51

MQ

September 2003

In this issue

3 Auto-starting WMQ on Unix

11 Design patterns for message-oriented middleware

23 Migrating to the SSL support in WMQ V5.3

30 Error handling in WMQ Integrator message flows

47 MQ news

update

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.75) each including postage.

Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

MQ Update on-line

Code from *MQ Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mq; you will need to supply a word from the printed issue.

Printed in England.

Auto-starting WMQ on Unix

INTRODUCTION

On Windows systems WebSphere MQ (WMQ) includes components that automatically start the queue manager and associated components during system reboot. On z/OS the operating system can also automatically start the queue manager, which in turn can be configured to automatically start the listener and command server; however, there is no such capability supplied with WMQ for Unix systems.

This article explains how queue manager processes can be started when a Unix machine is booted, and stopped cleanly as the machine shuts down. Information here can be applied to any of the currently supported Unix platforms, including AIX, HP-UX, Solaris, and Linux.

STARTING A UNIX MACHINE

A Unix system can operate in a number of different modes. It might, for example, be set to start in a single-user maintenance mode, or perhaps with a graphical log-in shell on the console, or it might be a fully-functioning multi-user environment. The decision about which mode to make the active one is called the 'run-level'. The system is configured to execute certain scripts depending on the run-level. In maintenance mode most of the system startup will not be needed. Configuring the network, for example, should not be done for that particular run-level and so the network scripts are not made available then.

The work to get queue managers started is primarily all about writing the correct scripts and putting them in the right place so that the system will run them during some but not all run-levels.

The actual numbers used will vary slightly by operating system. On my AIX box the normal run-level is 2, on Solaris it is 3, and

on my Linux machine it is 6. The approach we use here allows us easily to work with a selection of run-levels. The queue managers are probably only needed for one or two of the modes but we can change the list easily.

The */etc/rc.N* directories

Most Unix versions today use the subdirectories named *rc.<N>* for startup scripts, which are in the root filesystem. '*N*' is the run-level, a number between 0 and 9. So on Linux I can see */etc/rc.d/rc.1*, */etc/rc.d/rc.2*, */etc/rc.d/rc.3,* and so on. Linux also has an */etc/rc.d/init.d* directory, which is where the important code is actually stored. I am going to assume the existence of the *init.d* subdirectory in the later scripts so if your operating system does not include one you can create it or modify the scripts to be called from somewhere else.

HP-UX is slightly different in that it uses a root directory of */sbin* instead of */etc/rc.d* but otherwise it is the same as Linux. Solaris has its startup scripts in */etc/rc.<N>* directly, but this too is very similar to Linux.

If you look in */etc/rc.d/rc.2*, you will see files such as:

```
K100dtlogin.rc      K200tps.rc          K799ssh
K900nfs.server      S400nfs.core        S540sendmail
S120swconfig        S420nis.client      S730cron
S200clean_ex        S430nfs.client      S740supprtinfo
```

As part of the system boot the initialization process goes into the appropriate *rc.N* directory selected from the current run-level. It lists all the files in that directory that begin with the letter 'S', sorts them, and then executes the scripts in turn. The number in the filename is the way to determine dependencies of one component on another; starting up a Web server, for example, should only be done after the TCP/IP network configuration has been carried out.

Files that begin with 'S' are executed during startup, while files beginning with 'K' are for killing processes during shutdown (which is often regarded as run-level 0) or restarting at a different run-level.

It may seem that you need to write a lot of similar scripts to startup and shut down a component and then put copies in a directory for each of the run-levels you want to work with, but that is not what happens. Often you can write a single script, which handles both startup and shutdown of your component, and put it in the */etc/rc.d* tree with a meaningful name. Scripts in this directory are not directly executed. However, you then link the scripts into each of the *rc.N* directories with both an 'S' and a 'K' name. The only parameter passed to these scripts is a single word – either 'start' or 'stop' – so that scripts that implement both the 'K' and 'S' versions in a single file can tell which action is required.

People who have grown up with the AIX version of Unix might be wondering about the */etc/inittab* file. This is a different way of achieving the same startup processing, with scripts executed for a selection of run-levels. It used to be the only way on AIX for running user code during startup. While we could use that, AIX has also added the *rc.N* technique and for commonality we will use that. If you look in */etc/inittab* you will see it invoke an */etc/rc.d/rc.N* script for each run-level. AIX does not have many scripts in these subdirectories as the system-supplied startup code is called from its */etc/inittab* processing, but the directories are available for user-written scripts.

When the system is being shut down most Unix systems execute 'K' scripts in a new run-level – normally 0. AIX is a little different in that run-level 0 is considered to be reserved. The only user-modifiable script is */etc/rc.shutdown*.

A SIMPLE BOOTSTRAP SCRIPT

It is best if all WMQ control commands are issued by the mqm user because root is not treated as a special user-ID by WMQ and it might not be in the mqm group. The startup and shutdown scripts are run with root authority so we need to switch user-IDs before running the WMQ commands.

I mentioned earlier that many subsystems will put both the start and stop operations in the same script. We will do that for the module that is directly executed by the operating system but will

separate the ‘real’ code that is run after switching the user-IDs. Putting the working code in its own files will also make it easier to reuse in other situations, such as manually starting queue managers outside the automatic control given here.

This short script should be put in */etc/rc.d/init.d/WMQautostart.sh.*

```
#!/bin/ksh
case "$1" in
start)
   su mqm -c /etc/rc.d/init.d/startallqmgrs
   rc=$?
   ;;
stop)
   su mqm -c /etc/rc.d/init.d/stopallqmgrs
   rc=$?
   ;;
*)
  echo "Usage: $Ø (start | stop)"
  rc=1
  ;;
esac
exit $rc
```

Once it has been created make sure it is executable (use chmod +x) then link it into the directories for execution during startup and shutdown.

```
ln -s /etc/rc.d/init.d/WMQautostart.sh /etc/rc.d/rc.2/S96WMQautostart
ln -s /etc/rc.d/init.d/WMQautostart.sh /etc/rc.d/rc.2/K96WMQautostart
ln -s /etc/rc.d/init.d/WMQautostart.sh /etc/rc.d/rc.Ø/K96WMQautostart
```

Repeat the above lines for each of the run-levels on your operating system where you want the queue managers to be started and stopped. I’ve picked 96 as a sequence number as it is high enough that dependencies such as networking and filesystems will have already been started. You can use any other number provided those earlier components are available. I’ve explicitly put a link into *rc.0* for the shutdown script because most Unixes process that directory.

On AIX you should add the following single line to */etc/rc.shutdown*. If that file does not exist, create it and ensure it is an executable script.

```
su mqm -c /etc/rc.d/init.d/stopallqmgrs
```

THE CONFIGURATION FILE

When designing an automatic startup script you need to determine which items need to be configurable and which can be assumed to take defaults. For the purposes of this article I have decided on a very simple set of options. A file, */var/mqm/autostart.ini*, contains the names of the queue managers that are to be started, the portnumber for a TCP/IP listener, and whether to start the command server.

It should be obvious how to extend the format for additional options. I will always start the default trigger monitor.

The configuration file on this machine has just two lines, one for each of its queue managers:

```
QMGR=apix;LSRPORT=1414;CMDSERV=YES
QMGR=klein;LSRPORT=1415;CMDSERV=NO
```

NETWORK LISTENERS

On Unix systems the network protocols supported by WMQ are TCP/IP and LU6.2, with the vast majority of installations using only the former. There are two ways to configure a TCP/IP listener on these machines: you can either use **inetd** or the **runmqlsr** command. In earlier versions of WMQ I would normally recommend using **inetd**; however, changes to the architecture in V5.3 have made **runmqlsr** a much better choice. It is now more scalable than using **inetd**, allowing many more channels to run simultaneously.

Having made that choice you need to decide whether to have the listener running only when the queue manager is running. My preference is to keep the execution of the listener within the scope of the queue manager – it is started just after the queue manager starts and it is stopped just before the queue manager stops. I think this makes it clearer how the overall system is running; it is also good discipline because you will know which processes to stop when you want to apply maintenance.

You can if you wish run a listener completely independently of the execution of the queue manager so that any connection requests

get a ‘queue manager not available’ return code instead of a TCP/IP error that no-one is listening on the port. The scripts below, however, will follow my preferred configuration.

The LU6.2 listener processing is handled by the SNA products directly. If you are using LU6.2 you do not need to start an explicit MQ listener in these scripts and you do not need the LSRPORT clause in the configuration file.

STARTING QUEUE MANAGERS

This script should be put in */etc/rc.d/init.d/startallqmgrs*. Once it has been created make sure it is executable by the mqm user-ID (use chmod a+rx). However, it does not need to be linked into any of the */etc/rc.d/rc.N* directories.

Note that some of the commands have an ampersand (‘&’) to run them in the background, while others will end quickly once they have started a daemon process.

The **strmqm** command does not run in the background as its job is to start the queue manager daemons. I have seen some scripts in the past which add a ‘sleep’ command after the **strmqm** in order to ‘ensure the queue manager is properly running’. That should not be necessary – by the time the **strmqm** command exits, the queue manager is executing. It might still be going through a recovery phase and, therefore, applications cannot immediately get work done by the queue manager but they should just be blocked in the MQCONN call until recovery has succeeded.

```
#!/bin/ksh
rc=0
config="/var/mqm/autostart.ini"
if [ ! -r $config ]
then
  echo "Config file not found"
  exit 0
fi
qmgrlist=`cat $config | cut -d";" -f1 | cut -d"=" -f2`
for qmgr in $qmgrlist
do
  strmqm $qmgr
```

```
  if [ $? -eq Ø ]
  then
      lsrport=`cat $config |\
             grep "QMGR=$qmgr;LSRPORT=" |\
             cut -d";" -f2 |\
             cut -d"=" -f2 `
      if [ ! -z "$lsrport" ]
      then
     runmqlsr -t tcp -m $qmgr -p $lsrport &
      fi
      cmdserv=`cat $config |\
             grep "QMGR=$qmgr;" |\
             cut -d";" -f3 |\
             cut -d"=" -f2 `
      if [ "$cmdserv" = "YES" ]
      then
     strmqcsv $qmgr
      fi
      runmqtrm -m $qmgr &
  else
    rc=1
  fi
done
exit $rc
```

The bootstrap script is run with root authority. It switches into the mqm user-ID for the WMQ control operations. If you want to run additional commands, such as another trigger monitor, that do not have mqm authority, these should be started from the bootstrap script (with another su call perhaps) and not from the startallqmgrs script.

