

7. MQSeries

Version 1.5 v. 15.11.01

7.1 Einführung

MQSeries ist ein sehr erfolgreiches Middleware-Produkt der Firma IBM für kommerzielles Messaging und Queuing. Es wird weltweit von vielen Nutzern vorwiegend aus der Industrie in Hochgeschwindigkeits-Implementierungen von verteilten Anwendungen benutzt. MQSeries-Applikationen können mit minimalem Aufwand entwickelt und getestet werden.

Da MQSeries auf einer Vielzahl von Plattformen lauffähig ist, können Programme infolgedessen miteinander über ein Netzwerk von unterschiedlichen Komponenten, wie z.B. Prozessoren, Subsystemen, Betriebssystemen und Kommunikations-Protokollen kommunizieren. MQSeries-Programme verwenden ein konsistentes Application Program Interface (API) auf allen Plattformen. In der Abbildung 1 sind die Haupt-Komponenten einer MQSeries-Anwendung dargestellt.

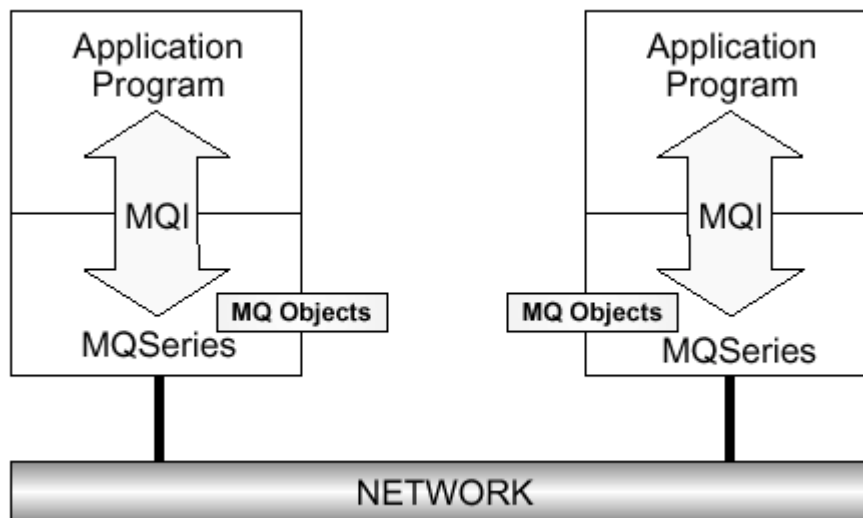


Figure 1. MQSeries at Run Time

Die Programme verwenden MQSeries-API-Calls, die das Message Queue Interface (MQI) implementieren, um mit einem Queue-Manager (MQM) zu kommunizieren. Letzterer realisiert das Run-Time-Programm von MQSeries. Die Arbeit des Queue-Managers bezieht sich auf Objekte wie z. B. Queues und Channels. Der Queue-Manager selbst ist auch ein Objekt.

Message Queuing ist eine Methode der Programm-zu-Programm-Kommunikation. Programme sind in der Lage, Informationen zu senden und zu empfangen, ohne dass eine direkte Verbindung zwischen ihnen besteht. Die Programme kommunizieren miteinander durch Ablegen von Nachrichten in Message-Queues und durch Herunternehmen der Nachrichten von den Message-Queues.

Die wichtigsten Charakteristika von Message-Queuing sind:

- Zeit-unabhängige (asynchrone) Kommunikation
Der Austausch von Nachrichten zwischen dem sendenden und dem empfangenden Programm ist zeitunabhängig. Das sendende Programm kann die Verarbeitung fortsetzen, ohne auf die Rückmeldung des Empfängers der Nachricht zu warten. MQSeries hält die Nachrichten solange in der Queue, bis sie verarbeitet werden.
- Verbindungslose Kommunikation
Sendende und empfangende Programme benutzen nur Queues für die Kommunikation. MQSeries ist für alle Aktivitäten, die mit einer solchen Kommunikation verbunden sind, verantwortlich: Erhalten der Queues und der Beziehungen zwischen Programmen und Queues, Handling der Netzwerk-Restarts, Bewegung der Nachrichten durch das Netzwerk.

- Parallelverarbeitung

MQSeries erlaubt eine 1:1-Beziehung zwischen den kommunizierenden Programmen. Es kann aber auch Anwendungsstrukturen und Nachrichten-Übertragungsformen unterstützen, die viel komplexer sind: many-to-one, one-to-many oder irgendeine Kombination dieser Beziehungen.

MQSeries ist eine Familie von Produkten für Cross-Network-Kommunikation. Sie ist auf folgenden Plattformen verfügbar:

- Host

- IBM MVS/ESA-Server and Client enabled

- Tandem NonStop Kernel-Server

- IBM VSE/ESA-Server

- Midrange

- Pyramid DC/Osx-Server and Client

- IBM OS/400-Server and Client enabled

- Digital Open VMS VAX-Server

- Workstation

- IBM AIX-Server and Client

- NCR (AT&T GIS) UNIX-Server and Client

- Siemens Nixdorf SINIX-Server and Client

- Hewlett Packard HP-UX-Server and Client

- Sun Solaris and SunOS-Server and Client

- Digital Unix-Client - available as SupportPac

- Linux Client - available as SupportPac

- Desktop

- IBM OS/2-Server and Client

- Microsoft Windows N- Server and Client

- Microsoft Windows (MQSeries V2.0 - 16-Bit)

- Microsoft Windows (MQSeries V2.1 - 32-Bit)

- Windows 95 (32-Bit)-Client

- SCO Unixware-Server

- SCO Unix-Server

- SCO Unix (L2)-Client

- DOS Client - available as SupportPac

- MacOS (OEM)-Client

Mittels MQSeries-Produkten sind Programme in der Lage, sich mit anderen über ein Netzwerk mit unterschiedlichen Komponenten zu unterhalten: Prozessoren, Betriebssysteme, Subsysteme, Kommunikations-Protokolle. Die Kommunikation verwendet ein einfaches und konsistent gemeinsames API, das Message Queue Interface (MQI).

Nachrichten (Messages) und Warteschlangen (Queues) bilden die Grundlage von MQSeries. Die Programme kommunizieren miteinander, indem sie sich Nachrichten senden, die spezifische Daten enthalten..

Nachrichten werden in Queues im Speicher abgelegt, so dass Programme unabhängig voneinander laufen können, mit unterschiedlichen Geschwindigkeiten, an unterschiedlichen Orten, ohne dass eine logische Verbindung zwischen ihnen besteht.

7.2 Messaging und Queuing

Messaging bedeutet, dass Programme durch Senden von Daten in Nachrichten (Messages) kommunizieren und nicht durch wechselseitiges, direktes Aufrufen.

Queuing heißt, dass Programme über Queues miteinander kommunizieren. Dadurch ist es nicht notwendig, dass diese Programme zeitlich parallel ausgeführt werden.

Eine Queue bezeichnet eine Datenstruktur, die Nachrichten speichert. Auf einer Queue können Anwendungen oder ein Queue-Manager Nachrichten ablegen. Queues existieren unabhängig von den Anwendungen, die Queues benutzen. Als Speichermedien für eine Queue kommen Hauptspeicher (wenn sie temporär ist), Platte bzw. ähnliche Zusatzspeicher oder beides in Frage. Jede Queue gehört zu einem Queue-Manager. Letzterer ist verantwortlich für die Verwaltung der Queues, er legt die empfangenen Messages auf der entsprechenden Queue ab. Queues heißen lokal, d.h. sie existieren in ihrem lokalen System, oder remote. Eine remote Queue ist einem anderen Queue-Manager, der nicht zu dem lokalen System gehört, zugeordnet

Von den Anwendungen werden Nachrichten gesendet und empfangen. Dabei benutzen die Anwendungen sogenannte MQI-Calls. Z.B. kann eine Applikation eine Nachricht auf einer Queue ablegen und eine andere Applikation diese Nachricht von derselbigen Queue zurückerhalten.

Beim asynchronen Messaging führt das sendende Programm seine eigene Verarbeitung weiter aus, ohne auf eine Antwort seiner Message zu warten. Im Gegensatz dazu wartet beim synchronen Messaging der sendende Prozeß auf eine Antwort, bevor er seine Verarbeitung fortsetzt. Für den Nutzer ist das zugrunde liegende Protokoll transparent.

MQSeries wird in Client/Server- oder in verteilten Umgebungen eingesetzt. Die zu einer Anwendung gehörenden Programme können in unterschiedlichen Rechnerarchitekturen auf verschiedenen Plattformen ausgeführt werden. Die Anwendungen sind von einem System oder Plattform zu einem anderen übertragbar. Programme werden in verschiedenen Programmiersprachen einschließlich Java geschrieben. Für alle Plattformen ist derselbe Queuing-Mechanismus gültig.

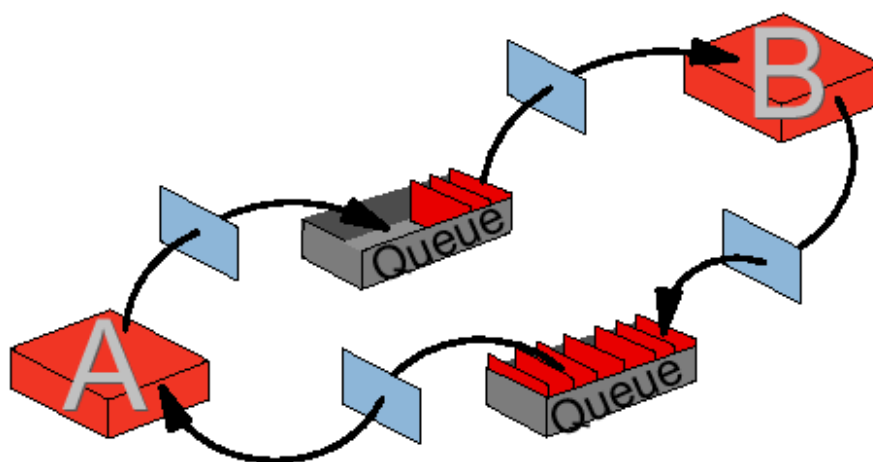


Figure 2. Messages and Queues

MQSeries kann aufgrund der Kommunikations-Art (mittels Queues) als indirekte Programm-zu-Programm-Kommunikation betrachtet werden. Der Programmierer ist nicht in der Lage, den Namen der Ziel-Anwendung, zu der eine Message gesendet wird, zu spezifizieren. Er kann dagegen einen Ziel-Queue-Namen angeben; jede Queue ist mit einem Programm verknüpft. Es können ein oder mehrere Eingangs-Queues und verschiedene Ausgangs-Queues für eine Applikation existieren. Die Ausgangs-Queues enthalten Informationen, die auf anderen Servern verarbeitet werden sollen, oder Antworten für Clients, die die Transaktion initiiert haben.

Bei einer Anwendung ist es nicht erforderlich, dass sich der Programmierer um das Ziel-Programm kümmert. Es ist belanglos, ob der Server momentan nicht im Betrieb ist oder keine Verbindung zu ihm besteht. Der Nutzer schickt eine Message an die Queue, die mit der Applikation verbunden ist. Letztere kann zur Zeit des Requests verfügbar sein oder nicht. MQSeries beobachtet den Transport zur Ziel-Applikation und startet diese, wenn es notwendig ist.

Wenn das Ziel-Programm nicht verfügbar ist, steht die Message in der Queue und wird später verarbeitet. Abhängig davon, ob die Verbindung zwischen zwei Systemen hergestellt ist oder nicht, befindet sich die Queue entweder in der Ziel-Maschine oder in dem sendenden Rechner. Eine Anwendung kann prinzipiell über mehrere Tage laufen oder sie wird getriggert, d.h. sie wird automatisch gestartet, wenn eine oder eine spezifische Anzahl von Nachrichten ankommt (s. Kapitel 7.8).

In der Abbildung 2 ist die Kommunikation von zwei Programmen A und B dargestellt. Es existieren zwei Queues, wobei die eine die Ausgangs-Queue von A und gleichzeitig die Eingangs-Queue von B bildet, während die zweite Queue für die Antwort von B nach A benutzt wird. Die Rechtecke zwischen den Queues und den Programmen repräsentieren das Message-Queuing-Interface (API), das von dem Programm verwendet wird, um mit dem Run-Time-Programm von MQSeries (Queue Manager) zu kommunizieren. Das API stellt ein einfaches Multi-Plattform-API mit 13 Call's dar.

