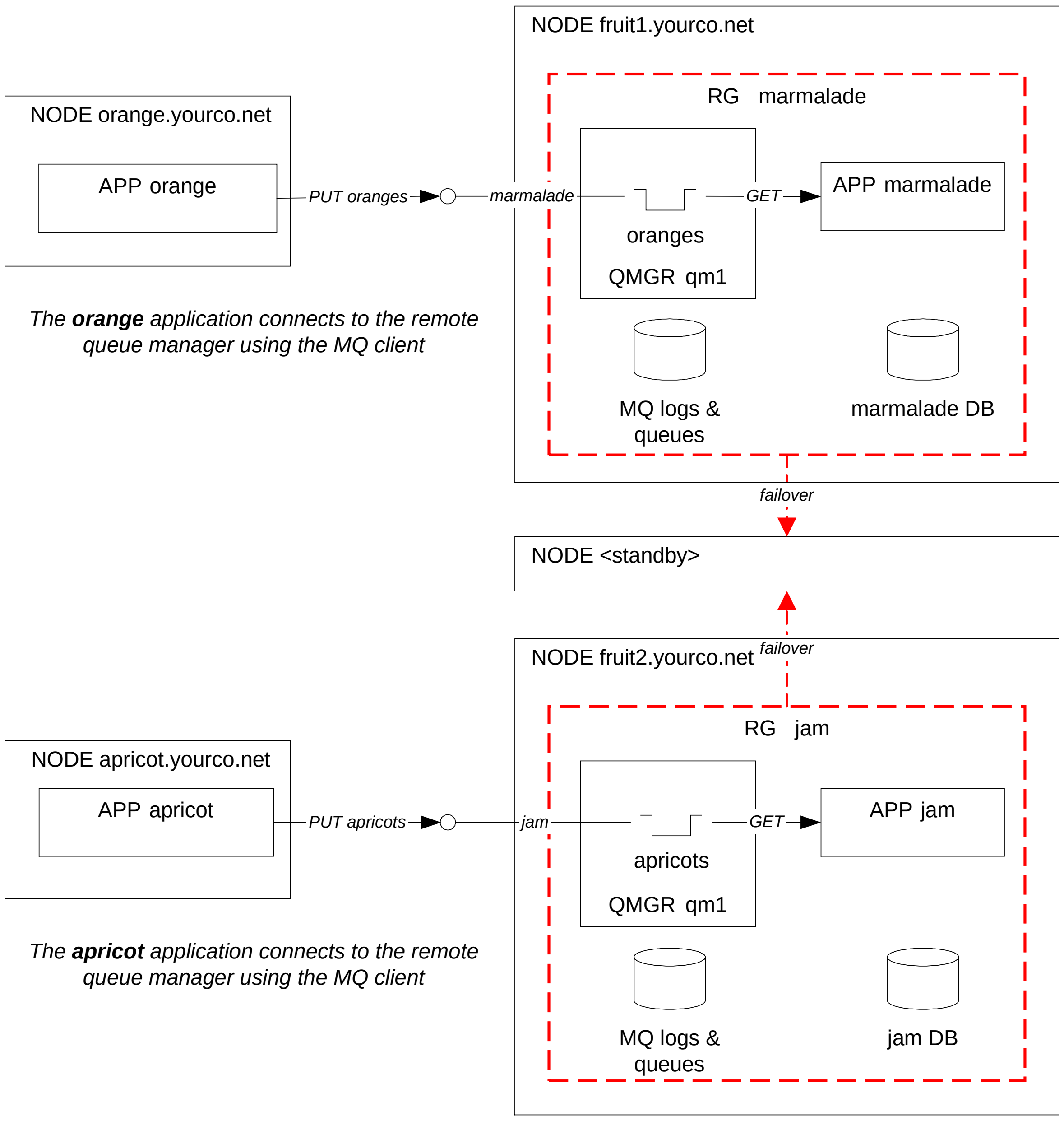
There are two ways of thinking about the resource group visualized above:

- “in a failure scenario, all of these components move to another node”, or
- “if any single sub-component fails, the whole group is considered failed (and all the components move to another node)”

Note the “RG” prefix to the resource group name: this helps to distinguish the resource group from clusters and queue sharing groups, which also use a dotted container symbol. We have also used red for the boundary but this is not required (and doesn’t work in monochrome media).

Designing real MQ high-availability configurations requires consideration of the physical storage that MQ requires for its message queues and transaction logs. We have used the data-source symbol here for this purpose.

The next diagram features an ‘N+1 standby configuration’. Here, a spare node with queue manager acts as standby for resource groups running on several other (N) nodes. Note that the number of service entry points reflects the number of active or “hot” components.