STOPPING QUEUE MANAGERS

This script should be put in */etc/rc.d/init.d/stopallqmgrs*. Once it has been created make sure it is executable by the mqm user-ID (use chmod a+rx). It does not, however, need to be linked into any of the */etc/rc.d/rc.N* directories.

```
#!/bin/ksh
rc=Ø
config="/var/mqm/autostart.ini"
if [ ! -r $config ]
then
  echo "Config file not found"
  exit Ø
fi
qmgrlist=`cat $config | cut -d";" -f1 | cut -d"=" -f2`
```

```
for qmgr in $qmgrlist
do
  cmdserv=`cat $config |\
         grep "QMGR=$qmgr;" |\
         cut -d";" -f3 |\
         cut -d"=" -f2 `
  if [ "$cmdserv" = "YES" ]
  then
    endmqcsv $qmgr
  fi
  # Do these together in background so we don't
  # delay overall shutdown. If they haven't finished
  # then it won't matter too much.
  ( endmqm -i $qmgr ; endmqlsr -w -m $qmgr) &
done
exit $rc
```

RELATIONSHIP TO THE HIGH AVAILABILITY SUPPORTPACS

There are several HA SupportPacs that you can download for configuring WMQ resources inside products such as Veritas and HA/CMP. These perform a similar job of automatically starting and stopping queue managers although the scripts provided do not include any consideration for processes other than the queue manager itself; you have to edit the scripts for listeners, command servers, etc.

If you are using the HA SupportPacs you should not use the scripts from this article. Allow the HA product to control the queue manager; it will know whether a stand-by machine needs to invoke a startup script and whether disk partitions are online. Mixing *the rc.N* scripts with an HA script is likely to lead to confusion over the state of the queue manager.

SUMMARY

This article should have given you a good idea of how to automate start-up and shut down of WMQ resources on a Unix system. There are many extensions that could be made to them, such as also starting WMQI brokers, but the basic framework will always be the same as that presented here.

Mark E Taylor
IBM Hursley (UK)

Design patterns for message-oriented middleware

INTRODUCTION

Over the years design patterns have become the standard way of defining and implementing a particular software technology in applications. With the pervasive use of object-oriented technology, design patterns have become especially popular among developers. A design pattern can be defined as a standard way of implementing a technology that has become generally accepted; a design 'best practice' if you will.

Most message-oriented middleware applications generally fall into one of a small number of scenarios so this lends itself well to the generalization of the characteristics that make up the scenario, in other words, the creation of design patterns.

Once the developer knows which scenario his application falls into then the specific details of the implementation as well as which features of the middleware to use become more or less automatic based on the design pattern.

In this article I will take a look at five of the most common middleware design patterns. They are:

- Synchronous inquiry.
- Asynchronous inquiry.
- Synchronous update.
- Asynchronous update.
- Fire and forget.

These five design patterns will cover the majority of the message-oriented middleware application processing scenarios.

It should be noted that, although these design patterns can be applied to any message-oriented middleware technology,

whenever I need to give specific details about an implementation I will be referring to WebSphere MQSeries (WMQ).

PATTERN: SYNCHRONOUS INQUIRY

What it does

This pattern is generally used when the end users of an application need to retrieve and display data from some server application, eg customer service representatives using an application running on a Windows workstation that accesses customer information housed in a mainframe database system. Generally the client cannot continue until the information is available. The client application is said to be 'blocked' during this synchronous request. Figure 1 illustrates the message flow of the synchronous inquiry application.

Pattern highlights

There are two main characteristics of the synchronous inquiry pattern. First and foremost the pattern is synchronous in nature. This means that the client application is effectively blocked until the request is resolved either by successfully returning the requested data or by some error condition. Second, this pattern

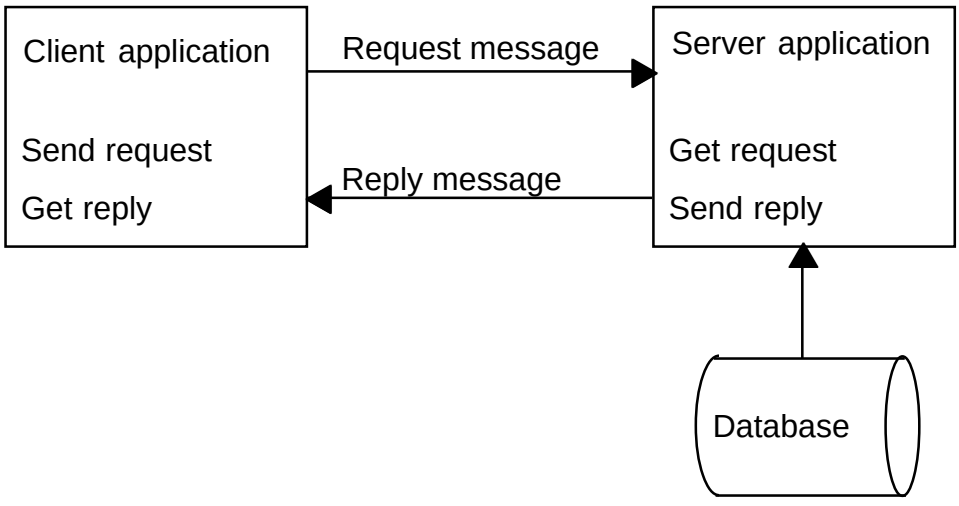


Figure 1: Message flow of a synchronous inquiry application

is simply an inquiry. Data on the server is not updated. If the operation fails the user can simply retry until the desired response is received. Because inquiry operations can be retried if they fail, error-handling logic can be kept to a minimum. There is no need to employ complex recovery logic or any kind of compensating transaction.

Implementation

Two important considerations when implementing any messaging design pattern are the message and queue attributes that will be used. We have already said that since the synchronous inquiry does not change any data we do not need messages to be recoverable. For WMQ this means that we can use non-persistent messages. This will benefit us in terms of performance because non-persistent messages do not need to be hardened to disk as do their persistent counterparts.

With respect to the types of queue needed we can say that request messages flowing into the server should be routed to a predefined permanent queue and reply messages flowing back to the client can be sent to a temporary dynamic queue. WMQ creates temporary dynamic queues automatically and they are deleted when the client application ends.

Another important factor to consider is the length of time that the client application should wait for a response before considering the request to have timed-out. As we have already mentioned, during the request the client is blocked and cannot proceed so waiting for a reply message indefinitely is out of the question. The specific time value is very application-specific but values in the range of 10-30 seconds are quite common.

We also need to remember that, although the client application may have considered a request to be timed-out and gone on to other work, the original request message could still very well be in the system and may get processed at some later time. The client application needs to account for these delayed responses. The most common way to handle this is to use the Message-ID/ Correlation-ID feature of WMQ. In that way the client application

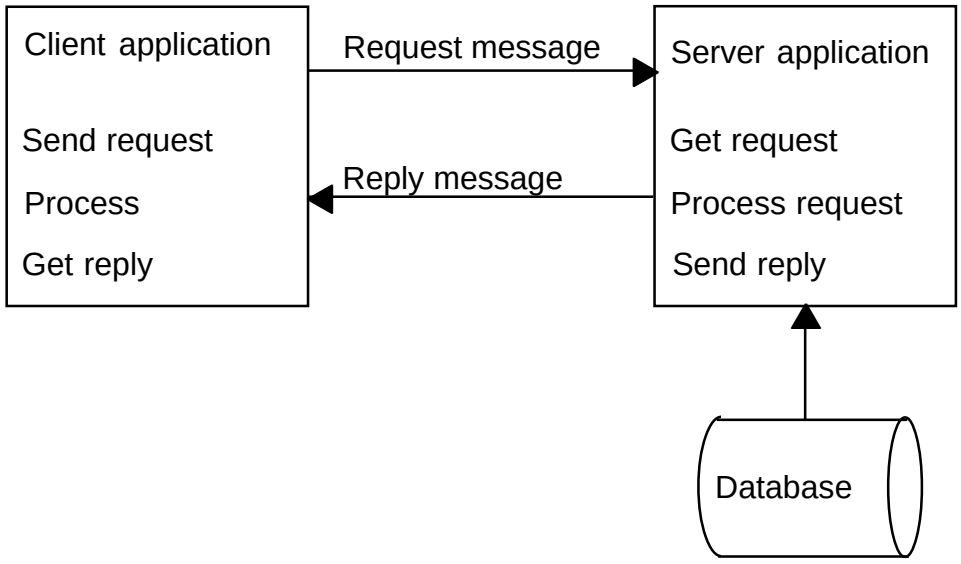


Figure 2: Message flow of an asynchronous inquiry application

can simply discard responses for which there is no current request.

When sending results back to the client the server application can use the reply-to queue and queue manager information that is carried in the WMQ message header.

PATTERN: ASYNCHRONOUS INQUIRY

What it does

On the surface the asynchronous request looks very similar to the synchronous request (see Figure 2). From a message queueing standpoint many of the design decisions are the same. The major difference, however, is that with the asynchronous inquiry the client application does not wait for replies to the requests it has made before continuing to process other work. In order to use this pattern effectively there must be some useful work that the client application can do even though it has not yet received a reply to the request. Because the client application does not wait for replies our design must consider other ways to process the replies when they do arrive.

Pattern highlights

Like the synchronous inquiry this pattern also has two main characteristics. First, the client is asynchronous, or non-blocked. When a request is made the client continues with other work without waiting for a reply.

Second, the pattern is an inquiry. This again means that data on the server is not changed. There are no updates and the operation can be repeated as many times as needed should it fail.

A major design difference between the synchronous inquiry and the asynchronous inquiry is that with the asynchronous inquiry we do not discard responses that are delayed.

Implementation

The implementation of this pattern is quite similar to that of the synchronous inquiry pattern. Since the messages are non-recoverable we can again use WMQ non-persistent messages. Similar queues can be used as well. For request messages flowing into the server a predefined permanent queue can be used and for reply messages flowing back into the client a temporary dynamic queue can be used.

As I mentioned earlier, since the client application does not wait for a single reply for each request our design must consider some other way of processing replies as they arrive. The most common way of handling this is to use a multi-threaded client application where one thread is dedicated to reading and processing reply messages as they arrive.

As in the synchronous inquiry pattern the server application can use the reply-to queue and queue manager information carried in the WMQ message header in order to know where to route reply messages.