7.2.1 Messages

Eine Message besteht aus zwei Teilen:

- Daten, die vom einem Programm zu einem anderen gesendet werden
- Message-Deskriptor oder Message-Header

Der Message-Deskriptor identifiziert die Message (Message-ID) und enthält Steuerinformationen (Attribute), wie z.B. Message-Type, Zeit-Ablauf, Korrelations-ID, Priorität und Namen der Antwort-Queue.

Eine Message kann bis zu 4 MByte oder 100 MByte lang sein. Die Länge ist von der benutzten MQSeries-Version abhängig. MQSeries-Version 5 (für verteilte Plattformen) unterstützt eine maximale Message-Länge von 100 MByte.

7.2.2 Message-Segmentierung und -Gruppierung

In der MQSeries-Version 5 können die Messages segmentiert oder gruppiert werden. Die Message-Segmentierung erfolgt transparent für den Anwendungs-Programmierer. Der Queue-Manager segmentiert unter bestimmten Bedingungen eine umfangreiche Nachricht, wenn er diese in einer Queue findet. Auf der Empfängerseite hat die Anwendung die Option, entweder die gesamte Message in einzelnen Teilen oder jedes Segment separat zu empfangen. Die Auswahl hängt davon ab, wie groß der verfügbare Puffer für die Applikation ist. Eine zweite Segmentierungs-Methode überlässt es dem Programmierer, eine Message entsprechend der verfügbaren Puffer-Größe zu teilen. Der Programmierer betrachtet jedes Segment als eine separate physikalische Message. Folglich bilden verschiedene physikalische Messages eine logische Message. Der Queue-Manager stellt sicher, dass die Reihenfolge der Segmente erhalten bleibt.

Um den Verkehr über das Netzwerk zu reduzieren, können auch verschiedene kleine Nachrichten zu einer großen physikalischen Message zusammengefasst werden. Diese wird dann an das Ziel gesendet und dort disassembliert. Das Message-Grouping garantiert ebenfalls die Reihenfolge der gesendeten Nachrichten.

7.2.3 Distribution List

In der Version 5 von MQSeries ist es möglich, eine Message an mehr als eine Ziel-Queue zu senden. Dafür steht der MQPUT-Call zur Verfügung und erfolgt mit Hilfe der dynamischen Distribution List. Eine Distribution List stellt z.B. ein File dar, das zum Zeitpunkt des Applikations-Starts gelesen wird. Das File kann auch modifiziert

werden. Es enthält eine Liste der Queue-Namen und der zugehörigen Queue-Manager. Wird eine Message an mehrere Queues gesendet, die zu demselben Queue-Manager gehören, dann erfolgt das Senden nur einmal und reduziert somit den Netzwerk-Verkehr. Der empfangende Queue-Manager kopiert die Nachrichten und stellt sie in die Ziel-Queues. Diese Funktion heißt "Late Fan-Out".

7.2.4 Message-Typen

MQSeries unterscheidet vier verschiedene Message-Typen:

Datagram: Eine Message enthält Informationen, auf die keine Antwort erwartet wird.
 Request: Eine Message, für die eine Antwort angefordert wird.
 Reply: Eine Antwort auf eine Request-Message
 Report: Eine Message, die einen Event beschreibt (z.B. Fehlermeldung oder Bestätigung beim Eintreffen der Nachricht)

7.2.5 Persistente und nicht-persistente Messages

Das Applikations-Design bestimmt, ob eine Message ihr Ziel unbedingt erreichen muss oder diese gelöscht werden, wenn das Ziel nicht innerhalb einer vorgegebenen Zeit erreicht wird. MQSeries unterscheidet zwischen persistenten und nicht-persistenten Messages. Die Übertragung von persistenten Nachrichten wird sichergestellt, d.h. sie werden protokolliert, um bei Systemausfällen erhalten zu bleiben. In einem AS/400-System heißen diese Protokolle "Journal Receivers".

Nicht-persistente Messages können nach einem System-Restart nicht wieder hergestellt werden.

7.2.6 Message Descriptor

In der Abbildung 3 sind einige interessante Attribute des Message-Descriptors dargestellt. Die Attribute erklären einige Funktionen des verwendeten Queue-Managers.

Version	Return address
Message ID / Correlation ID	Format
Persistent / non-persistent	Sender application and type
Priority	Report options / Feedback (COA, COD)
Date and time	Backout counter
Lifetime of a message	Segmenting / grouping information

Figure 3. Some Attributes of the Message Descriptor

- Die Version des Message-Descriptors hängt von der MQSeries-Version und der verwendeten Plattform ab. Für die Funktionen, die mit Version 5 eingeführt wurden, sind Informationen über Segmente und ihre Reihenfolge notwendig.
- Message- und/oder Korrelation-ID werden zur Identifikation einer spezifischen Anforderung oder einer Antwort-Message verwendet. Bevor der Programmierer die Request-Message auf die Queue legt, kann er die IDs speichern und benutzt diese in einer anschließenden Get-Operation für die Antwort-Message. Das Programm, das die Request-Message empfängt, kopiert diese Information in die Antwort-Message. Das erzeugende Programm (das die Antwort erhält) kann dadurch MQSeries beauftragen, nach einer spezifischen Nachricht zu suchen. Standardmäßig erhält es die erste Message in der Queue.
- Man kann einer Message eine Priorität zuordnen und somit die Reihenfolge, in der sie verarbeitet wird, steuern.

- Der Queue-Manager speichert Zeit und Datum, wenn im Message-Header "MQPUT" erscheint. Die Zeit ist GMT-kompatibel, das Jahr hat 4 Stellen und ist so in Übereinstimmung mit Y2K.
- Ein Verfallsdatum kann auch spezifiziert werden. Wenn dieses Datum erreicht und ein MQGET ausgegeben wird, dann wird die Nachricht gelöscht. Es existiert kein Dämon, der die Queues auf gelöschte Messages hin überprüft. Letztere können in einer Queue solange stehen, bis ein Programm versucht, sie zu lesen.
- Die Antwort-Adresse ist sehr wichtig für die Request/Antwort-Messages. Es muss dem Server-Programm mitgeteilt werden, wohin die Antwort-Message gesendet werden soll. Clients und Server verfügen über eine one-to-many-Beziehung, und das Server-Programm kann normalerweise nicht aus den Nutzerdaten schließen, woher die Request-Message kommt. Deshalb stellt der Client das Reply-to-Queue und den Reply-to-Queue-Manager in dem Message-Header zur Verfügung. Der Server benutzt diese Information beim Ausführen des MQPUT-API-Call's.
- Der Sender kann in dem Format-Feld einen Wert spezifizieren, den der Empfänger benutzt, um eine Daten-Konvertierung vorzunehmen oder nicht. Er wird auch benutzt, um zu zeigen, dass ein zusätzlicher Header vorhanden ist.
- Die Message kann auch Informationen über die sendende Anwendung (Programm-Name und Pfad) und die Plattform, auf der diese läuft, enthalten.
- Report-Optionen und Rückkehr-Code werden verwendet, um Informationen wie Bestätigung vom Eintreffen oder Lieferung vom empfangenden Queue-Manager anzufordern. Beispielsweise kann der Queue-Manager eine Report-Message an die sendende Applikation schicken, wenn er die Message auf die Ziel-Queue legt oder die Applikation sie von der Queue bekommt.
- Jedesmal, wenn eine Message zurückkommt, wird der Backout-Zähler erhöht. Eine Anwendung kann diesen Zähler überprüfen und z.B. die Message zu einer anderen Queue schicken, wo die Fehlerursache durch einen Administrator analysiert wird.
- Der Queue-Manager benutzt den Message-Header, um Informationen über die physikalische Message zu speichern, z.B. wenn es sich um eine Message-Group handelt, dann das erste oder letzte Segment oder eines dazwischen.

7.3 Queue-Manager

7.3.1 Einführung

Den Kern von MQSeries bildet der Message-Queue-Manager (MQM). Der MQM wird durch das MQSeries-Run-Time-Programm implementiert. Seine Aufgabe besteht darin, die Queues und die Messages zu verwalten. Er stellt das Message-Queuing-Interface für die Kommunikation mit den Anwendungen zur Verfügung. Die Applikations-Programme rufen Funktionen des Queue-Managers durch Ausgabe von API-Call's auf. Der MQPUT-API-Call z.B. legt eine Message auf eine Queue, die von einem anderen Programm mit Hilfe eines MQGET-API-Call's gelesen werden soll. Dieser Mechanismus ist in der Abbildung 4 dargestellt.

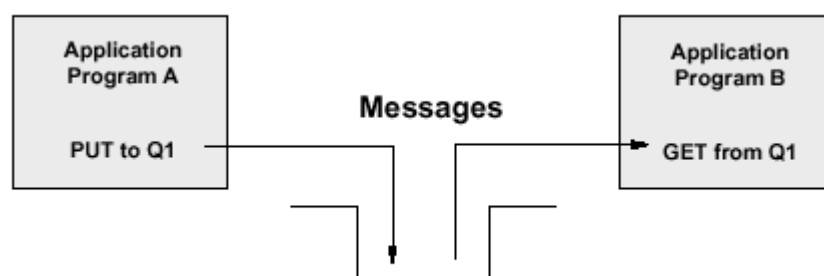


Figure 4. Program-to-Program Communication - One System

Ein Programm kann Nachrichten an ein anderes Programm (-Ziel) schicken. Das Ziel-Programm läuft dabei entweder auf derselben Maschine wie der Queue-Manager oder auf einem Remote-System (Server oder Host). Das Remote-System verfügt über seinen eigenen Queue-Manager bzw. seine eigenen Queues. Abbildung 5 zeigt diesen Fall.

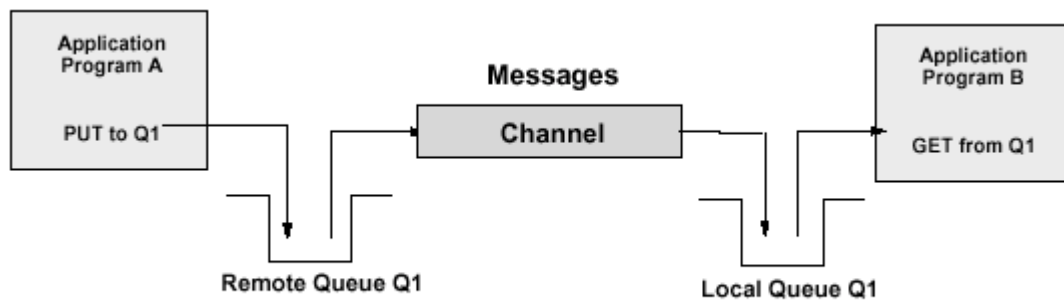


Figure 5. Program-to-Program Communication - Two Systems

Der Queue-Manager überträgt Nachrichten zu einem anderen Queue-Manager über Channels, wobei bestehende Netzwerk-Facilities wie TCP/IP, SNA oder SPX verwendet werden. Auf derselben Maschine können mehrere Queue-Manager residieren. Letztere benötigen aber Channels für die Kommunikation. Der Anwendungs-Programmierer braucht sich nicht darum zu kümmern, wo das Programm, an das er Messages schickt, abläuft. Er legt seine Nachrichten auf eine Queue und überlässt es dem Queue-Manager, die Ziel-Maschine zu suchen und die Nachrichten auf diese zu übertragen. MQSeries weiß, was zu tun ist, wenn das Remote-System nicht verfügbar oder das Ziel-Programm beschäftigt ist bzw. nicht läuft.

Die Arbeit des Queue-Managers bezieht sich auf Objekte, die durch einen Administrator im allgemeinen beim Generieren des Queue-Managers oder mit einer neuen Applikation definiert werden. Der Queue-Manager übernimmt folgende Aufgaben:

- Er verwaltet die Message-Queues der Applikations-Programme.
- Er liefert ein Applikations-Programm-Interface: Message Queue Interface (MQI).
Das Networking Blueprint unterscheidet 3 Kommunikations-Stile:
 1. Common Programming Interface-Communications (CPI-C)
 2. Remote Procedure Call (RPC)
 3. Message Queue Interface (MQI)
- Er benutzt vorhandene Netzwerk-Facilities, um Messages zu anderen Queue-Managern zu übertragen, wenn es notwendig ist.
- Er ordnet Updates von Datenbanken und Queues durch Verwendung einer 2-Phasen-Übergabe an. GET's und PUT's von/auf Queues werden zusammen mit SQL-Updates übergeben oder wenn nötig zurückgeschickt.
- Er segmentiert, wenn es erforderlich ist, die Nachrichten und übersetzt sie. Er kann auch Messages gruppieren und diese als physikalische Message zu ihrem Ziel senden, wo diese automatisch disassembliert wird.
- Er kann eine Message zu mehr als einem Ziel mit Hilfe eines API-Call's schicken. Dabei wird eine Nutzer-definierte dynamische Distribution-List verwendet, wodurch sich der Netzwerk-Verkehr reduziert.
- Er liefert zusätzliche Funktionen, die es dem Administrator erlauben, Queues zu erzeugen und zu löschen, die Eigenschaften existierender Queues zu verändern und die Operationen des Queue-Managers zu steuern. MQSeries für Windows NT, Version 5.1 stellt graphische User-Interfaces zur Verfügung. Andere Plattformen benutzen das Command-Line-Interface oder Panels.