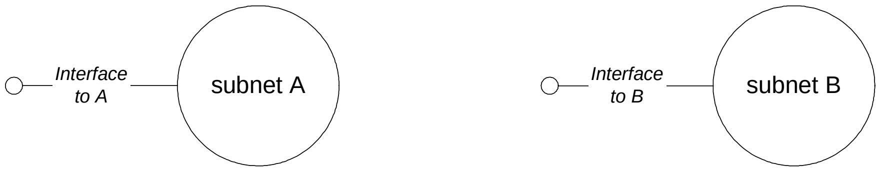


*Background notes on preserve making:

- Node fruit1 makes marmalade from oranges
- Node fruit2 makes jam from apricots
- In this diagram, the name ‘marmalade’ is used for the resource group, the entry interface, a database
- …and similarly for the name ‘jam’

Messaging Subnets

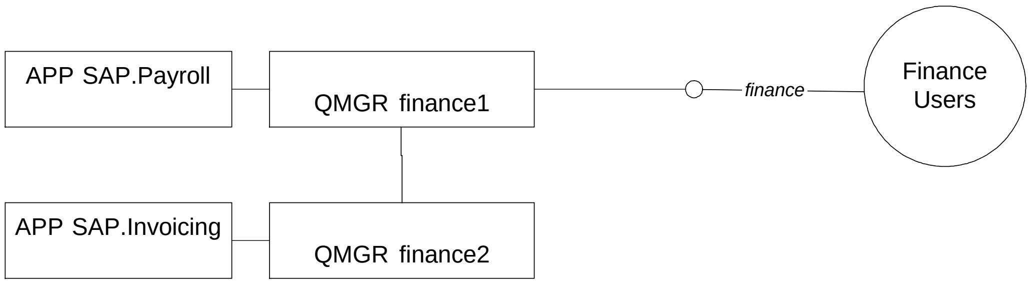
To support diagrams of large messaging networks, the notation includes shorthand for referring to a complete **sub-network** as a unit.



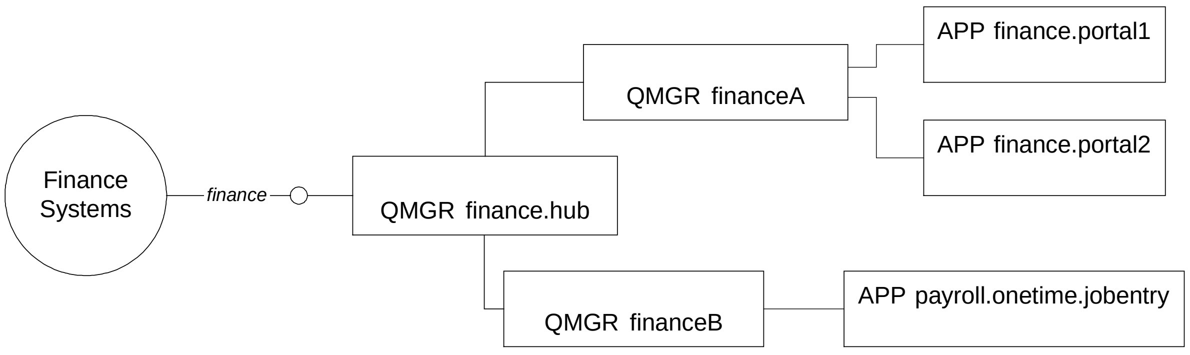
The interface symbol, borrowed from UML and object-oriented modeling, is used here to represent the **messaging interface** to a network. In practice, a specification of this interface might include:

- The types of message formats flowing through that interface
- The destination (and perhaps reply-to) queue names
- The expected patterns of message flow

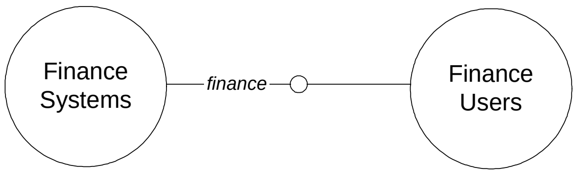
As a more concrete example, the following diagram details the backend portion of a messaging network (on the left) that interfaces with Finance Users sub-network:



The corresponding detail for the “Finance Users” subnet might look like this:



The implication is that the “finance” interface specifies an integration point (a degree of “pluggability”) between these sub-networks. The high-level network could be rendered (somewhat trivially) as:



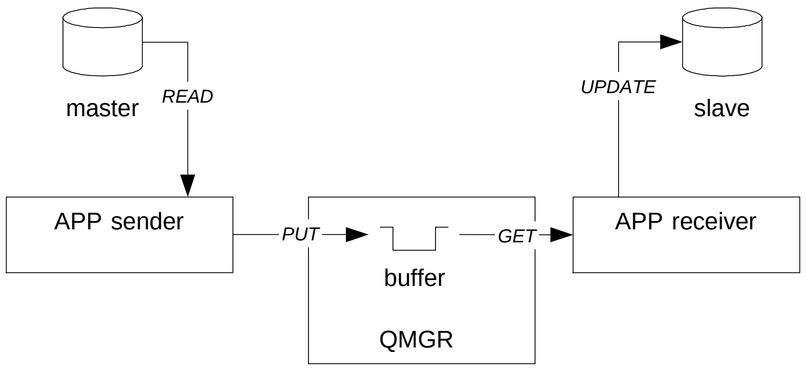
We anticipate that for some messaging networks (especially those that seek to implement re-usable subnets), it is desirable to be able to draw these high-level diagrams that expose the major interfaces (perhaps with accompanying documented specifications), before dropping into more detailed views of the sub-networks themselves.

Other Symbols

Data Sources

A major factor in the design of many message queuing networks is the positioning of data sources. We have included a datasource symbol so that application access to these resources can be visualized. The symbol is intended to represent any form of file- or disk-based storage and could be used to represent databases, log files, lookup tables, registries, etc.

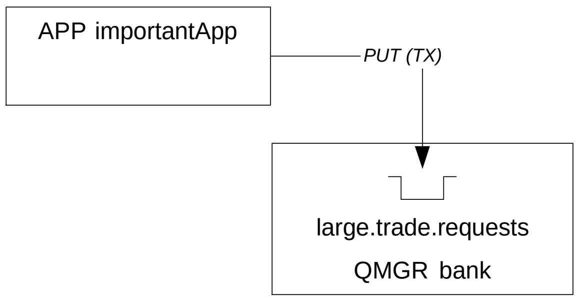
This diagram illustrates a simple network to support a message-based data replication scheme.



Transactional Messaging Operations

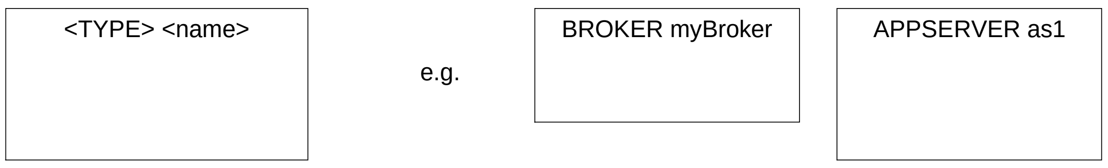
Some applications (especially those hosted by application servers) are able to send or receive messages within the scope of a transactional Unit of Work.

A transactional put or get is distinguished simply by a (TX) suffix on the PUT/GET operation.



Generic Software Components

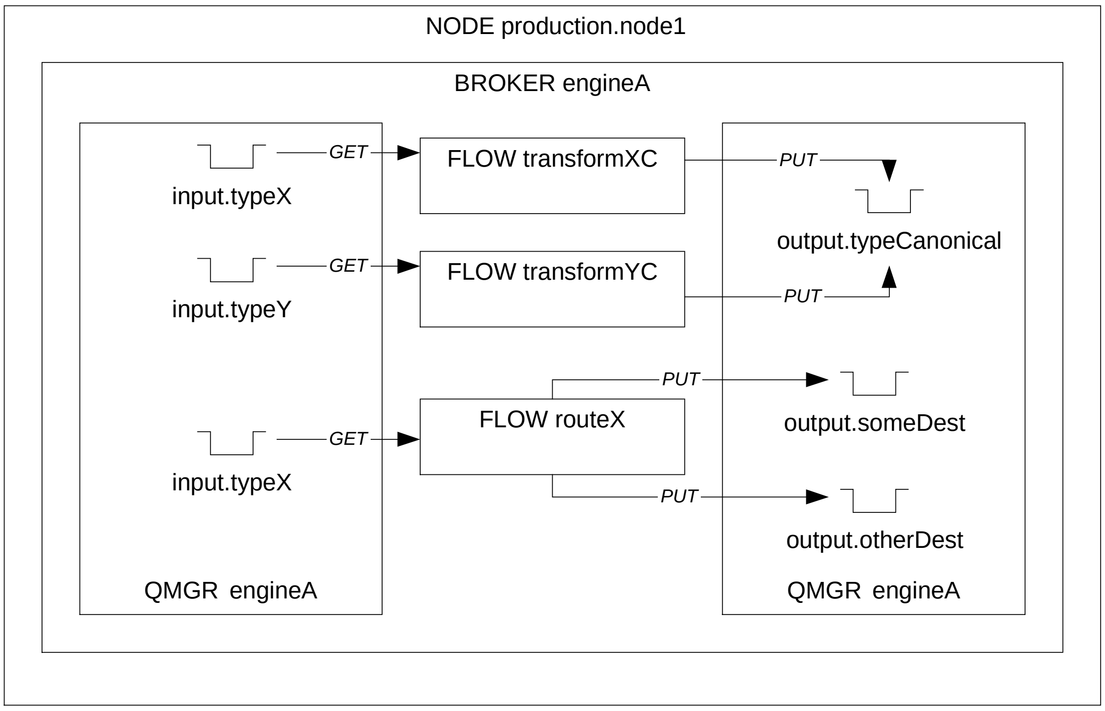
There are many software parts that can appear in the context of a messaging infrastructure. For these, we suggest use of the generic software component symbol, a simple rectangle using the naming form adopted elsewhere in this notation. For example, see the generic software component symbol instantiated as a message broker and an application server below:



WebSphere MQ Integrator Brokers and Flows

Traditionally, WebSphere MQ Integrator message brokers receive and send their work through MQ queues. There are frequent cases where it is desirable to depict brokers, message flows and queues on a diagram to clarify the relationships between these objects.

Here we use of the notation to show a simple message broker configuration, using the generic software component symbol to represent BROKERs and message FLOWs:



Notes for illustrating WebSphere MQ Integrator brokers:

- In MQ Integrator, a queue manager instance is dedicated to a particular broker instance, so we have chosen to show the queue manager contained in the broker. Further, in accordance with general practice, we have given the broker and queue manager the same name

- Though not essential, we represent the same queue manager instance twice, so that message operations can be drawn more easily

- Message flows appear in much the same way as regular messaging applications – they connect to queues and perform GET and PUT operations (at the Input and Output flow nodes respectively)

Chapter 4. Other Information

Usage Guidelines

We suggest that the following be considered for network design diagrams drawn using this notation:

i) **Consider static aspects of the network only**: Dynamic aspects such as message flow sequencing, routing and queue build-up are best modelled using other techniques (including Data Flow Diagrams, UML Interaction or Collaboration diagrams, etc.)

ii) **Represent only what is sensible**: A lot of network design information is not suitable for visual representation and should be captured in notes, documents, flow diagrams, models, etc.

iii) **Capture the essence of the design**: Normally, it will be impractical and unnecessary to model every component of a network in a single diagram. In many cases, drawing a subset of cloned applications or showing a sub-chain of a long multi-hop sequence should be sufficient to get the concept across and is much easier for a reader to comprehend.

iv) **Consider drawing different views**: Do not try to serve too many needs with one diagram. Omit detail when it is not directly relevant to the viewer's purposes. For example, it is not necessary to show every single message put or get. Considering whether all objects need names or whether it is sufficient just to indicate their type (this also avoids the potential problem of a reader inferring something you didn't intend from an arbitrarily chosen name.)

Tips for Clarity and Emphasis

There are many ways to add emphasis to diagrams to draw out interesting detail:

- Line weight
- Line type (solid, dashed, dotted)
- Color
- Font type, size and options (bold, italics, etc.)
- Arrowhead size and style
- Containment and overlap
- Shadows and other effects

However, with these techniques, it is all too easy to inadvertently **reduce** the clarity of a diagram*, especially when using drawing software.

This is most likely when:

- Too many techniques are being used at the same time. If you find that you don't have enough forms of emphasis available, this is probably a sign that the diagram needs to be broken up into multiple, simpler views.
- Techniques are being used inconsistently throughout one or more diagrams

The hazard is that a diagram viewer will construe the importance of various elements differently than you intended, or worse, will be plain confused!

In this document, we have tried to make minimum possible use of these emphasis techniques, in order to leave as many as possible in the hands of users of this notation.

We suggest that when using any of the techniques above, take a step back from your diagram and try to imagine what emphasis a viewer will see.

Happy drawing!

*As discovered during the development of this notation!

Bibliography

- *Design Rules, Naming Convention and A Minimum Symbol Set for MQSeries System Design,* Benjamin Zhou (Financial Services Corporation). WebSphere MQ SupportPac MD04

- *MQSeries Queue Manager Clusters*, IBM Corporation SC34-5349-02

- *MQSeries Intercommunication*, IBM Corporation SC33-1872-05

- *MQSeries for AIX - Implementing with HACMP Version 2.0* IBM Corporation. WebSphere MQ SupportPac MC63

- *MQSeries Standards and Conventions*, IBM Corporation. WebSphere MQ SupportPac MD01

-- End of Document --