PATTERN: SYNCHRONOUS UPDATE

What it does

Of all the design patterns that we will look at in this article the

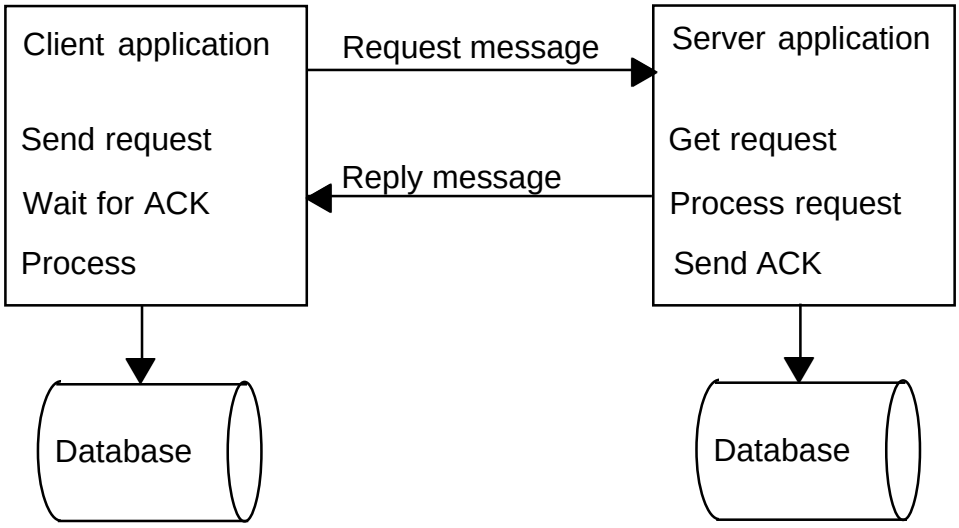


Figure 3: Message flow of a synchronous update application

synchronous update pattern is perhaps the most tightly coupled. In many ways it is also the least desirable of all of the patterns and we shall see why later.

Figure 3 shows the message flow of this pattern. As with the synchronous inquiry pattern the client application cannot continue until a reply has been received from the server. As a result we can encapsulate the messaging operations into a single routine that sends the request and waits for a reply. On the server side we will need separate routines to read the request and send the acknowledgment back to the client as well as to process the request.

Pattern highlights

Once again this design pattern has two main characteristics. Again, the request is of a synchronous nature. The client sends a request to the server and must wait for some acknowledgement before continuing. The client will require some type of acknowledgment from the server that the request has been successfully processed. Second, this is an update operation. Data will be changed at the server so it is important that the update is not lost.

There is a problem with this pattern: what to do at the client if an acknowledgment is not received from the server. In the case of an inquiry we could just wait a pre-determined amount of time and if we had not received a reply we could consider the request to be timed-out.

We cannot take this approach with this pattern. Often the client will modify some data on the client side based on an acknowledgment of a successful completion of the operation on the server. If the acknowledgment is delayed for any reason the client application has no way of knowing whether or not the operation succeeded on the server. Because the operation is of a synchronous nature there is no opportunity for the server to reply later. If no acknowledgment is received the client application will assume that the operation failed, remember the failure, and then take some action later in order to resolve it.

In order to handle this kind of issue there must be additional logic coded into the application and/or the infrastructure must be made more complex with the introduction of a distributed transaction processing system such as CICS. The transaction-processing monitor will guarantee that all updates on both the client and the server happen within the same unit-of-work. Either they all work and are committed or they all fail and are rolled back. This added complexity and cost (transaction processing monitors are not cheap) makes this design pattern less desirable than another alternative that we will discuss later.

Implementation

While it is recommended that this design pattern be avoided wherever possible, if it must be used it can be implemented in the following way. We can use WMQ persistent messages to guarantee once-and-only-once delivery of both the request message and the acknowledgment. For most types of update that replace data, duplicate updates are not an issue and will not adversely affect the system; however, for those types of update that cannot be duplicated, the ability to have once-and-only-once message delivery is critical.

The downside of using persistent messages is that it causes a performance hit since, in order to guarantee delivery of the message, the message queueing software must harden each message to disk and this will result in extra processing time.

As with the inquiry design patterns, update request messages can be sent to a predefined permanent queue on the server. This time, however, on the client, acknowledgement messages must be sent to a WMQ permanent dynamic queue as opposed to a temporary dynamic queue as with the inquiry patterns. This is because the messages are persistent, and temporary dynamic queues do not have the capability to store persistent messages.

Although the messaging infrastructure affords us a good deal of reliability in terms of transactional integrity it is still possible for application failures to compromise this. For example, the server application might read a request from a queue but then fail before it has completed processing it. From the client's perspective the message was successfully delivered but, even so, now it has been lost without the required updates being performed.

The obvious solution to this is to perform the messaging operation along with the data update in the same transaction, or unit-of-

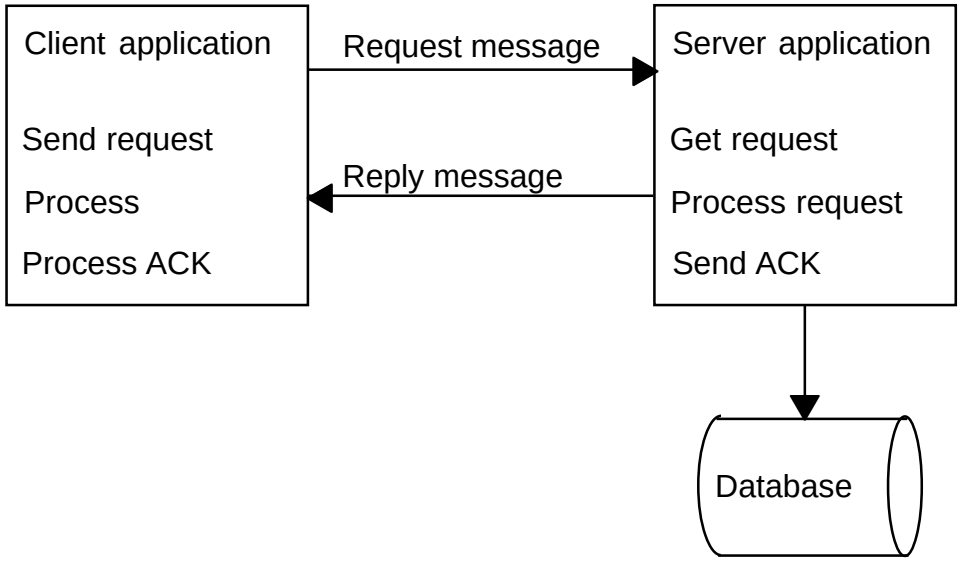


Figure 4: Message flow of an asynchronous update application

work. If the server application fails then data updates are backed out and the original message is placed back on the queue ready to be processed again after the error has been resolved. You can use transactions on almost any WMQ queue and this is specified at the message level.

PATTERN: ASYNCHRONOUS UPDATE

What it does

With this pattern the client application making the request is also interested in the success or failure of the update but does not need it to complete before continuing. Figure 4 shows the message flow associated with the asynchronous update pattern.

Pattern highlights

Once again we have two main characteristics with this pattern. First, it is asynchronous in nature. The client application sends the request and can immediately continue with other work. The client assumes that the request will be processed and still requires some type of positive acknowledgement when the update happens. The second characteristic of this pattern is that it is an update. Data will be updated on the server and it is important that the update is not lost. It may also be undesirable inadvertently to repeat an update.

The topology of this pattern is very similar to that of the asynchronous inquiry pattern that we looked at earlier, the major difference being that we cannot lose the request or the acknowledgment message.

Implementation

This pattern shares many of the same implementation issues as the synchronous update pattern. The update and acknowledgment messages require the same message delivery guarantee so we must use WMQ persistent messages in order to achieve once-and-only-once delivery. With respect to queues

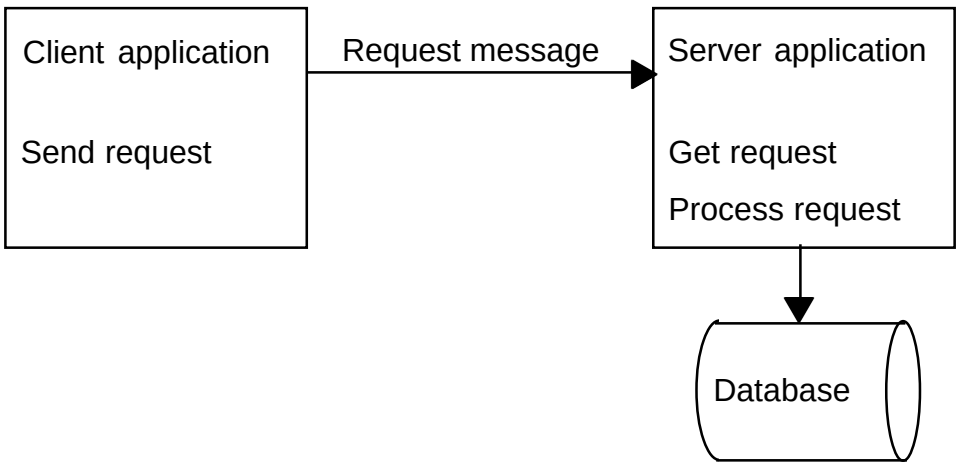


Figure 5: Message flow of a fire-and-forget application

we can once again use a predefined, permanent queue on the server to accept the update messages. For acknowledgment messages flowing back to the client we can use either a permanent dynamic queue (because the messages are persistent) or a predefined permanent queue. This choice is mainly an administrative one.

The same transactional, or unit-of-work, issues apply to this pattern as to the other update style patterns. It is beyond the scope of this article to discuss this in detail; however, at a very high level here is what needs to be considered:

- WMQ syncpoint processing.
- Implementation of distributed transaction processing monitors, eg CICS.
- Transaction demarcation, or what constitutes a logical unit-of-work.
- Additional application code and compensating transactions.

PATTERN: FIRE-AND-FORGET UPDATE

What it does

In the fire-and-forget design pattern the client application requests

some type of update on the server but it does not need to wait for an acknowledgement from the server before carrying on with other work. By using the features of the underlying message queueing system we can be assured that the update will eventually take place. One of the key benefits of this pattern is that, by removing the requirement for an acknowledgement, the workload of the server application is greatly reduced. Figure 5 illustrates the message flow of a fire-and-forget application.

Pattern highlights

With the fire-and-forget pattern we see two main characteristics. First, the client application is asynchronous. The request message is sent and then the client application continues immediately with other work. The client trusts that the underlying message queueing technology will process the update at some point in the future. No acknowledgement from the server is required.