MQSeries-Clients besitzen in ihren Maschinen keinen Queue-Manager. Client-Rechner verbinden sich mit einem Queue-Manager in einem Server. Dieser Queue-Manager verwaltet die Queues für alle Clients, die mit dem Server verbunden sind.

Im Gegensatz zu MQSeries-Clients hat jeder Rechner, auf der MQSeries for Windows (Version 2) läuft, ihren eigenen Queue-Manager und Queues. MQSeries for Windows ist ein Single-User-Queue-Manager und ist nicht für die Funktion des Queue-Managers anderer MQSeries-Clients vorgesehen. Das Produkt ist für mobile Umgebungen entwickelt worden.

MQSeries for Windows und for Windows NT stellen zwei unterschiedliche Produkte dar.

7.3.2 Queue-Manager-Cluster

MQSeries für MVS/ESA und Version 5.1 für verteilte Plattformen erlauben es, Queue-Manager in Clustern miteinander zu verbinden. Queue-Manager in einem Cluster können auf einem oder auf unterschiedlichen Rechnern verschiedener Plattformen laufen. Normalerweise unterhalten zwei dieser "Cluster Queue Manager" ein Repository. Letzteres beinhaltet Informationen über alle Queue-Manager und Queues im betreffenden Cluster. Dieses Repository wird als "Full-Repository" bezeichnet. Die anderen Queue-Manager unterhalten nur ein Repository von interessierenden Objekten. Dieses heißt dagegen "Partial-Repository". Es gestattet, dass irgendein Queue-Manager in dem Cluster eine beliebige Cluster-Queue und deren Besitzer findet. Die Queue-Manager benutzen für den Austausch spezielle Cluster-Channels.

Das Clustering gestattet auch mehrfache Queue-Instanzen mit demselben Namen in verschiedenen Queue-Managern. Das bedeutet für das Workload, dass der Queue-Manager Nachrichten an unterschiedliche Instanzen einer Anwendung schicken kann. In einer normal verteilten Verarbeitung wird eine Message an eine spezifische Queue eines bestimmten Queue-Managers geschickt. Alle Nachrichten, die für diesen speziellen Queue-Manager bestimmt sind, werden in einer Transmission-Queue auf der Sender-Seite abgelegt. Diese Transmission-Queue hat denselben Namen wie der Ziel-Queue-Manager. Die Message-Channel-Agenten bewegen die Nachrichten durch das Netzwerk und platzieren sie in den Ziel-Queues. Die Abbildung 6 zeigt die Beziehung zwischen einer Transmission (Xmit)-Queue und dem Ziel-Queue-Manager.

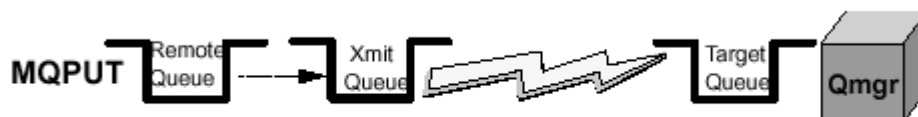


Figure 6. MQPUT to a Remote Queue

Beim Clustern wird eine Message an eine Queue mit einem speziellen Namen irgendwo in dem Cluster geschickt (dargestellt durch die Wolke in Abbildung 7). Man spezifiziert den Ziel-Queue-Namen, nicht den Namen der Remote-Queue-Definition. Das Clustern erfordert keine Remote-Queue-Definitionen. Letztere sind nur dann erforderlich, wenn eine Message an einen Queue-Manager gesendet wird, der nicht zum Cluster gehört. Es ist auch möglich, einen Queue-Manager zu bestimmen und die Nachricht an eine spezielle Queue zu richten, oft wird es aber dem Queue-Manager überlassen, darüber zu entscheiden, wo sich eine Queue befindet und wohin die Nachricht gesendet werden soll.

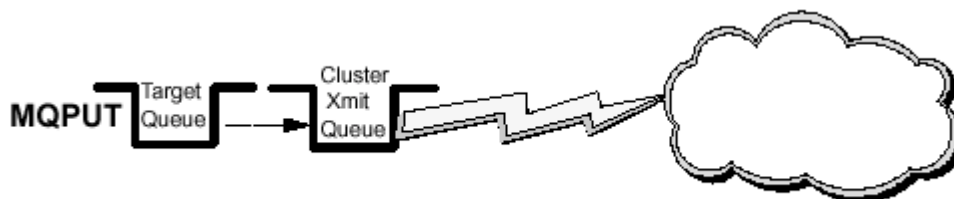


Figure 7. MQPUT to a Cluster Queue

Die Vorstellung eines MQSeries-Clusters besteht in einer bestimmten Lokalität, wo mehrfache Queue-Instanzen existieren können, die sich verändern, wie ein Administrator die Erfüllung von wechselnder Verfügbarkeit und

wechselndem Durchsatz verlangt. Dieses muss rein dynamisch erfolgen und ohne großen Aufwand bei der Konfiguration und Steuerung seitens des Administrators. Zusätzlich muss sich der Programmierer keine Gedanken über mehrfache Queues machen, d.h. er behandelt diese so wie beim Schreiben auf eine einzelne Queue. Letzteres bedeutet aber nicht, dass dem Programmierer oder Administrator sämtliche Arbeit abgenommen wird. Z. B. erfordern entsprechende Stufen der Verfügbarkeit und Ausnutzung von inhärenter Parallelität umfangreiche Überlegungen. Der Administrator bzw. System-Designer muss sicherstellen, dass in der festgelegten Konfiguration genügend Redundanz vorhanden ist. Der Anwender ist dafür verantwortlich, dass die Nachrichten an mehreren Stellen verarbeitet werden können.

Man erzeugt mehrfache Instanzen einer Queue durch eine Queue-Definition mit demselben Namen in mehrfachen Queue-Managern, die zu dem Cluster gehören. Es muss auch das Cluster benannt werden, wenn man die Queue definiert. Ohne dieses Attribut wäre die Queue nur lokal bekannt. Wenn die Applikation nur den Queue-Namen spezifiziert, wohin wird dann die Message eigentlich geschickt?

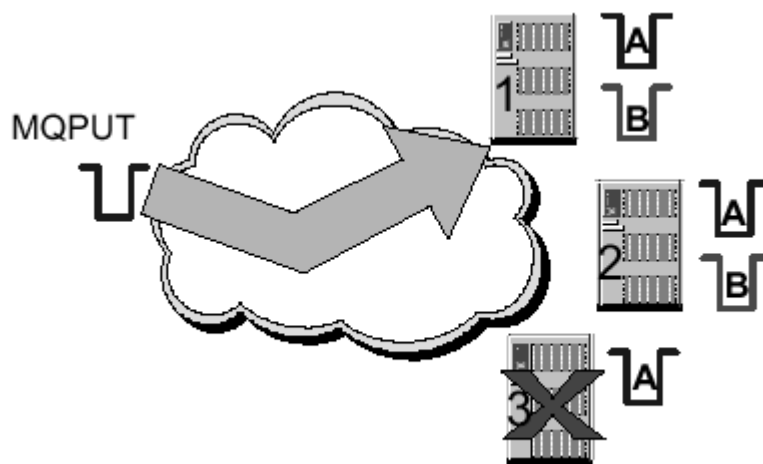


Figure 8. Accessing Cluster Queues

Die Abbildung 8 gibt eine Antwort auf diese Frage. MQSeries verteilt die Nachrichten nach dem Round-Robin-Algorithmus. Es ist natürlich auch möglich, diese Standard-Aktion durch Schreiben einer eigenen Workload-Ausgabe-Routine zu verändern. In der Abbildung 8 sind Nachrichten dargestellt, die in einer von drei Cluster-Queues mit dem Namen "A" abgelegt werden. Jeder der drei Queue-Manager besitzt eine Queue mit diesem Namen. Standardmäßig wird die erste Message in der Queue A des Queue-Managers 1 platziert, die nächste in der Queue A des Queue-Managers 2, die dritte geht in Queue A vom Queue-Manager 3 und die vierte wieder zur Queue des Queue-Managers 1 usw. Ein anderes Szenario enthält die Queue B, wobei der dritte Queue-Manager nicht aktiv und die dritte Instanz der Queue B nicht verfügbar ist. Der sendende Queue-Manager erfährt von diesem Problem, weil er Informationen über alle Queue-Manager und Queues, die für ihn interessant sind, erhält, d.h. auch über die, wohin er die Nachrichten sendet. Sobald er herausfindet, dass es ein Problem mit der dritten Instanz von B gibt, verteilt der sendende Queue-Manager die Messages nur an die ersten beiden Instanzen. Spezielle Nachrichten über Zustandsänderungen der Cluster-Objekte werden sofort allen an dem Objekt beteiligten Queue-Managern bekannt gemacht.

7.3.3 Queue-Manager-Objekte

Zu den Queue-Manager-Objekten zählen z.B. Queues und Channels. Der Queue-Manager selbst ist auch ein Objekt. Normalerweise erzeugt ein Administrator ein oder mehrere Queue-Manager und seine Objekte. Ein Queue-Manager kann Objekte der folgenden Typen benutzen:

1. Queues
2. Process-Definitionen
3. Channels

Diese Objekte gelten für alle unterschiedlichen MQSeries-Plattformen. Es gibt noch andere Objekte, die nur für MVS-Systeme gültig sind, z.B. Buffer Pool, PSID und Storage Class. AS/400-MQ-Objekte sind dem entsprechenden Betriebssystem als Objekt-Typen USRSPC in der QMQMDATA-Bibliothek bekannt.

Queues

Message Queues werden verwendet, um die von Programmen gesendeten Nachrichten zu speichern. Es gibt lokale Queues, die der lokale Queue-Manager besitzt, und remote Queues, die zu einem dazu verschiedenen Queue-Manager gehören.

Channels

Ein Channel stellt einen Kommunikationspfad dar. Dabei wird zwischen 2 Channel-Typen unterschieden:

- Message Channel
- Message-Queue-Interface (MQI)-Channel

Der Message-Channel verbindet zwei Queue-Manager über Message-Channel-Agenten (MCAs). Ein Channel ist unidirektional; er integriert zwei Message-Channel-Agenten, einen Sender und einen Empfänger, und ein Kommunikations-Protokoll. Ein MCA stellt ein Programm dar, das Nachrichten von einer Transmission-Queue zu einem Kommunikations-Link und von dem Kommunikations-Link in die Ziel-Queue überträgt. Für die bidirektionale Kommunikation müssen zwei Channels mit jeweils einem Sender und einem Empfänger definiert werden. Message-Channel-Agenten werden auch als Treiber (mover) bezeichnet.

Ein Message-Queue-Interface-Channel verbindet einen MQSeries-Client mit einem Queue-Manager in seiner Server-Maschine. Clients besitzen keinen eigenen Queue-Manager. Ein MQI-Channel ist immer bidirektional (MQI-Call und Antwort).

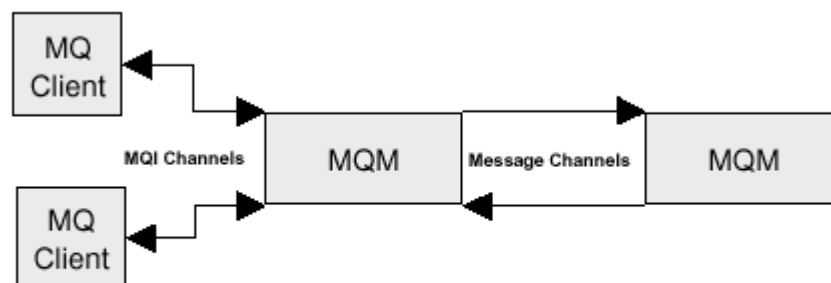


Figure 9. MQSeries Channels

Die Abbildung 9 zeigt beide Channel-Typen. Es sind insgesamt vier Maschinen dargestellt, zwei Clients sind mit ihrer Server-Maschine über MQI-Channels und der Server ist mit einem anderen Server oder Host mittels zweier unidirektionaler Message-Channels verbunden. Einige Channels können von MQSeries automatisch definiert werden. Abhängig davon, wie die Sitzung zwischen den Queue-Managern initiiert wird und welchem Zweck sie dient, existieren unterschiedliche Typen von Message-Channels. Für die Übertragung nicht-persistenter Nachrichten kann ein Message-Channel mit zwei verschiedenen Geschwindigkeiten arbeiten: Fast und Normal. Fast-Channels verbessern die Performance, aber nicht-persistente Messages können im Fall eines Channel-Fehlers verloren gehen.