Second, this pattern is also an update. Data on the server will be changed and it is important that these updates are not lost. It may also be important not to inadvertently repeat updates. The once-and-only-once delivery guarantee of WMQ ensures that this will not happen.

Implementation

As with the other update patterns we require some guarantee that our messages are delivered. Once again, with WMQ we shall use persistent messages for the update requests. This assures us of once-and-only-once delivery and even safeguards the messages in the event of a failure of the message queueing subsystem. With this pattern only one queue is needed. Update requests can flow to a predefined permanent queue on the server.

To avoid the possibility of application failures compromising transactional integrity, as discussed previously, we need to perform the messaging operation along with the data update in the same transaction, or unit-of-work.

A COUPLE OF GENERAL GUIDELINES

- I want to clarify that throughout this article I have always associated persistent messages with update operations because of the notion that since it is an update it must be important and, therefore, I cannot afford to lose the message. This is a good rule of thumb but in practice it is not always the case. How many safeguards you build into a system is always a trade-off between the costs to implement them *versus* the importance of the data and what it would cost the organization – in dollars and time – to recreate lost messages.
- Since all the design patterns we have discussed almost always span multiple systems and these systems are often likely to be different operating systems, a good general rule is to use only character data in messages wherever possible. Character data is the easiest to translate between different machine representations and in fact WMQ can perform these translations automatically on behalf of the application.

SUMMARY

In this article we have looked at five of the most common design patterns used by message-oriented middleware applications. We have seen that some patterns share many similarities but that there can be subtle differences as well. By having a standard set of design patterns to work with the developer needs only to decide which one his application falls into. Once that decision has been made many of the subsequent implementation decisions are automatic, being derived from the characteristics of the pattern. This standard 'cookie-cutter'approach is sure to cut development time – always a good thing – as well as providing many other benefits within the organization.

Dale Eckert
Middleware Architect (Canada)

Migrating to the SSL support in WMQ V5.3

INTRODUCTION

One of the new features of WMQ V5.3 is the ability to use SSL to encrypt data on a WMQ channel and to do this both ends of the channel must be running the latest level of WMQ. For existing WMQ users this means upgrading all the WMQ components within your network at the same time, so they all have the ability to use SSL.

This article explains how WMQ Internet Pass-Thru (MQIPT) can be used to simulate one end of the SSL channel, allowing an older version of WMQ Client to connect to a WMQ V5.3 Queue Manager using SSL. There are many configuration features available to MQIPT but this article focuses only on the SSL feature.

WHAT IS MQIPT?

MQIPT is a 'category 3' WMQ SupportPac, which means it is fully supported and can be downloaded free of charge from *http://www.ibm.com/webspheremq/supportpacs*.

It was designed to be installed in the Demilitarized Zone (DMZ) of a firewall and act as a 'proxy' for WMQ traffic but there is no reason why it cannot be installed locally on the same machine as a WMQ client. It will accept a connection request from the WMQ client and route it to the desired destination based on its predefined configuration data.

Once MQIPT has established the connection and the handshaking process has completed, WMQ messages are sent and received as on any other WMQ channel connection.

The only change required to use MQIPT is to the WMQ CONNAME of the channel that's being started. In this example the CONNAME of the CLNTCONN channel must point to the local MQIPT

instead of the target WMQ server. The local MQIPT will be configured to connect to the destination queue manager, using SSL.

SAMPLE SETUP

A typical WMQ client connection is illustrated in Figure 1.

To allow an older version of WMQ client to use an SSL connection MQIPT needs to be inserted into the connection, as shown in Figure 2.

This sample configuration shows how to use MQIPT to simulate the client end of a CLNTCONN channel such that the QM will

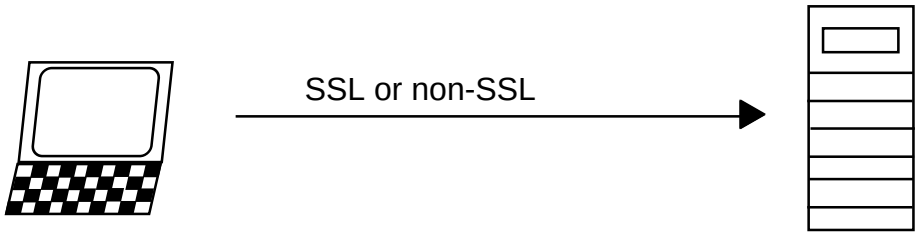


Figure 1: A typical WMQ client connection

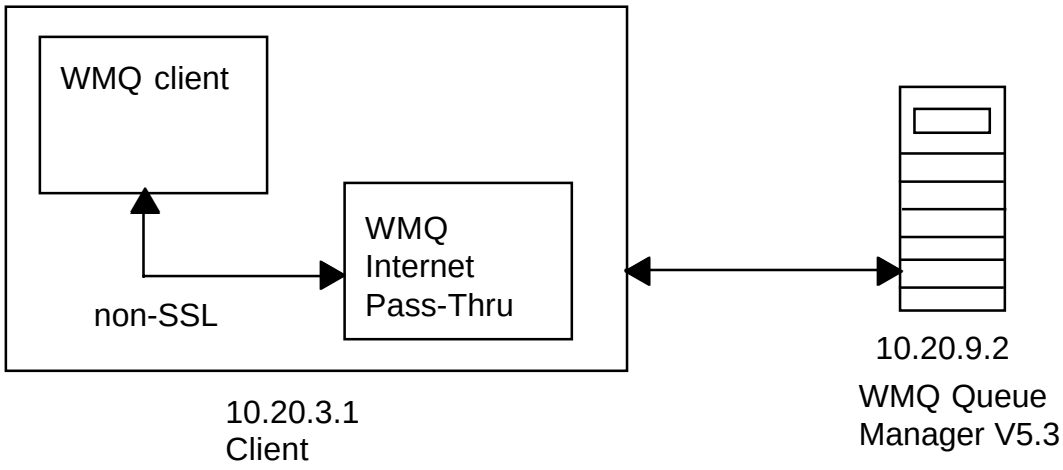


Figure 2: Inserting MQIPT into the connection

think the WMQ client is using an SSL-defined channel.

This setup will use the sample SSL certificates supplied with MQIPT but any valid certificate and its trusted CA certificate(s) could be used instead. The certificates provided with MQIPT are self-signed so a trusted CA certificate is not required. The same self-signed certificate will be used by the QM and MQIPT.

For this example the following assumptions are made:

- You are familiar with defining queue managers, queues, and channels on WMQ.
- You have already installed a WMQ client and server.
- The client and QM are installed on separate machines.
- MQIPT is installed in a directory called *C:\mqipt* on the same machine as the client (note, they have the same IP address).
- You are familiar with putting messages on a queue using the **amqsputc** command.
- You are familiar with getting messages from a queue using the **amqsgetc** command.

CONFIGURING WMQ

On the WMQ server you need to do the following:

- Define a queue manager called MQIPT.QM1.
- Define a server connection channel called MQIPT.CONN.CHANNEL.
- Define a local queue called MQIPT.LOCAL.QUEUE.
- Start a TCP/IP listener for MQIPT.QM1 on port 1414.

You will also need to follow these steps to assign the MQIPT self-signed certificate to the queue manager MQIPT.QM1. These instructions take you through the steps of importing the certificate into Windows Internet Explorer and, from there, importing it into WMQ Explorer and assigning it to MQIPT.QM1. The channel can then be configured to use SSL.

Importing into Windows Internet Explorer

- Start Internet Explorer.
- Select Internet Options from the Tools menu.
- Select the Content tab.
- Press the Certificates button.
- Press the Import button.
- Follow the instructions on the wizard to import the self-signed certificate:
  - located in c*:\mqipt\ssl\sslSample.pfx*
  - enter the password ‘mqiptV1.3’ (without the quotes)
  - leave all options unselected
  - select ‘Automatically select the certificate store based on the type of certificate’
  - press Finish.
- Close the Internet Options dialogue.

Importing into WMQ Explorer

- Open up the properties dialogue for MQIPT.QM1.
- Select the SSL tab.
- Press the Manage SSL certificates button.
- Press the Add button.
- Scroll through the list and select the sample certificate (the owner is Phil Blake).
- Press the Add button.

Assigning the certificate to the queue manager

- Select the sample certificate.

- Press the Assign button.
- Close the QM properties dialogue.

Defining the CipherSpec

- Open the properties dialogue for channel MQIPT.CONN.CHANNEL.
- Select the SSL tab.
- From the standard settings drop down list select RC4_MD5_EXPORT.
- Press the OK button.

CipherSpec	CipherSuite
DES_SHA_EXPORT	SSL_RSA_WITH_DES_CBC_SHA
DES_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA
NULL_MD5	SSL_RSA_WITH_NULL_MD5
NULL_SHA	SSL_RSA_WITH_NULL_SHA
RC2_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
RC4_56_SHA_EXPORT1024	SSL_RSA_EXPORT1024_WITH_RC4_56_SHA
RC4_MD5_US	SSL_RSA_WITH_RC4_128_MD5
RC4_MD5_EXPORT	SSL_RSA_EXPORT_WITH_RC4_40_MD5
RC4_SHA_US	SSL_RSA_WITH_RC4_128_SHA
TRIPLE_DES_SHA_US	SSL_RSA_WITH_3DES_EDE_CBC_SHA

Table 1: Matching CipherSuites to CipherSpecs for use by MQIPT and WMQ

CONFIGURING MQIPT

MQIPT needs a route to be defined and configured to act as an SSL client. The CipherSuite used on the route must match the CipherSpec used by the WMQ channel. Table 1 lists the matching CipherSpecs and CipherSuites. (Various key exchange mechanisms are supported in the SSL protocol that allow for the sharing of secret keys. SSL can make use of a variety of algorithms for encryption and hashing. Many cryptographic algorithms are supported and they are specified by using an SSL CipherSuite or CipherSpec.)

Note the Destination and DestinationPort properties need to reflect your own server and port address.

Define a route to act as an SSL client

- edit *c:\mqipt\mqipt.conf* and add a route definition.
- [route].
- ListenerPort=1415.
- Destination=10.20.9.2.
- DestinationPort=1414.
- SSLClient=true.
- SSLClientKeyRing=c:\\mqipt\\ssl\\sslSample.pfx.
- SSLClientKeyRingPW=c:\\mqipt\\ssl\\sslSample.pwd.
- SSLClientCipherSuite=SSL_RSA_EXPORT_WITH_RC4_40_MD5.