Ein Channel ist in der Lage, folgende Transport-Typen zu benutzen: SNA LU 6.2, TCP/IP, NetBIOS, SPX, DEC Net. Der Support ist außerdem noch von der Art der Plattform abhängig. MQSeries for Windows, Version 2, benutzt für die Verbindung mit anderen Maschinen Message-Channels. Da dieses Produkt als Single-User-System entwickelt wurde, unterstützt es keine MQI-Channels (nur TCP/IP).

Process-Definitionen

Ein Process-Definition-Objekt definiert eine Applikation für einen Queue-Manager. Es enthält z.B. den Namen des Programms (und seinen Pfad), das getriggert werden soll, sobald eine Message dafür eintrifft.

7.4 Message-Queues

7.4.1 Queue-Arten

Queues werden als Objekte, die zu einem Queue-Manager gehören, definiert. MQSeries kennt eine bestimmte Anzahl von unterschiedlichen Queue-Typen, jeder für einen spezifischen Zweck. Die benutzten Queues sind entweder auf der lokalen Maschine platziert und gehören zu dem Queue-Manager, mit dem die lokale Maschine verbunden ist, oder sie liegen auf dem Server (wenn die lokale Maschine den Client darstellt). Die Abbildung 10 zeigt verschiedene Queue-Typen und ihren Zweck.

Local queue	is a real queue
Remote queue	structure describing a queue
Transmission queue (xmitq)	local queue with special purpose
Initiation queue	local queue with special purpose
Dynamic queue	local queue created "on the fly"
Alias queue	if you don't like the name
Dead-letter queue	one for each queue manager
Reply-to-queue	specified in request message
Model queue	model for local queues
Repository queue	holds cluster information

Figure 10. Queue Types

Lokale Queue

Eine Queue heißt lokal, wenn sie sich im Besitz des Queue-Managers befindet, mit dem das Applikations-Programm verbunden ist. Sie wird zur Speicherung von Nachrichten für Programme benutzt, die denselben Queue-Manager verwenden. Z.B. haben die Programme A und B je eine Queue für eingehende und ausgehende Messages. Da der Queue-Manager beide Programme bedient, sind alle vier Queues lokal.

Beide Programme müssen nicht auf demselben Rechner laufen. Die Client-Maschine benutzt im Normalfall einen Queue-Manager auf dem Server.

Cluster-Queue

Eine Cluster-Queue ist eine lokale Queue, die überall in einem Cluster von Queue-Managern bekannt ist, d.h. irgendein Queue-Manager, der zu dem Cluster gehört, kann Nachrichten an sie schicken, ohne eine Remote-Definition oder Channels zu dem Queue-Manager, dem sie gehört, zu definieren.

Remote-Queue

Eine Queue heißt "remote", wenn sie zu einem anderen Queue-Manager gehört. Die Remote-Queue stellt keine reale Queue dar. Sie implementiert eine Struktur, die etwas von den Charakteristiken der Queue eines anderen Queue-Managers enthält.

Der Name einer Remote-Queue kann genauso wie der einer lokalen Queue benutzt werden. Der MQSeries-Administrator definiert, wo die Queue aktuell ist. Remote-Queues werden mit einer Transmission-Queue verknüpft. Ein Programm ist nicht in der Lage, Nachrichten von einer Remote-Queue zu lesen.. Für eine Cluster-Queue ist keine Remote-Queue-Definition erforderlich.

Transmission-Queue

Die Transmission-Queue ist eine spezielle lokale Queue. Eine Remote-Queue ist verbunden mit einer Transmission-Queue. Transmission-Queues werden als Zwischenschritt benutzt, wenn Nachrichten an Queues gesendet werden, die unterschiedlichen Queue-Managern angehören. Typischerweise existiert nur eine Transmission-Queue für jeden Remote-Queue-Manager (oder Rechner). Für alle Messages, die auf Queues mit einem Remote-Queue-Manager als Besitzer abgelegt werden, erfolgt die Abspeicherung zunächst in einer Transmission-Queue dieses Remote-Queue-Managers. Die Messages werden dann von der Transmission-Queue gelesen und an den Remote-Queue-Manager geschickt.

Bei der Benutzung von MQSeries-Clustern gibt es nur eine Transmission-Queue für alle Messages, die an alle anderen Queue-Manager in dem Cluster gesendet werden.

Die Transmission-Queues sind für die Applikation transparent. Sie werden intern von dem Queue-Manager benutzt. Wenn ein Programm eine Remote-Queue öffnet, erhält es die Queue-Attribute von der Transmission-Queue, d.h. das Schreiben von Messages zu einer Queue durch ein Programm wird durch die Transmission-Queue-Charakteristiken ermöglicht.

Dynamic Queue

Eine Dynamic-Queue wird "on the fly" definiert, wenn sie von einer Applikation benötigt wird. Dynamic-Queues können vom Queue-Manager belegt oder automatisch gelöscht werden, sobald das Anwendungsprogramm endet. Sie stellen lokale Queues dar und werden oft in der Sprachverarbeitung zur Speicherung von Zwischenergebnissen benutzt. Dynamic-Queues können sein:

- Temporäre Queues, deren Lebensdauer mit einem Queue-Manager-Restart endet
- Permanente Queues, die Queue-Manager-Restarts überleben

Alias-Queue

Alias-Queues sind keinen realen Queues sondern Definitionen. Sie werden verwendet, um derselben physikalischen Queue unterschiedliche Namen zuzuweisen. Dadurch können mehrere Programme mit derselben Queue arbeiten, indem auf diese mit unterschiedlichen Namen und verschiedenen Attributen zugegriffen wird.

Generieren eines Queue-Managers

Das Erzeugen eines Queue-Managers erfolgt mit dem Kommando

```
crtmqm
```

Standardmäßig wird noch der Parameter /q spezifiziert. Das folgende Kommando generiert den Default-Queue-Manager MYQMGR (in einer Windows NT-Umgebung)

```
crtmqm /q MYQMGR
```

(Queue-Manager-Namen sind Case-sensitive)

Es gibt Standard-Definitionen für die von jedem Queue-Manager benötigten Objekte, z.B. Modell-Queues. Letztere Objekte werden automatisch erzeugt. Andere Objekte, die sich auf die Applikationen beziehen, müssen vom Anwender generiert werden. Gewöhnlich befinden sich Anwendungs-spezifische Objekte in einem Script-File (mydefs.in). Es wird einem neu-generierten Queue-Manager mit dem Kommando

```
runmqsc < mydefs.in
```

übergeben.

MQSeries for Windows NT, Version 5.1, liefert ein graphisches Nutzer-Interface, mit dem Queue-Manager und ihre Objekte erzeugt und manipuliert werden. Eine Dead-Letter-Queue wird nicht automatisch erzeugt; sie wird mit dem Queue-Manager wie in dem folgenden Beispiel generiert:

```
crtmqm /q /u system.dead.letter.queue MYQMGR
```

Zum Start des Queue-Managers ist das Kommando "strmqm" notwendig.

7.4.2 Events

Um die Operationen des Queue-Managers zu verfolgen, können die MQSeries-Instrumentations-Events benutzt werden. Diese Events generieren spezielle Nachrichten (Event Messages), d.h. immer dann, wenn der Queue-Manager eine vordefinierte Menge von Bedingungen anzeigt. Z.B. verursachen folgende Bedingungen einen Queue-Voll-Event:

- Queue-Voll-Events werden aktiviert für eine spezifizierte Queue.
- Eine Anwendung gibt einen MQPUT-Call aus, um eine Nachricht auf dieser Queue abzulegen; der Call ist aber fehlerhaft, weil die Queue voll ist.

Andere Bedingungen können Instrumentation-Events verursachen:

- Wenn eine vordefinierte Begrenzung für die Anzahl der Nachrichten erreicht wird.
- Wenn eine Queue innerhalb einer festgelegten Zeit bedient wird.
- Wenn eine Channel-Instanz gestartet oder gestoppt wird.

Wenn Event-Queues als Remote-Queues definiert werden, dann können alle Event-Queues von einem einzelnen Queue-Manager (für die Knoten, die Instrumentation-Events unterstützen) bearbeitet werden.

MQSeries-Events können in folgende Kategorien eingeteilt werden:

Queue-Manager-Events: Diese Events beziehen sich auf die Ressource-Definitionen innerhalb des Queue-Managers. Wenn z.B. eine Anwendung versucht, eine Queue zu öffnen, aber das zugehörige UserID ist nicht für diese Operation autorisiert, dann wird ein Queue-Manager-Event generiert.

Performance-Events: Diese Events werden erzeugt beim Erreichen eines Ressource-Schwellenwertes. Als Beispiel dient eine Queue, deren Tiefe erreicht wird und noch ein MQGET-Request erhält.

Channel-Events: Diese Events werden von Channels als Ergebnis von Bedingungen, die während ihrer Operation erscheinen, angezeigt. Beispiel: Ein Channel-Event wird generiert, wenn eine Channel-Instanz gestoppt wird.

7.5 Manipulation von Queue-Manager-Objekten

MQSeries für verteilte Plattformen stellt die Utility RUNMQSC zur Verfügung. Letztere generiert und löscht Queue-Manager-Objekte und manipuliert sie. Um dieses Werkzeug zu benutzen, muss der Queue-Manager laufen. RUNMQSC kann unterschiedlich aktiviert werden:

- Durch Eingeben des Kommandos.
- Durch Erzeugen eines Files, das eine Liste von Kommandos enthält, und Benutzung diese Files als Eingang.

In der Abbildung 11 starten die Kommandos den Default-Queue-Manager (der schon läuft, wie die Antwort zeigt) und generieren die lokale Queue QUEUE1 für ihn. Ein anderes Kommando ändert die Queue-Manager-Eigenschaften, um die Dead-Letter-Queue zu definieren.

Um die Utility in einem interaktiven Modus zu starten, wird "runmqsc" eingegeben. Zum Beenden wird das Kommando "end" verwendet. Die andere Methode, MQSeries-Objekte zu erzeugen, besteht darin, ein Input-File zu verwenden, z.B. kann das mit Hilfe des folgenden Kommandos erreicht werden:

```
runmqsc < mydefs.in > a.a
```

mydefs.in gibt das Script-File mit den Kommandos an; a.a stellt das File dar, das die Antworten der RUNMQSC-Utility aufnehmen soll. Damit kann überprüft werden, ob irgendein Fehler aufgetreten ist. Der Ausgang erscheint entweder in dem Fenster oder wird in das File umgeleitet.

```
C:\strmqm
MQSeries queue manager running.

runmqsc
84H2001,6539-B42 (C) Copyright IBM Corp. 1994, 1997. ALL RIGHTS RESERVED
Starting MQSeries Commands.

define qlocal('QUEUE1') replace descr ('test queue')
  1 : define qlocal('QUEUE1') replace descr ('test queue')
AMQ8006: MQSeries queue created.
alter qmgr deadq(system.dead.letter.queue)
  2 : alter qmgr deadq(system.dead.letter.queue)
AMQ8005: MQSeries queue manager changed.
end
  3 : end
2 MQSC commands read.
0 commands have a syntax error.
0 commands cannot be processed.

C:\
```

Figure 11. Manipulating Objects Using Control Commands

7.6 Clients und Server

MQSeries unterscheidet Clients und Server. Vor der Installation von MQSeries auf einer verteilten Plattform muss der Anwender entscheiden, ob die Maschine die Funktion eines MQSeries-Clients, eines MQSeries-Servers oder die beider übernehmen soll.

Mit MQSeries for Windows wurde ein neuer Term eingeführt: Leaf Node (Blatt-Knoten).

Man unterscheidet zwei Arten von Clients:

- Slim-Client oder MQSeries-Client
- Fat-Client

Fat-Clients verfügen über einen lokalen Queue-Manager, Slim-Client besitzen diesen nicht. Wenn ein Slim-Client nicht mit seinem Server verbunden werden kann, ist ersterer nicht arbeitsfähig, weil der Queue-Manager und die Queues des Slim-Clients auf dem Server liegen. Im Normalfall ist ein MQSeries-Client identisch mit einem Slim-Client. Mehrere dieser Clients teilen sich die MQSeries-Objekte, wobei der Queue-Manager auf dem Server eines dieser Objekte darstellt. Der MQSeries-Client for Java ist ein Slim-Client. In manchen Fällen kann es vorteilhaft sein, dass die End-Nutzer-Maschine (speziell in einer mobilen Umgebung) Queues besitzt. Dieser Umstand erlaubt es dem Nutzer, seine Anwendung auch dann laufen zu lassen, wenn keine Verbindung (temporär) zwischen Client und Server besteht.

Client- und Server-Software kann auf demselben System installiert werden. Unter dem Betriebssystem Windows NT sind MQSeries for Windows NT V5.1 oder MQSeries For Windows V2.1 (MQWin) lauffähig. Für Windows 95 ist MQWin V2.1 geeignet. Letzteres Produkt ist für End-Nutzer entwickelt worden und verwendet weniger Ressourcen. Der Unterschied zwischen einem Rechner eines End-Nutzers, der als Client arbeitet, und einem mit einem Queue-Manager besteht in dem Nachrichten-Weg. Die Queues liegen entweder auf der Maschine des End-Nutzers oder auf dem Server.

Die Abbildung 12 zeigt wieder die Nutzung von MQI und der Message-Channels.

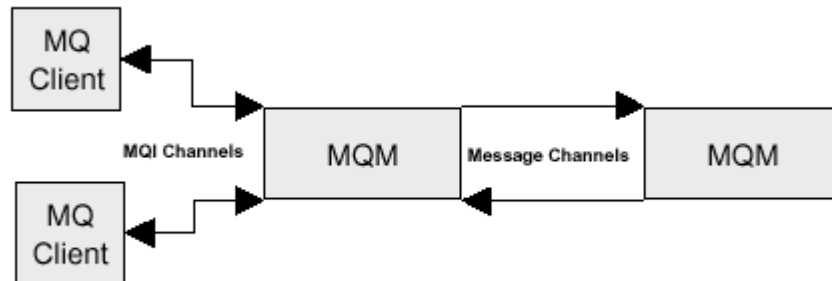


Figure 12. MQI and Message Channels

- MQI-Channels verbinden Clients mit einem Queue-Manager in einer Server-Maschine. Alle MQSeries-Objekte für den Client befinden sich auf dem Server. MQI-Channels sind schneller als Message-Channels.
- Ein Message-Channel verbindet einen Queue-Manager mit einem anderen Queue-Manager. Letzterer kann auf derselben oder auf einer andern Maschine residieren.

Die drei verschiedenen Rechner-Typen sind:

MQSeries-Client

Ein Client-Rechner besitzt keinen eigenen Queue-Manager. Er teilt einen Queue-Manager in einem Server mit anderen Clients. Alle MQSeries-Objekte befinden sich auf dem Server. Da die Verbindung zwischen Client und Server synchron ist, kann die Applikation nicht laufen, wenn die Kommunikation unterbrochen ist. Solche Maschinen werden als Slim-Clients bezeichnet.

MQSeries-Server

Ein Rechner kann als Client und als Server arbeiten. Ein Server implementiert einen Zwischen-Knoten innerhalb anderer Knoten. Er bedient Clients, die keinen Queue-Manager haben und verwaltet den Nachrichtenfluss zwischen seinen Clients, seinen eigenen und den zwischen ihm und anderen Servern. Zusätzlich zu der Server-Software kann auch die des Client installiert werden. Diese Konfiguration wird in einer Applikations-Entwicklungsumgebung verwendet.

Leaf-Node

MQSeries for Windows wurde für die Nutzung durch einen Single-User entworfen. Dieses System hat seinen eigenen "Small Footprint"-Queue-Manager mit eigenen Objekten. Es ist aber kein "Intermediate Node" unter anderen Knoten und heißt "Leaf Node". Man kann es auch "als Fat-Client" bezeichnen. Dieses Produkt ist in der Lage, ausgehende Nachrichten in eine Warteschlange einzureihen, wenn die Verbindung zu einem Server oder Host nicht verfügbar und eingehende Messages, wenn die entsprechende Applikation nicht aktiv ist.

7.7 MQSeries-Architektur

Die Bausteine und die Gesamtarchitektur von MQSeries ist in der Abbildung 13 wiedergegeben. Das Applikations-Programm benutzt das Message-Queue-Interface (MQI), um mit dem Queue-Manager zu kommunizieren. Das MQI wird in Kapitel 7.11 detaillierter beschrieben. Das Queuing-System besteht aus folgenden Teilen:

- Queue-Manager (MQM)
- Listener
- Trigger-Monitor
- Channel-Initiator

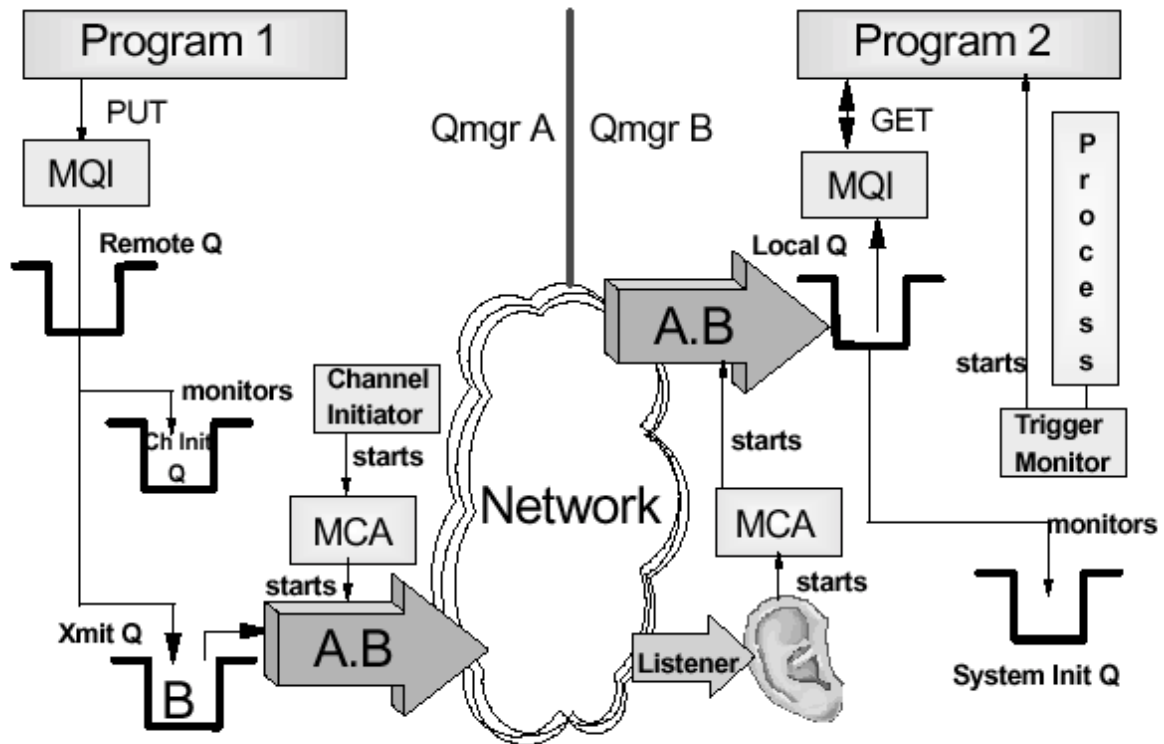


Figure 13. MQSeries - Parts and Logic

- Message-Channel-Agent (MCA oder Mover)

Wenn das Anwendungsprogramm eine Message auf eine Queue legen möchte, gibt es einen "MQPUT API"-Call aus. Letzterer ruft das MQI auf. Der Queue-Manager überprüft, ob die im Ruf referenzierte Queue "local" oder "remote" ist. Wenn es sich um eine Remote-Queue handelt, wird die Message in der Transmission-Queue (Xmit) abgelegt. Der Queue-Manager addiert einen Header hinzu, der Informationen der Remote-Queue-Definition sowie den Namen des Ziel-Queue-Managers und der Ziel-Queue enthält. Dabei muss jede Remote-Queue mit einer Xmit-Queue verbunden werden. Normalerweise benutzen alle Messages, die für eine Remote-Maschine bestimmt sind, dieselbe Xmit-Queue.

Die Übertragung erfolgt über Channels, die manuell oder automatisch gestartet werden können. Für den automatischen Start eines Channels muss die Xmit-Queue mit einer Channel-Initiation-Queue verbunden werden. Aus der Abbildung 13 ist erkennbar, dass der Queue-Manager eine Nachricht auf die Xmit-Queue und eine andere Nachricht auf die Channel-Initiation-Queue legt. Letztere Queue wird überwacht durch den Channel-Initiator. Dieser stellt ein MQSeries-Programm dar, das für die Überwachung der Initiation-Queues laufen muss. Wenn der Channel-Initiator eine Message in der Initiation-Queue erkennt, startet er den Message-Channel-Agent (MCA) für den bestimmten Channel. Dieses Programm bewegt die Nachricht über das Netzwerk zu einer anderen Maschine, indem es den Senderteil des unidirektionalen Message-Channel-Paares benutzt.

Auf der Empfangsseite muss ein Listener-Programm gestartet werden. Der Listener, der auch mit MQSeries geliefert wird, überwacht einen speziellen Port, der für MQSeries standardmäßig mit 1414 festgelegt ist. In dem Fall, dass eine Message eintrifft, startet er den Message-Channel-Agent. Letzterer bewegt die Message in die spezielle lokale Queue, wobei er den Empfängerseite des Message-Channel-Paares verwendet. Beide Channel-Definitionen, Sender und Empfänger, müssen denselben Namen tragen. Für die Antwort wird ein anderes Message-Channel-Paar gebraucht.

Das Programm, das die eintreffende Message verarbeitet, kann manuell oder automatisch gestartet werden. Um das Programm automatisch zu starten, ist es notwendig, eine Initiation-Queue und einen Prozess mit der lokalen Queue zu verbinden. Zusätzlich muss sich der Trigger-Monitor im Zustand "Run" befinden. Wenn das Programm automatisch startet, legt der MCA die eintreffende Message in die lokale Queue und eine Trigger-Message in die Initiation-Queue. Letztere Queue wird von dem Trigger-Monitor überwacht. Dieses Programm ruft das Applikations-Programm auf, das in der Prozess-Definition spezifiziert ist. Die Applikation gibt einen "MQGET API"-Ruf aus, um die Message von der lokalen Queue zu finden.

7.8 Kommunikation zwischen Queue-Managern

7.8.1 Einführung

Ausgangspunkt bilden die Message-Channels für die Kommunikation zwischen Queue-Managern in der Abbildung 12. Der Mechanismus ist in der Abbildung 14 dargestellt, und die notwendigen MQSeries-Definitionen können der Abbildung 15 entnommen werden. Jede Maschine verfügt über einen installierten Queue-Manager und jeder Queue-Manager verwaltet verschiedene lokale Queues. Die für einen Remote-Queue-Manager bestimmten Nachrichten werden in einer Remote-Queue abgelegt. Letztere stellt keine reale Queue dar, sie ist die Definition einer lokalen Queue in der Remote-Maschine. Die Remote-Queue wird mit einer Transmission (Xmit)-Queue, die eine lokale Queue implementiert, verbunden. Gewöhnlich existiert eine Xmit-Queue für jeden Remote-Queue-Manager.

Eine Transmission-Queue wird mit einem Message-Channel verknüpft. Diese Channels sind unidirektional, d.h. es müssen zwei Channels für eine Konversations-Kommunikation definiert werden. Jeder Channel muss zweimal definiert werden, einmal in dem System, das die Nachricht sendet (Sender-Channel) und einmal in dem System, das die Nachricht empfängt (Receiver-Channel). Jedes Channel-Paar (Sender und Receiver) besitzen denselben Namen. Dieses Szenario ist in der Abbildung 14 dargestellt.

7.8.2 Definition der Verbindung zwischen zwei Systemen

Für die Verbindung von zwei Queue-Managern sind in jedem System notwendig (s. Abbildung 14):

- Eine Remote-Queue-Definition, die die lokale Queue in der Empfänger-Maschine spiegelt und sie mit einer Transmission-Queue (Q1 im System A und Q2 im System B) verbindet.
- Eine Transmission-Queue, die alle für das Remote-System bestimmten Messages solange enthält, bis der Kanal sie überträgt (QMB im System A und QMA im System B).
- Ein Sender-Channel, der Nachrichten von der Xmit-Queue bekommt und diese zum anderen System durch das existierende Netzwerk überträgt (QMA.QMB im System A und QMB.QMA im System B).
- Ein Receiver-Channel, der die Nachrichten empfängt und sie in einer lokalen Queue (QMB.QMA im System A und QMA.QMB im System B) ablegt; Receiver-Channels können durch den Queue-Manager automatisch gestartet werden, wenn die Channel-Auto-Definition (CHAD) aktiviert ist.
- Eine lokale Queue, von der das Programm seine Messages erhält (Q2 im System A und Q1 im System B).