Start MQIPT

- Open a command prompt:
  - c:
  - cd \mqipt\bin

– mqipt ..

- The following messages will be seen on the MQIPT console:

```
5639-L92 (C) Copyright IBM Corp. 2000, 2003 All Rights Reserved
MQCPI001 WMQ internet pass-thru Version 1.3.0 starting
MQCPI004 Reading configuration information from
     C:\mqipt\mqipt.conf
MQCPI011 The path C:\mqipt\logs will be used to store the log
     files
MQCPI006 Route 1415 has started and will forward messages to :
MQCPI034 ....10.20.9.2(1414)
MQCPI036 ....SSL Client side enabled with properties :
MQCPI031 ......cipher suites SSL_RSA_WITH_RC4_128_MD5
MQCPI032 ......keyring file c:\mqipt\ssl\sslSample.pfx
MQCPI047 ......CA keyring file <null>
MQCPI038 ......distinguished name(s) CN=* O=* OU=* L=* ST=* C=*
MQCPI078 Route 1415 ready for connection requests
```

CONFIGURING WMQ CLIENT

The queue manager and MQIPT have both been configured and are ready to accept connections. Note that the IP address defined in MQSERVER needs to reflect your own server address. The WMQ client can be started as follows:

- Open a command prompt:

```
SET MQSERVER=TEST.CONN.CHANNEL/TCP/10.20.3.1
```

- To put a WMQ message on the queue issue the command:

```
amqsputc MQIPT.LOCAL.QUEUE MQIPT.QM1
```

- To get the message issue the command:

```
amqsgetc MQIPT.LOCAL.QUEUE MQIPT.QM1
```

Congratulations! You have now successfully configured an older version MQ client application to communicate with a V5.3 queue manager using SSL communications by making use of MQIPT.

CONCLUSIONS

- Using MQIPT allows each queue manager within an enterprise to be upgraded in a controlled fashion.

- MQIPT can be used at either end of the channel to emulate an SSL connection.
- This scenario can be extended so MQIPT can be used for QM to QM connections as well as client connections.
- To show an SSL connection has been made between MQIPT and the queue manager turn on tracing in MQIPT by setting Trace=5 for the defined route and then run the sample put/get commands. The MQIPT trace file will show the SSL handshaking process before any WMQ data flows.

Phil Blake
IBM Hursley (UK)

Error handling in WMQ Integrator message flows

This article explains error handling in WMQ Integrator (WMQI) message flows and discusses some strategies for making best use of the facility. It applies to all products of the WMQI family.

INTRODUCTION

WMQI is a message broker. Application logic is implemented in WMQI as a message flow. A message flow is a network of interconnected nodes. A node fulfils a specific task and can be:

- An IBM primitive node (these are nodes delivered with WMQI, eg a database node or a compute node).
- A user-defined node.
- A subflow node.

WMQI NODES

To understand the error handling in a message flow we will first have a look at the error behaviour of the three different types of WMQI node.

IBM primitive node

An IBM primitive node usually has one input terminal (in) and two output terminals (out and failure); however:

- Some have more output terminals; for example, the filter node has four (true, false, unknown, and failure).
- Some have fewer output terminals; for example, the trace node has only one (out).
- Some, such as the throw node, have no output terminals.

All nodes that have more than one output terminal have a failure terminal. Messages are received by the input terminal of the node and are processed in the node. If the processing is successful the message is propagated to an output terminal other than the failure terminal. If the processing results in an exception, subsequent processing depends on whether the failure terminal of the node is connected to another node. If it is connected a new entry containing the reason for the exception is added to the exception list of the input message and the message is propagated to the failure terminal of the node. If the node failure terminal is not connected to another node the processing is rolled back to the input terminal and a new entry containing the reason for the exception is added to the exception list of the message.

User-defined node

For a user-defined node it is recommended that you implement the same error behaviour as that of an IBM primitive node with a failure terminal. This means that a user-defined node should have one input terminal (in) and at least two output terminals (out and failure). In case of an error during node processing the original input message is either propagated to the failure terminal or rolled back to the input terminal and a new entry is added to the exception list of the message.

Subflow node

For a subflow node the behaviour in case of an error depends on

the implementation. Behaviour similar to that of an IBM primitive node is difficult to implement. The reason why is shown in Figure 1, which illustrates a simple subflow consisting of three IBM primitive or user-defined nodes, each of which contains one input (in) and two output terminals (out and failure). The subflow itself also holds one input (in) and two output terminals (out and failure).

If, for example, an exception occurs in node 3, the message, including the exception list, is propagated to the failure terminal of the node if the failure terminal of the subflow is connected to another node. But MQ or database operations in the processing of nodes 1 and 2 are not rolled back. The state of the subflow processing is undefined because outside of the node it is not known which node threw the exception and which part of the subflow processing has already been performed and should not be rolled back.

Adding a try-catch node to the beginning of the subflow, as shown in Figure 2, does not really implement the error behaviour of an IBM primitive node with a failure terminal because a disconnected catch path completes the processing of the subflow successfully.

Connecting a user-defined node to the catch terminal of the try-catch node to detect whether the failure terminal of the subflow is connected to another node would be possible (see Figure 3), but in the case of a disconnected failure terminal this node would have to throw an additional exception to rollback the message to

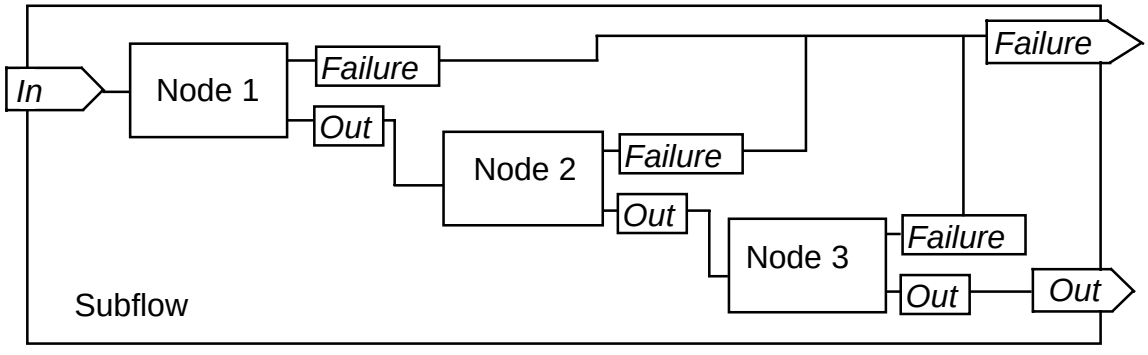


Figure 1: A simple subflow with three IBM primitive nodes

the in terminal of the subflow. This would add an exception to the message exception list for something that is not a true processing exception.

The only simple way to get a consistent state of the processing in case of an exception in a node is to roll back the message to the input terminal of the subflow. An alternative is to implement a compensation method for the MQ and database operations in the subflow but this is usually difficult to carry out.

The following sections describe ways to handle exceptions in a typical message flow. They use the term 'node network' to mean a network of interconnected WMQI nodes.

WMQI MESSAGE FLOW PROCESSING

Figure 4 shows a typical WMQI message flow. It starts with a WMQI MQ input node that has a node network connected to each of its output terminals. The dashed arrow shows the flow of the'normal' processing path (*N*), which first passes the node network A then routes the message to the node network B, which is connected to the'try' terminal of a WMQI try-catch node. The node network paths other than that shown by the dashed arrow are the three main error paths (F1-F3) of a message flow, whereby F1 can also be contained several times in any of the node networks.

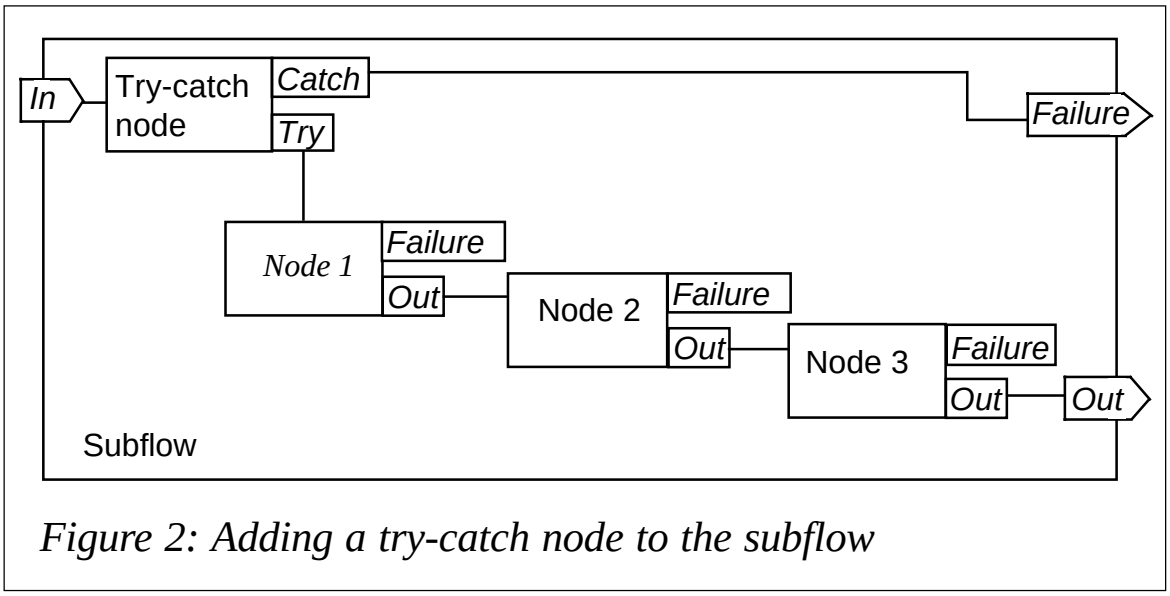


Figure 2: Adding a try-catch node to the subflow

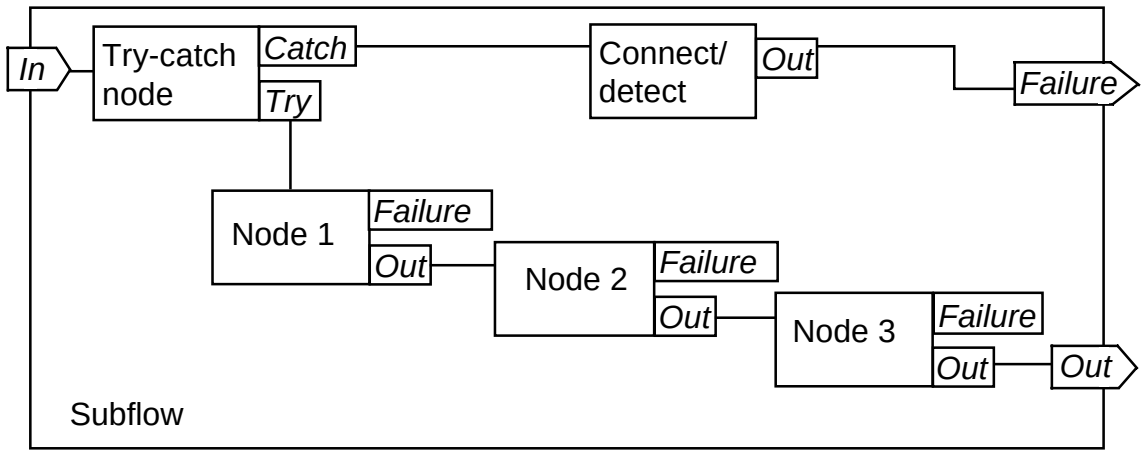


Figure 3: Adding a user-defined connect/detect node

To discuss and understand the processing in the message flow in Figure 4, especially the error processing, we will first take a closer look at the internal processing of the MQ input node, which is shown in Figure 5.