In jedem System müssen geeignete Queue-Manager-Objekte definiert werden. Das erfolgt mit Hilfe zweier Script-Files (s. Abbildung 15).

Beim Clustering müssen die Transmission-Queues nicht definiert werden. Es existiert nur eine Transmission-Queue pro Queue-Manager und diese wird automatisch erzeugt, wenn der Queue-Manager generiert wird. Es müssen auch keine Channels definiert werden, weder Sender- noch Receiver-Channels; sie werden bei Bedarf automatisch erzeugt.

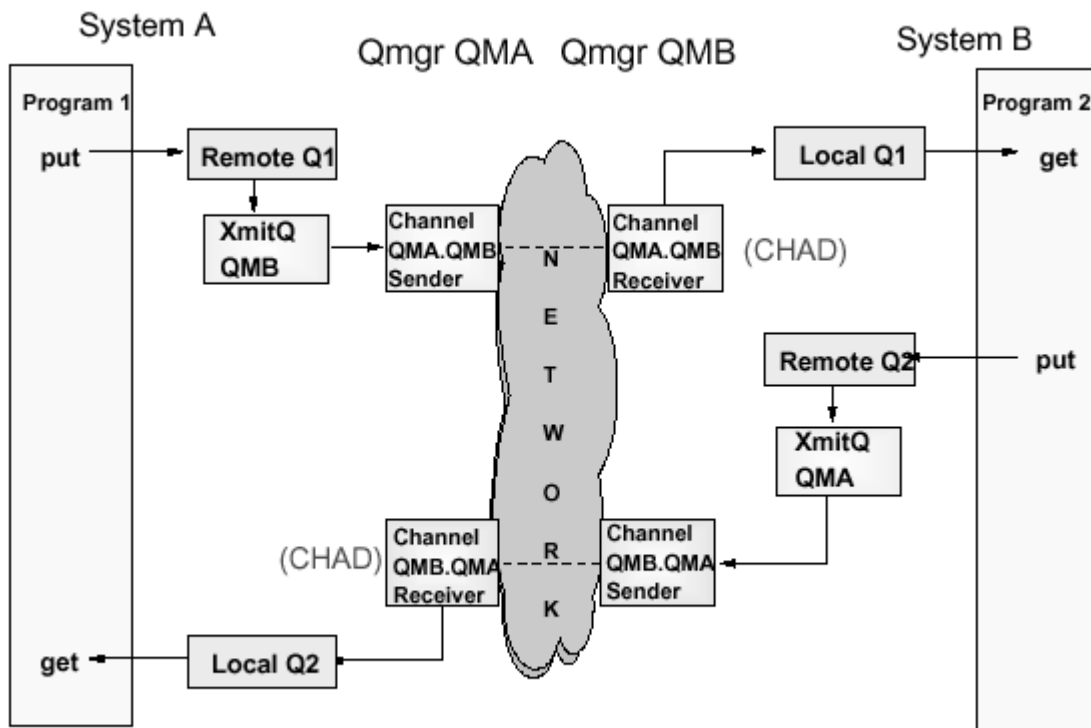


Figure 14. Communication between Two Queue Managers

System A (QMA)	System B (QMB)
DEFINE QREMOTE(Q1) + RNAME(Q1) RQMNAME(QMB) + XMITQ(QMB)	DEFINE QLOCAL(Q1)
DEFINE QLOCAL(QMB) + USAGE(xmitq)	
DEFINE CHANNEL(QMA.QMB) + CHLTYPE(sdr) + XMITQ(QMB) + TRPTYPE(tcp) + CONNNAME(9.24.104.123)	DEFINE CHANNEL(QMA.QMB) + CHLTYPE(rcvr) + TRPTYPE(tcp)
DEFINE QLOCAL(Q2)	DEFINE QREMOTE(Q2) + RNAME(Q2) RQMNAME(QMA) + XMITQ(QMA)
	DEFINE QLOCAL(QMA) + USAGE(xmitq)
DEFINE CHANNEL(QMB.QMA) + CHLTYPE(rcvr) + TRPTYPE(tcp)	DEFINE CHANNEL(QMB.QMA) + CHLTYPE(sdr) + XMITQ(QMA) + TRPTYPE(tcp) CONNNAME(ABC1)

Figure 15. MQSeries Objects Defining Connection between Two Queue Managers

7.8.3 Manueller Kommunikations-Start

Primär müssen die Objekte dem Queue-Manager bekannt gegeben werden. Es wird RUNMQSC benutzt, um die Objekte zu generieren. Dabei muss sicher gestellt sein, dass sich der Queue-Manager im Zustand "Run" befindet. Im nächsten Schritt starten die Listener und die Channels. Der Anwender muss nur den Sender-Channel in jedem System starten. MQSeries startet den Receiver-Channel. Die Kommandos zum Start von Listener und Channel für den Queue-Manager QMA sind:

```
strmqm QMA  
  
start runmqlsr -t tcp -m QMA -p 1414  
  
runmqsc  
  
start channel (QMA.QMB)  
  
end
```

Mit dem ersten Kommando erfolgt der Start des Queue-Managers QMA. Das nächste Kommando startet den Listener. Es hört im Interesse von QMA den Port 1414 ab. Als Übertragungsprotokoll dient TCP/IP. Das dritte Kommando startet runmqsc im interaktiven Modus. Der Channel QMA.QMB wird durch runmqsc gestartet. Für den anderen Queue-Manager gelten äquivalente Kommandos. Die Applikationen müssen in beiden Systemen gestartet werden.

7.8.4 Automatischer Kommunikations-Start

Mit Hilfe des Channel-Initiators können die Channels gestartet werden. Anstatt der obigen Kommandos im Fall des manuellen Kommunikations-Starts gibt man folgende Anweisungen ein (Windows NT, Unix, OS/2):

```
start runmqlsr -t tcp -m QMA -p 1414  
  
start runmqchi
```

Das erste Kommando startet den Listener und mit dem zweiten Kommando erfolgt der Start des Channel-Initiator-Programms. Der Channel-Initiator überwacht eine Channel-Initiation-Queue und startet den geeigneten Channel, um die Message einzulesen. Die Default-Initiation-Queue bildet SYSTEM.CHANNEL.INITQ. Eine andere Möglichkeit, den Channel-Initiator zu starten, bietet das Kommando RUNMQSC (Windows NT, Unix, OS/2). Folgende Kommandos sind anzuwenden:

```
start chinit  
  
oder  
  
start chinit initq (SYSTEM.CHANNEL.INITQ)
```

Um die Transmission-Queue zu triggern, werden drei Parameter hinzugefügt (unten fett-geduckt):

```
DEFINE QLOCAL ( A.TO.B ) REPLACE +  
    USAGE ( xmitq ) +  
    TRIGGER  
    TRIGTYPE ( every ) +  
    INITQ ( SYSTEM.CHANNEL.INITQ ) +  
    DESCR ( 'Xmit QUEUE' )
```

Der Queue-Manager kann den Prozeß, der das Channel-Programm startet, unterschiedlich triggern, d.h. dafür gibt es drei Möglichkeiten:

- Wenn die erste Message auf die Transmission-Queue gelegt wird.
- Jedesmal, wenn eine Message auf die Xmit-Queue gelegt wird.
- Wenn die Queue eine spezifische Anzahl von Messages enthält.

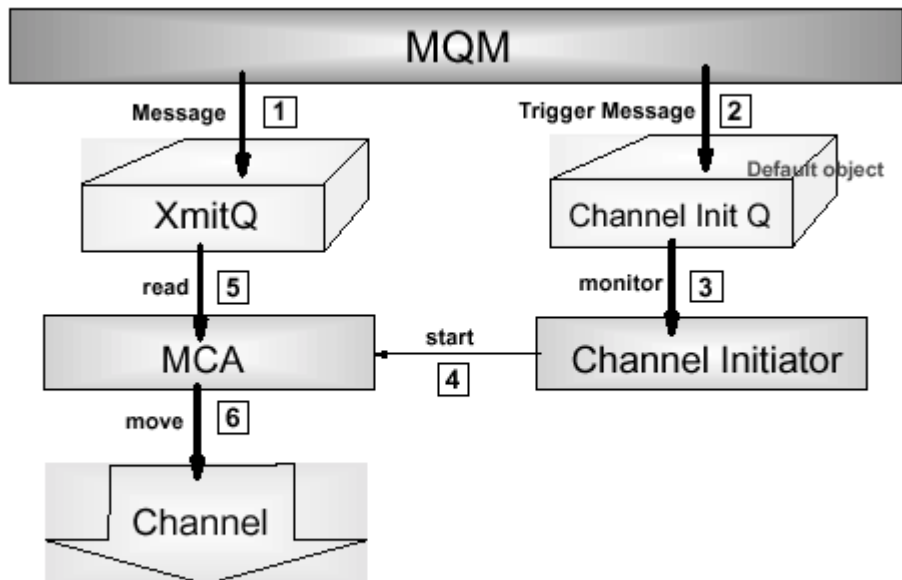


Figure 16. Triggering Channels

Die Abbildung 16 zeigt den Trigger-Ablauf:

1. Das Programm gibt ein MQPUT an eine Remote-Queue aus und stellt eine Message in die Transmission-Queue.
2. Wenn der Queue-Manager eine Message auf die Transmission-Queue legt, überprüft er den Trigger-Typ, der in der Queue-Definition spezifiziert ist. Abhängig von dieser Definition und davon, wieviele Messages in der Queue sind, kann er eine zusätzliche Message in der Channel-Initiation-Queue ablegen. Diese "Trigger-Message" ist für den Nutzer transparent.
3. Da der Channel-Initiator früher gestartet wurde, z.B. zur Boot-Zeit, überwacht er die Channel-Initiation-Queue und entfernt die Trigger-Message.
4. Der Channel-Initiator startet den Message-Channel-Agent (Mover).
5. Das Channel-Programm erhält die Message aus der Transmission-Queue und ruft eine Channel-Exit-Routine auf, wenn diese spezifiziert ist.
6. Die Message wird dann über das Netzwerk zu ihrem Ziel transportiert.

7.9 Triggern von Applikationen

Beim Eintreffen auf einer Queue können die Nachrichten (Messages) automatisch eine Anwendung starten. Dabei wird ein Mechanismus benutzt, der als Triggering bezeichnet wird. Die Anwendungen können wenn notwendig auch in Abhängigkeit von dem Verarbeitungszustand der Nachrichten gestoppt werden. Der Trigger-Mechanismus, d.h. wie ein Applikations-Programm, das auf der Server-Maschine läuft, getriggert werden soll, ist in der Abbildung 17 skizziert. Dabei können das Triggering und die getriggerten Anwendungen unter demselben oder unterschiedlichen Queue-Managern laufen.