If a message is processed in a transactional context it can be specified by a node property of the MQ input node. The default behaviour for processing messages in a message flow is in a transactional context initiated and completed, either with a commit or a rollback, by the input node. Whether the transaction will be committed or rolled back depends on the result returned from the node networks connected to the output terminals of the input node.

In the following discussion we consider only the processing in a transactional context.

Figure 5 shows the processing behaviour of the MQ input node. The out and catch terminals can be seen as the terminals of an internal try-catch node, which processes the first processing path as transaction T1. If the try-catch path fails, the processing and, therefore, transaction T1 are rolled back and started again until the backout count of the message is equal to or exceeds the backout threshold defined for the input queue. This is indicated by the two arrows in the box for transaction T1, where *x* is the backout threshold of the input queue.

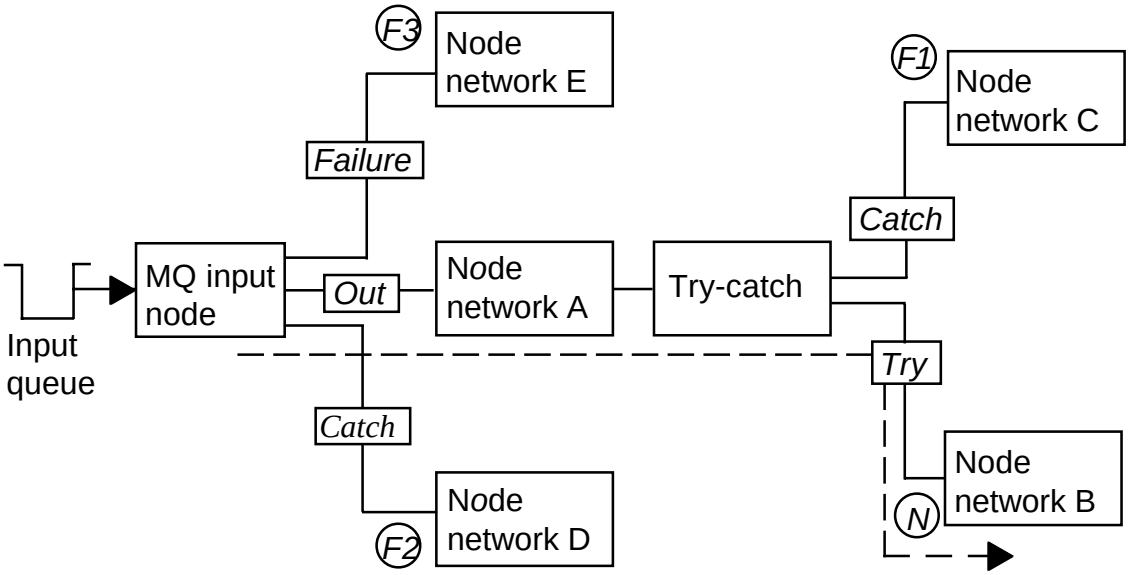


Figure 4: A typical WMQI message flow

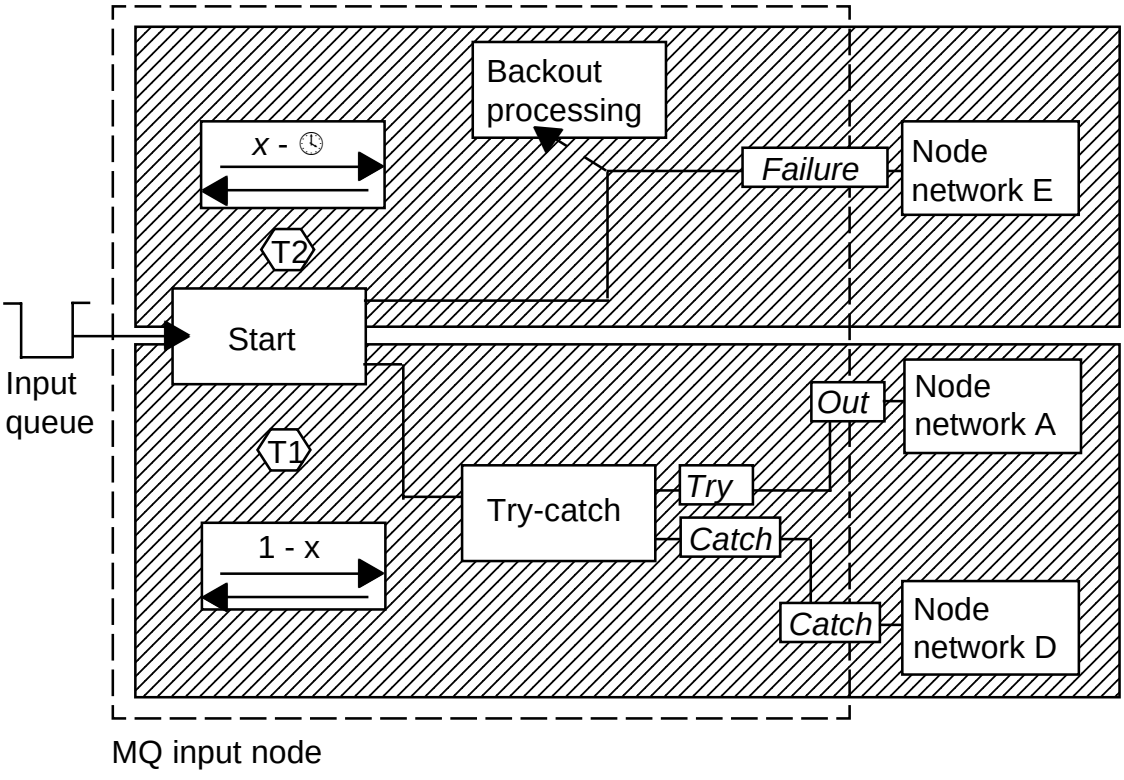


Figure 5: Processing behaviour of the MQ input node

If the backout count is equal to or greater than the backout threshold a new transaction (T2) for the failure path is started. This means that the message propagated to the failure terminal of the MQ input node is the original message from the input queue without an exception list but with the current backout count. If the failure terminal is not connected the backout processing, marked by the dashed arrow, is performed. If the failure path fails, the transaction T2 is rolled back and started again. For T2, unlike T1, there is no limit to the number of restarts; the failure path is tried by the MQ input node until the transaction completes successfully. This is indicated by the two arrows in the box for T2, where *x* is the backout threshold of the input queue.

Figure 6 shows the processing of a message flow. The first transaction (T1) is started by the MQ input node by getting a message from the input queue and propagating it to the output terminal. The message is processed in node network A and routed to node network B, which is connected to the try terminal of the try-catch node.

If the processing in node network B fails with an exception the message is rolled back to the try-catch node and is processed by node network C, which is connected to the catch terminal of the try-catch node. This is the first error path, F1. Modifications in the message performed by node network B are rolled back, but not the database and MQ operations. This should be kept in mind when implementing error path F1.

If the catch path also fails, the processing is rolled back to the MQ input node and propagated to the catch terminal, the second error path, F2. Note again that only the modifications to the message performed in the node networks A, B, and C are rolled back; database and MQSeries operations performed in these node networks are still active until the catch path has finished.

If the error path F2 also fails (see Figure 7), the first transaction T1 is rolled back and a new transaction is started. The roll-back of a message automatically increases the backout count of that message. All database and MQSeries operations performed in transaction T1 are now also rolled back if they were not performed

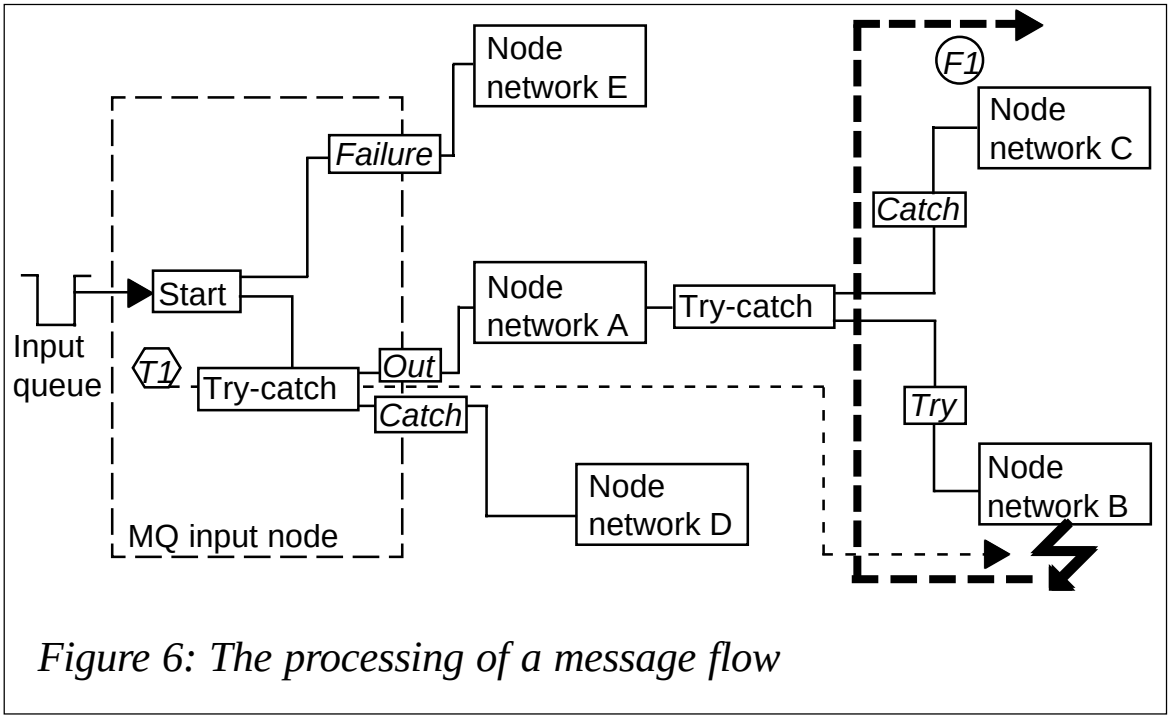


Figure 6: The processing of a message flow

outside transactional control (for example, if 'Transaction No' was specified for the MQ output node).