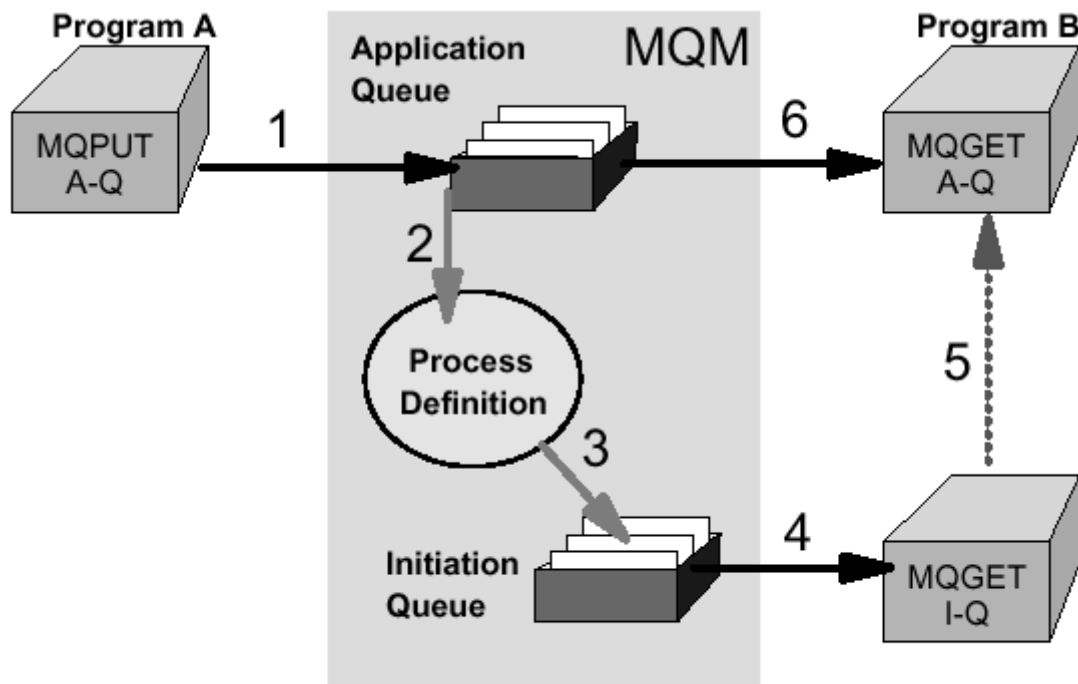


Figure 17. Triggering an Application

MQSeries for Windows V2.1 unterstützt kein Triggering. In der Abbildung 17 sendet das Programm A eine Nachricht an A-Q und soll vom Programm B verarbeitet werden. Der MQSeries-Trigger-Mechanismus funktioniert wie folgt:

1. Programm A gibt ein MQPUT aus und legt eine Message in A-Q für Programm B.
2. Der Queue-Manager verarbeitet diesen API-Call und legt die Message in die Application-Queue.
3. Er findet auch heraus, dass die Queue getriggert wird. Er erzeugt eine Trigger-Message und sieht in der Process-Definition nach, um den Namen der Applikation zu finden und ihn in die Trigger-Message zu schreiben. Die Trigger-Message wird in die Initiation-Queue gestellt.
4. Der Trigger-Monitor bekommt die Trigger-Message von der Initiation-Queue und startet das spezifizierte Programm.
5. Das Applikation-Programm startet und gibt ein MQGET aus, um die Message von der Application-Queue zu finden.

Zum Triggern einer Applikation sind die folgende Definitionen notwendig:

- Die Ziel-Queue muss Trigger-Eigenschaften (fett-gedruckt) besitzen:

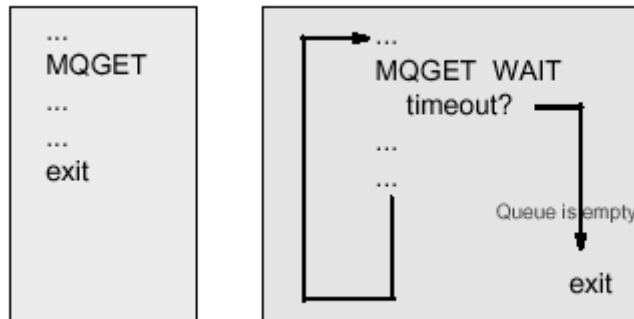
```
DEFINE QLOCAL ( A-Q ) REPLACE +
  TRIGGER
  TRIGTYPE ( first ) +
  INITQ ( SYSTEM . DEFAULT . INITIATION . QUEUE ) +
  PROCESS ( proc1 )
  DESCR ( 'This is a triggered queue' )
```

- Die Process-Definition, die mit der Ziel-Queue verbunden ist, kann wie folgt aussehen:

```
DEFINE PROCESS ( proc1 ) REPLACE +
  DESCR ( 'Process to start server program' ) +
```

APPLTYPE (WINDOWSNT) +
APPLICID (' c: \ test \ myprog . exe ')

Die Wahl des Trigger-Type's hängt davon ab, wie die Applikation geschrieben ist. Es gibt drei verschiedene Typen:



- **EVERY** Jedesmal, wenn eine Message in die Ziel-Queue geschrieben wird, wird auch eine Trigger-Message in der Initiation-Queue abgelegt. Dieser Typ ist zu verwenden, wenn das Programm nach der Verarbeitung einer Message oder Transaktion endet, wie oben links gezeigt ist.
- **FIRST** Eine Trigger-Message wird in die Initiation-Queue nur geschrieben, wenn die Ziel-Queue leer geworden ist. Das wird benutzt, wenn das Programm nur dann endet, wenn es keine weiteren Nachrichten in der Queue gibt, wie oben rechts gezeigt ist.
- **n messages** Eine Trigger-Message wird in der Initiation-Queue abgelegt, wenn sich n Messages in der Target-Queue befinden. Beispiel: Ein Batch-Programm kann gestartet werden, wenn die Queue 1000 Messages beinhaltet.

7.10 Kommunikation zwischen Client und Server

7.10.1 Einführung

Die Client/Server-Verbindung ist in der Abbildung 18 dargestellt. Diese zeigt, dass der MQSeries-Client auf der Client-Maschine installiert ist. Clients und Server werden über MQI-Channels miteinander verbunden. Ein MQI-Channel besteht aus einem Sender/Receiver-Paar, dem sogenannten Client-Connection (CLNTCONN)- und Server-Connection (SVCONN)-Channel.

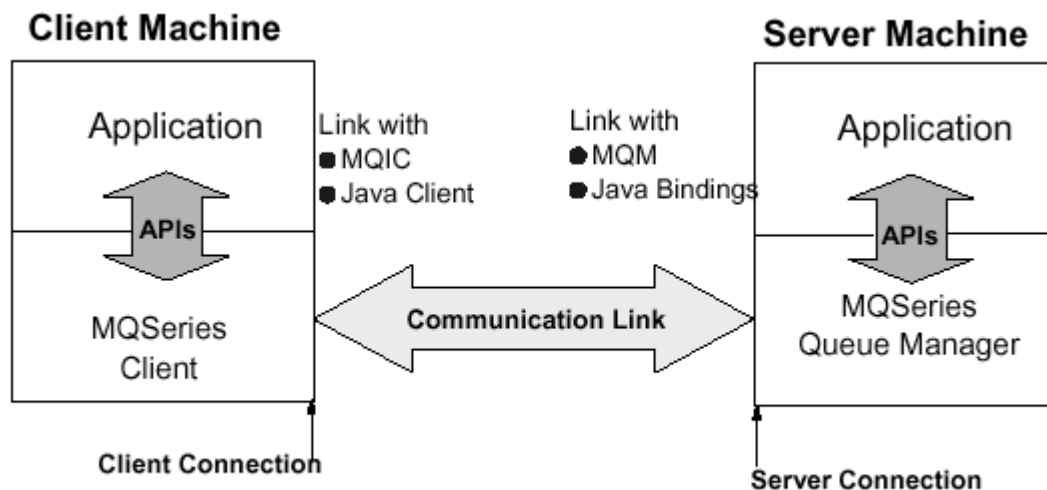


Figure 18. Client/Server Connection

Für die Kommunikation ist es notwendig zu wissen, welches Übertragungs-Protokoll (z.B. TCP/IP) benutzt und welcher Port vom Listener abgehört wird (1414 ist der Default-Port), zusätzlich ist die Adresse des Systems, mit dem die Verbindung hergestellt werden soll, notwendig. Als Adresse kann ein LU-Name, ein Host- bzw. Rechner-Name oder eine TCP/IP-Adresse spezifiziert werden.

Der Client-Connection-Channel wird als Umgebungs-Variable definiert, z.B.

```
set MQSERVER = CHAN1/TCP/9.24.104.206(1414)
```

mit

- MQSERVER ist der Name der Umgebungs-Variable.
- CHAN1 ist der Name des Channels, der für die Kommunikation zwischen Client und Server verwendet wird. Dieser Channel wird im Server definiert. MQSeries erstellt ihn automatisch, sollte er nicht existieren.
- TCP bedeutet, dass TCP/IP für die Verbindung zu dem Rechner mit der folgenden Adresse benutzt werden soll.
- 1414 ist die Default-Port-Nummer für MQSeries. Dieser Parameter kann weggelassen werden, wenn der Listener auf der Server-Seite diesen Default-Wert auch verwendet.

Die Definition auf dem Server sieht aus wie folgt:

```
DEFINE CHANNEL ('CHAN1') CHLTYPE (SVRCONN) REPLACE +
TRPTYPE (TCP) MCAUSER ('')
```

Im MQSeries-Client for Java werden die Umgebungs-Variablen im Applet-Code gesetzt. Ein Applet kann auf irgendeiner Maschine laufen (z.B. auf einem Netzwerk-Rechner), es hat aber keinen Zugriff zu den Umgebungs-Variablen. Das Beispiel unten zeigt, welche Statements im Java-Programm enthalten sein müssen:

```
import com.ibm.mq.*;

MQEnvironment.hostname = "9.24.104.456";
MQEnvironment.channel = "CHAN1";
MQEnvironment.port = 1414;
```


Das Programm kann natürlich viele Messages in der Queue ablegen, bevor sie geschlossen und abgetrennt wird. Das Schließen der Queue und die Trennung vom Queue-Manager kann erfolgen, wenn die Applikation beendet ist, weil es keine weiteren Messages zu verarbeiten gibt.

Der MQSeries-Client-Code, der auf der Client-Maschine läuft, verarbeitet die API-Calls und routet sie zu der in der Umgebungs-Variablen definierten Maschine.

7.10.4 Empfang des Request's durch den Server

In der Server-Maschine werden folgende Queue-Manager-Objekte gebraucht:

- Einen Channel vom Typ Server Connection.
- Eine lokale Queue, hier QS1, in die Clients ihre Messages stellen können.
- Eine Initiation-Queue, in die der Queue-Manager eine Trigger-Message ablegen kann, wenn ein Request die Queue QS1 erreicht. Dabei kann die Default-Initiation-Queue benutzt werden.
- Eine Process-Definition, die den Namen des Programms enthält, wird gestartet, wenn der Trigger-Event (S1) erscheint.
- Eine oder mehrere Queues, in denen das Programm die Antwort-Messages (QA1 und QB1) abspeichert.

In der Server-Maschine müssen zwei Programme gestartet werden: Listener und Trigger-Monitor. Der Listener hört Nachrichten auf dem Channel ab und legt diese auf die Queue QS1. Da QS1 getriggert wird, legt der MQM jedesmal eine Trigger-Message auf der Trigger-Queue ab, wenn eine Message in QS1 abgespeichert wird. In dem Fall, dass eine Nachricht auf der Trigger-Queue platziert wird, startet der Trigger-Monitor das im Process definierte Programm.

Das Server-Programm S1 stellt die Verbindung zum Queue-Manager her, öffnet die Queue S1 und gibt ein MQGET aus, um die Message zu lesen.

7.10.5 Senden einer Server-Antwort

Nach der Verarbeitung eines Request's legt der Server die Antwort in die Antwort-Queue des Clients. Dafür muss er die Output-Queue (QA1 oder QB1) öffnen und ein MQPUT ausgeben.

Da verschiedene Clients dieselbe Server-Applikation benutzen, ist es ratsam, dem Server eine Rückkehr-Adresse zu geben, d.h. die Namen der Queue und des Queue-Managers, der die Antwort-Message empfängt. Diese Felder befinden sich im Header der Request-Message, die der Reply-to-Queue-Manager und die Reply-to-Queue (d.h. QA1 oder QB1) enthält. Das Client-Programm ist dafür verantwortlich, diese Werte zu spezifizieren.

Im Normalfall bleibt das Server-Programm aktiv und wartet zumindest eine bestimmte Zeit auf mehr Nachrichten. Diese Wartezeit kann in der Wait-Option des MQGET API festgelegt werden.

7.10.6 Empfang der Antwort durch den Client

Das Client-Programm kennt den Namen seiner Input-Queue, in diesem Fall QA1 oder QB1. Die Applikation kann zwei Kommunikations-Modi verwenden:

- synchron (konversational)

Wenn die Applikation diesen Kommunikations-Modus mit dem Server-Programm benutzt, wartet sie auf die eintreffende Message, bevor sie die Verarbeitung fortsetzt. Das bedeutet, dass die Antwort-Queue geöffnet und ein MQGET mit Wait-Option ausgegeben wird.

Die Client-Applikation muss in der Lage sein, mit zwei verschiedenen Möglichkeiten umzugehen:

- Die Messages kommt in der Zeit an.
- Der Timer läuft ab, es erscheint aber keine Message.

- Rein asynchron

Wenn dieser Modus verwendet wird, interessiert es den Client nicht, wann die Request-Message ankommt. Normalerweise betätigt der Nutzer einen Knopf im Window-Menue zur Aktivierung eines Programms, das die Antwort-Queue auf Nachrichten überprüft. Wenn eine solche vorhanden ist, kann dieses oder ein anderes Programm die Antwort verarbeiten.