The message is again read from the input queue and the backout count is compared with the backout threshold defined for the input queue. If the backout count is lower than the backout threshold the started transaction is T1 (see Figure 8), where the

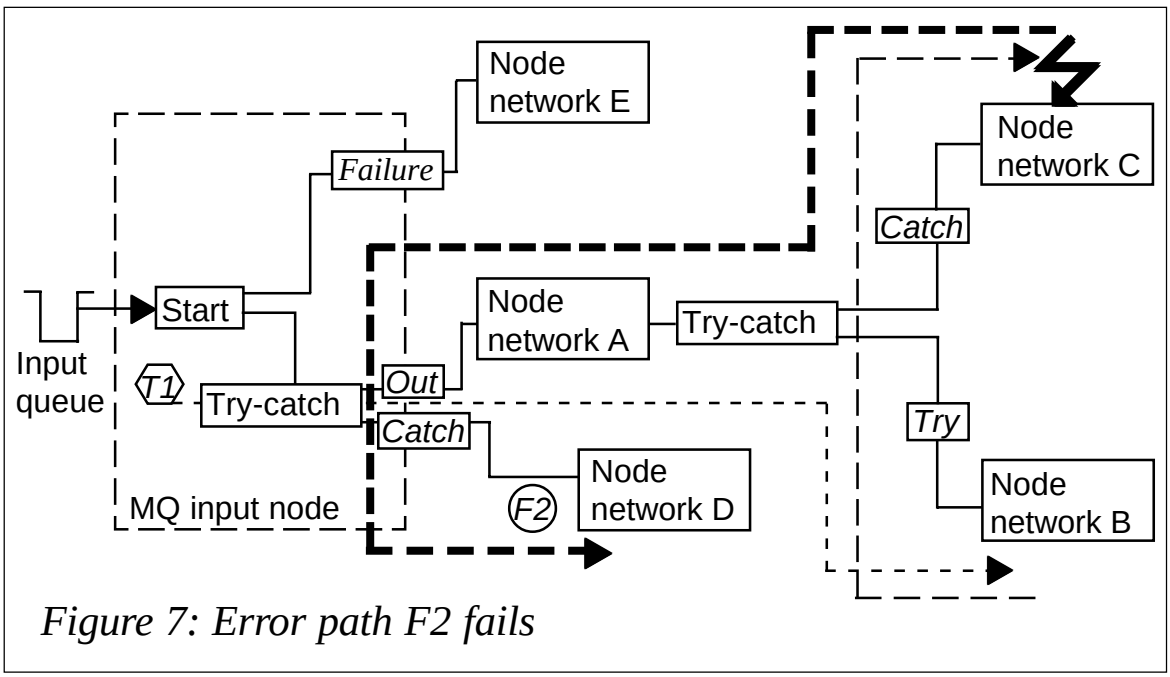


Figure 7: Error path F2 fails

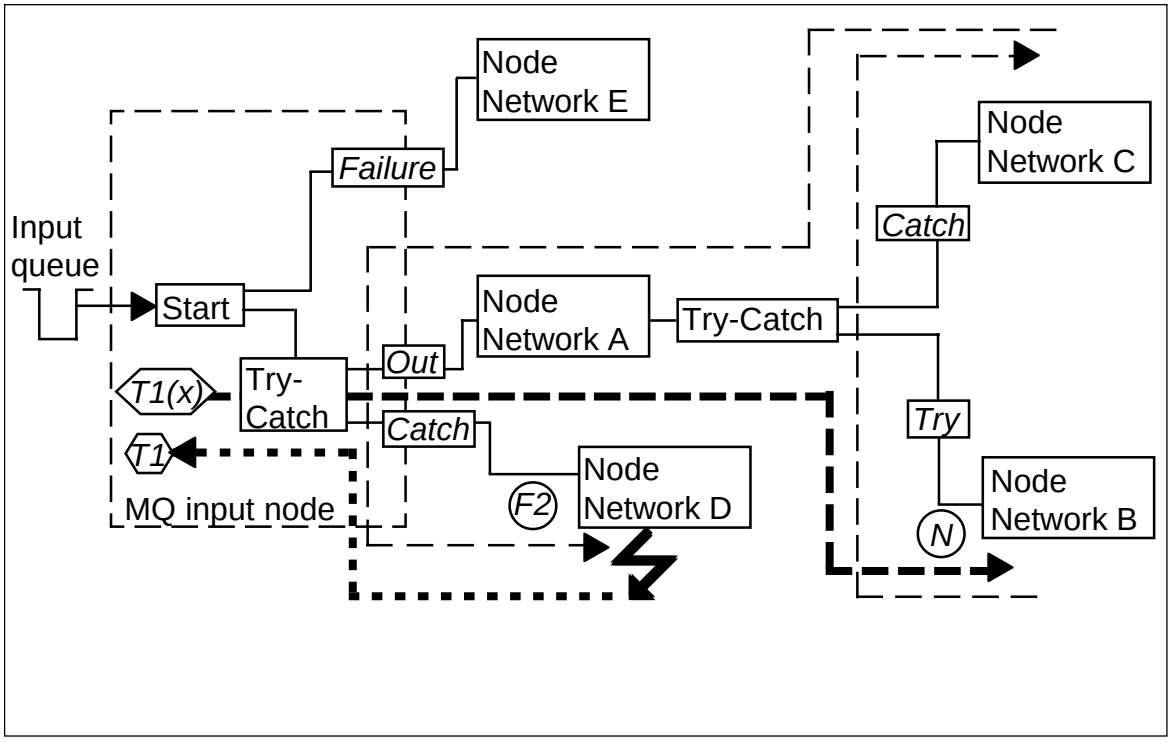
x in the parentheses '()' following T1 represents the current backout count of the message; otherwise the started transaction is T2 for the failure terminal (see Figure 9) or the backout processing.

If the node network D in the catch path finishes successfully, transaction T1 is committed. This implies that all database and MQSeries operations performed in this transaction (node networks A, B, C, and D) are also committed.

Figure 9 shows the case where transaction T1 failed and the backout count of the message read from the input queue within the new started transaction T2 has reached the backout threshold of the input queue. The message is now propagated to the failure terminal of the MQ input node.

If the processing in node network E fails (see Figure 10), transaction T2 is rolled back and started again until the processing in node network E completes successfully and the transaction can be committed.

If the failure terminal is not connected the backout processing is performed. An attempt is then made to put the message into either the backout queue, if one is defined for the input queue, or the default dead-letter queue if one is defined for the queue