7.11 Das Message-Queuing-Interface (MQI)

Ein Programm redet direkt mit seinem Queue-Manager. Er residiert auf demselben Prozessor oder Domäne (für Clients) wie das Programm selbst. Letzteres benutzt das Message-Queuing-Interface (MQI), das aus einer Menge von API-Calls besteht, die wieder Dienste vom Queue-Manager anfordern.

Wenn die Verbindung zwischen einem Client und seinem Server unterbrochen ist, können keine API-Calls ausgeführt werden, da sich alle Objekte auf dem Server befinden.

Insgesamt existieren 13 APIs; diese sind in der Abbildung 20 aufgelistet. Die wichtigsten davon sind MQPUT und MQGET. Alle anderen werden weniger verwendet.

MQCONN	Connect to a queue manager
MQDISC	Disconnect from a queue manager
MQOPEN	Open a specific queue
MQCLOSE	Close a queue
MQPUT	Put a message on a queue
MQGET	Get a message from a queue
MQPUT1	MQOPEN + MQPUT + MQCLOSE
MQINQ	Inquire properties of an object
MQSET	Set properties of an object
MQCONNX	Standard or fastpath bindings
MQBEGIN	Begin a unit of work (database coordination)
MQCMIT	Commit a unit of work
MQBACK	Back out

Figure 20. MQSeries APIs

Es folgen Kommentare zu den verschiedenen APIs:

MQCONN stellt eine Verbindung mit einem Queue-Manager mit Hilfe von Standard-Links her.

MQCONNX realisiert eine Verbindung mit einem Queue-Manager über schnelle Verbindungswege (Fastpath). Fastpath-PUTs und -GETs sind schneller, die Anwendung muss aber gut ausgetestet werden. Die Applikation und der Queue-Manager laufen in demselben Prozess. Wenn die Anwendung zusammenbricht, fällt auch der Queue-Manager aus. Dieser API-Call ist neu in MQSeries Version 5.

MQBEGIN startet eine Arbeitseinheit, die durch den Queue-Manager koordiniert wird und externe XA-kompatible Ressource-Manager enthalten kann. Dieses API ist mit MQSeries Version 5 eingeführt worden. Es wird für die Koordinierung der Transaktionen, die Queues (MQPUT und MQGET unter Syncpoint-Bedingung) und Datenbank-Updates (SQL-Kommandos) verwendet.

MQPUT1 öffnet eine Queue, legt eine Message darauf ab und schließt die Queue wieder. Dieser API-Call stellt eine Kombination von MQOPEN, MQPUT und MQCLOSE dar.

MQINQ fordert Informationen über den Queue-Manager oder über eines seiner Objekte an, wie z.B. die Anzahl der Nachrichten in einer Queue.

MQSET verändert einige Attribute eines Objekts.

MQCMIT gibt an, dass ein Syncpoint erreicht worden ist. Die als Teil einer Arbeitseinheit abgelegten Messages werden für andere Applikationen verfügbar gemacht. Zurück gekommene Messages werden gelöscht.

MQBACK teilt dem Queue-Manager alle zurück gekommenen PUT's- und GET's-Nachrichten seit dem letzten Syncpoint mit. Die abgelegten Messages als Teil einer Arbeitseinheit werden gelöscht. Zurück gekommene Messages werden wieder auf der Queue abgelegt.

MQDISC schließt die Übergabe einer Arbeitseinheit ein. Das Beenden eines Programms ohne Unterbrechung der Verbindung zum Queue-Manager verursacht ein "Rollback" (MQBACK). MQSeries für AS/400 verwendet nicht MQBEGIN, MQCMIT, MQBACK. Die Commit-Operation-Codes der AS/400 werden dagegen eingesetzt.

7.12 MQSeries-Code-Fragment

Das Code-Fragment in Abbildung 21 zeigt die APIs, die eine Message auf eine Queue legen und die Antwort von einer anderen Queue erhalten. Die Felder CompCode und Reason enthalten fertig-erstellten Code für die APIs. Dieser ist dem Application Programming Reference zu entnehmen.

Kommentare zur Abbildung 21:

- 1 Dieses Statement verbindet die Applikation mit dem Queue-Manager MYQMGR (Name). Wenn der Parameter QMName keinen Namen enthält, dann wird der Default-Queue-Manager benutzt. MQ speichert den Queue-Manager-Zugriff in der Variablen HCon ab. Dieser Zugriff muss in allen anschließenden APIs verwendet werden.
- 2 Um eine Queue zu öffnen, muss der Queue-Name in den für diese Queue geltenden Objekt-Descriptor bewegt werden. Dieses Statement öffnet QUEUE1 nur zum Ausgang (Open-Option MQOO_OUTPUT). Der Zugriff zu der Queue und Werten in dem Objekt-Descriptor wird zurück gegeben. Der Zugriff Hobj1 muss in MQPUT spezifiziert werden.
- 3 MQPUT platziert die übersetzte Message in einem Puffer auf der Queue. Die Parameter für MQPUT sind:
 - Der Zugriff des Queue-Managers (von MQCONN)
 - Der Zugriff der Queue (von MQOPEN)
 - Der Message-Descriptor
 - Eine Struktur, die Optionen für PUT enthält (s. Application Programming Reference)
 - Die Message-Länge
 - Der die Daten enthaltende Puffer
- 4 Dieses Statement schließt die Ausgangs-Queue. Da die Queue vordefiniert wird, erfolgt kein Schließen (MQOC_NONE).
- 5 Dieses Statement öffnet QUEUE2 zur Eingabe nur, wenn die Queue-definierten Standards benutzt werden. Man kann eine Queue auch zum Browsen öffnen, d.h dass die Message nicht entfernt wird.
- 6 Für das Get wird die "Nowait"-Option verwendet. Das MQGET benötigt die Puffer-Länge als Eingangs-Parameter. Da keine Message-ID oder spezifizierte Correlation-ID existiert, wird die erste Message von der Queue gelesen. Dabei kann ein Wait-Intervall (in Millisekunden) angegeben werden. Man kann den Rückkehr-Code prüfen, um herauszufinden, ob die Zeit abgelaufen und keine Nachricht eingetroffen ist.
- 7 Dieses Statement schließt die Eingangs-Queue.
- 8 Die Applikation wird vom Queue-Manager getrennt.

```

MQHCNN  HConn;           // Connection handle
MQHOBJ  HObj1;          // Object handle for queue 1
MQHOBJ  HObj2;          // Object handle for queue 2
MQLONG  CompCode, Reason; // Return codes
MQLONG  options;
MQOD    od1 = {MQOD_DEFAULT}; // Object descriptor for queue 1
MQOD    od2 = {MQOD_DEFAULT}; // Object descriptor for queue 2
MQMD    md = {MQMD_DEFAULT}; // Message descriptor
MQPMO   pmo = {MQPMO_DEFAULT}; // Put message options
MQGMO   gmo = {MQGMO_DEFAULT}; // Get message options
:
// 1 Connect application to a queue manager.
strcpy  (QMName,"MQMGR");
MQCONN  (QMName, &HConn, &CompCode, &Reason);

// 2 Open a queue for output
strcpy  (od1.ObjectName,"QUEUE1");
MQOPEN  (HConn,&od1, MQOO_OUTPUT, &HObj1, &CompCode, &Reason);

// 3 Put a message on the queue
MQPUT   (HConn, HObj1, &md, &pmo, 100, &buffer, &CompCode, &Reason);

// 4 Close the output queue
MQCLOSE (HConn, &HObj1, MQOD_NONE, &CompCode, &Reason);

// 5 Open input queue
options = MQOO_INPUT_AS_Q_DEF;
strcpy  (od2.ObjectName, "QUEUE2");
MQOPEN  (HConn, &od2, options, &HObj2, &CompCode, &Reason);

// 6 Get message
gmo.Options = MQGMO_NO_WAIT;
buflen = sizeof(buffer - 1);
memcpy  (md.MegId, MQMI_NONE, sizeof(md.MegId));
memset  (md.CorrelId, 0x00, sizeof(MQBYTE24));
MQGET   (HConn, HObj2, &md, &gmo, buflen, buffer, 100, &CompCode, &Reason);

// 7 Close the input queue
options = 0;
MQCLOSE (HConn, &HObj2,options, &CompCode, &Reason);

// 8 Disconnect from queue manager
MQDISC  (HConn, &CompCode, &Reason);

```

Figure 21. A Code Fragment

7.13 MQSeries-WWW-Interface-Nutzung

7.13.1 MQSeries-Internet-Gateway

Das MQSeries-Internet-Gateway implementiert eine Brücke zwischen dem synchronen World Wide Web und den asynchronen MQSeries-Anwendungen. Zusammen mit dem Gateway liefern Web Server-Software und MQSeries einen Web-Browser mit Zugriff auf MQSeries-Applikationen. Das bedeutet, dass Industrie-Unternehmen die Vorteile des kostengünstigen Zugriffs zum globalen Markt, der vom Internet bereitgestellt wird, begünstigt durch die robuste Infrastruktur und den zuverlässigen Nachrichtendienst von MQSeries.in Anspruch nehmen können.

Die Nutzer-Wechselwirkung mit dem Gateway erfolgt durch die HTML-POST-Requests. MQSeries-Anwendungen antworten, indem sie HTML-Seiten zu dem Gateway über eine MQSeries-Queue zurückgeben. Das MQSeries-Internet-Gateway kann auf den Systemen MVS, AIX, OS/2, Sun, Solaris, HP-UX oder Windows NT installiert werden.. Das Gateway unterstützt die Interfaces CGI, ICAPI, ISAPI und NSAPI auf diesen Plattformen mit den Ausnahmen: HP-UX unterstützt nicht NSAPI und Sun Solaris, sondern nur CGI.

Die Web-Seite der MQSeries-Produkt-Familie liegt unter

<http://www.software.ibm.com/ts/mqseries/>

7.13.2 Verbindung zu Lotus Notes

MQSeries stellt einen Lotus Notes Add-in-Task-Server bereit. Dieser ermöglicht Lotus Notes-Anwendungen den Zugriff auf MQSeries-Nachrichten. Lotus Notes-Nutzer sind dadurch in der Lage, mit anderen Systemen über MQSeries zu kommunizieren. Lotus Notes ist eine Netzwerk-Applikation, die es Nutzern erlaubt, auf shared-Informationen zuzugreifen. Lotus Notes hat zwei Hauptkomponenten: Server und Client. Der Lotus Notes-Server liefert die Dienste an die Lotus Notes-Clients und an andere Server. Die angebotenen Dienste integrieren die Speicherung und Wiederherstellung von gemeinsamen (shared) Datenbanken und Mail-Routing. Lotus Notes-Clients greifen auf den Server zu, um die gemeinsame Datenbank zu nutzen, d.h. Nachrichten zu lesen und zu verschicken. Der Lotus Notes Add-in-Task-Server erkennt und interpretiert:

- Daten von Dokumenten, die Lotus Notes zu MQSeries senden möchte
- Von MQSeries zurückgesendete Nachrichten für ein Update eines Lotus Notes-Dokuments

Eine Lotus Notes-Anwendung besteht aus einer Datenbank, die speziell konstruierte Dokumente enthält. Letztere verfügen über Formeln (Makros), die durch den Nutzer ausgeführt werden können. Eine Formel kann eine Verbindung zu MQSeries herstellen und Teile eines Dokuments zu einer Mail-in-Datenbank, die mit dem Add-In-Server verbunden ist, übertragen.

Der Lotus Notes Add-in-Task-Server inspiziert die Mail-in-Datenbank und die dort gefundenen Dokumente, um die MQSeries-Messages zu generieren. Die verbundene Datenbank enthält Einträge, die die Beziehung eines Lotus Notes-Dokuments zu einer MQSeries-Message beschreiben, d.h. wie ein Lotus Notes-Dokument in eine MQSeries-Nachricht abgebildet wird. Die Datenbank-Verbindung ist notwendig, um das Mapping zu definieren. Letzteres ist für jeden Dokument-Typ, der mit MQSeries benutzt werden soll, erforderlich.

Für den Zugriff von Lotus Notes auf MQSeries steht Lotus Domino zur Verfügung. Release 2 enthält MQSeries und CICS-Verbindungen für Domino 1.0, das aus MQSeries Enterprise Integrator (MQEI) und der MQSeries-Verbindung LotusScript Extension (MQLSX) besteht. Beide Verbindungen ermöglichen den Zugriff auf MQSeries von LotusScript.