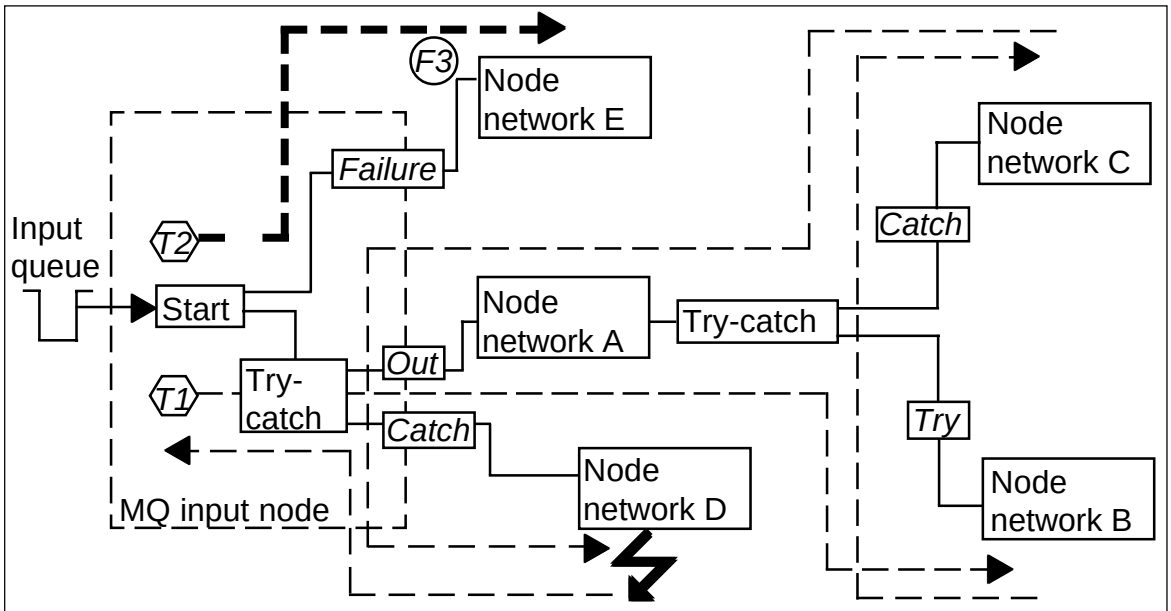


Figure 9: Started transaction is T2 for the failure terminal

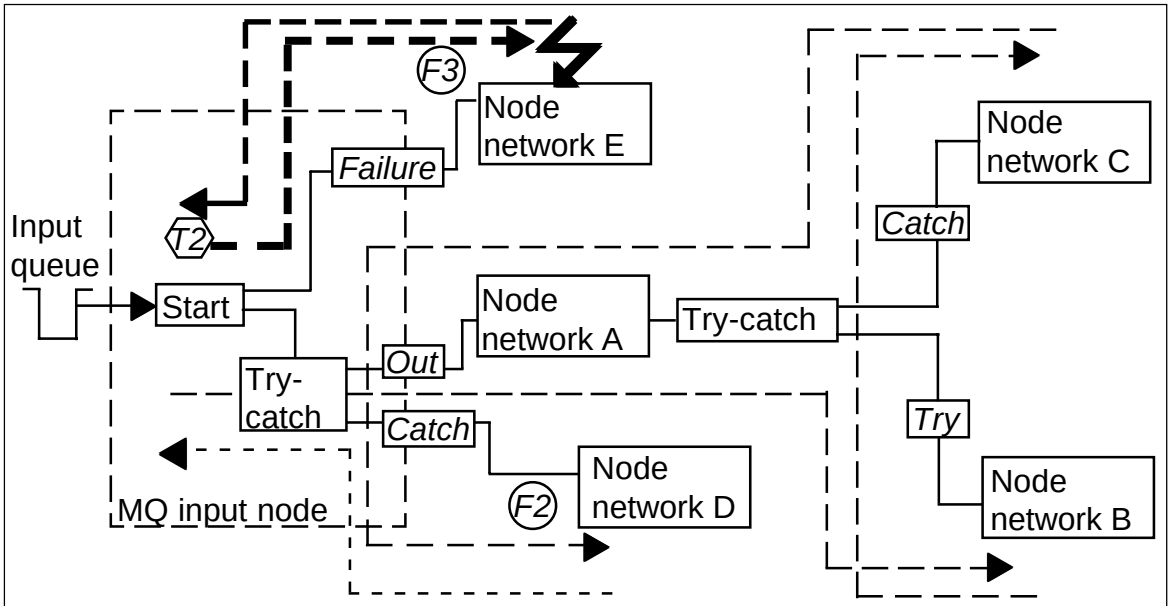


Figure 10: Rolling back T2 until processing completes in node network E

manager. If neither is specified, or if the put to the queues fails, the transaction is rolled back and started again until either the put succeeds or the broker ends.

Summary

If an error occurs in the processing of a message flow with a WMQI MQ input node, one of the following occurs:

- If the failure terminal is not connected, the message is put into the backout queue or the default dead-letter queue.
- If the failure terminal is not connected and if no backout or default dead-letter queue is defined and if the backout count has reached or exceeded the backout threshold, the message remains on the input queue until it is handled by another application.
- In any other case the message flow tries to process the message repeatedly.

If database or WMQ operations are performed in the normal processing path and there is an exception in the processing, transaction T1 should always be rolled back to get a consistent processing state.

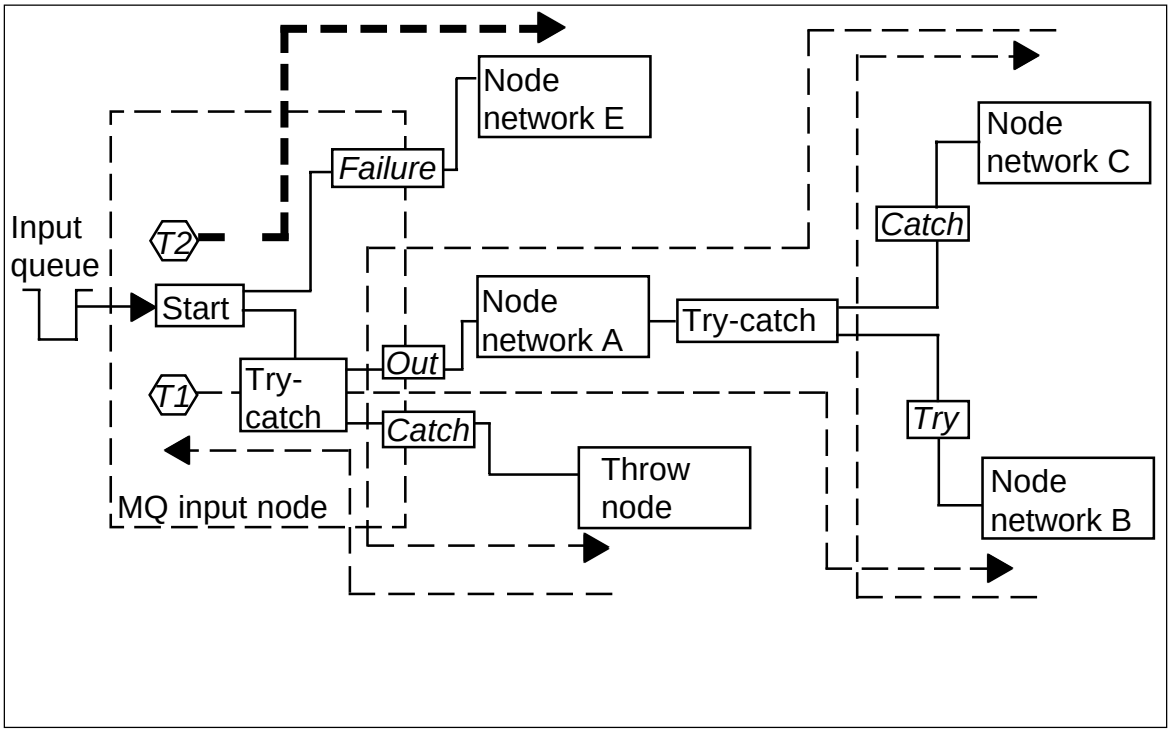
ERROR-HANDLING STRATEGIES

We have discussed the processing and error handling in a WMQI message flow that begins with an MQ input node and we have examined the special behaviour of MQ and database operations within a transactional context in the case of a roll back of the processing. Let's now look at some strategies for dealing with this special behaviour.

Strategy 1: perform error processing in failure path of MQ input node

Figure 11 shows a message flow that makes use of this error-handling strategy. The processing in the subflow connected to the output terminal of the MQ input node failed with an exception and is rolled back to the catch terminal.

To roll back possible MQ and database operations performed in the subflow at the output terminal, the processing and, therefore, transaction T1 are rolled back by a throw node at the catch



terminal of the MQ input node. A new process within a second transaction, T2, is started for the failure terminal, where new database and WMQ operations can be performed in node network E for the error processing.

The disadvantage of this strategy is that the exception list and, therefore, the reason for the error processing is not known to the error processing in the failure path.

Strategy 2: perform error processing in a separate error-handling message flow

This second strategy resolves the disadvantage of the first. Figure 12 shows a message flow that uses this particular error-handling strategy. The exception list is inserted, for example in a folder of the MQRFH2, in the compute node in the catch path of the MQ input node. The message is then put into the input queue of the external error-processing message flow by an MQ output node that is configured to perform its processing outside transactional control. Transaction T1 is then rolled back. This rollback is initiated by a throw node.

The WMQI trace node in the failure path of the MQ input node is

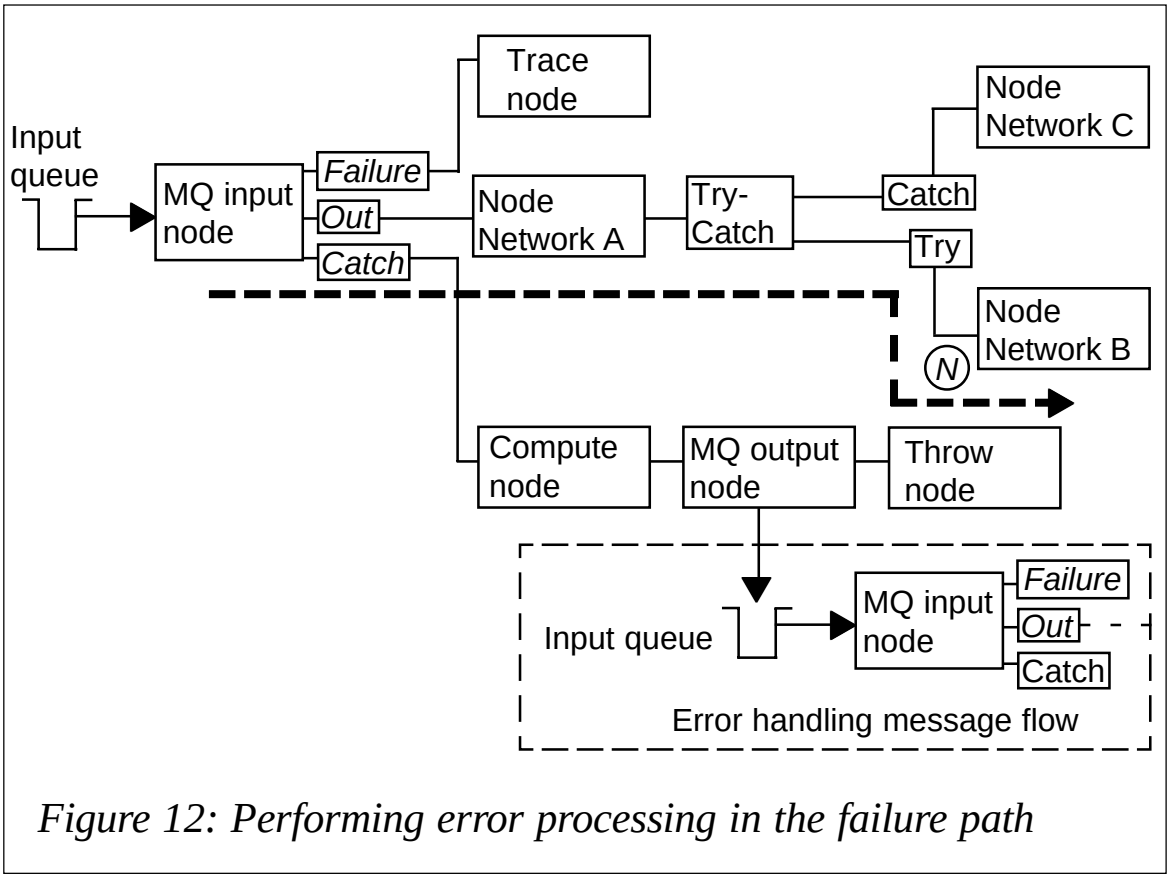


Figure 12: Performing error processing in the failure path

configured with destination 'None' and is used to complete transaction T2, which removes the message from the input queue.

For most scenarios this behaviour would be sufficient. But there are reasons why a message that has failed should not be removed from the input queue and put into an error or backout queue; it would also be preferable to avoid unnecessary use of system resources in the case of repetitive errors with the same message.

For these reasons it might be desirable to stop the message flow from within the message flow. We now discuss several additional error-handling strategies in a WMQI message flow that preserve the message order from the input queue.

Strategy 3: set input queue backout threshold to the maximum value

The simplest strategy is to set the backout threshold value of the

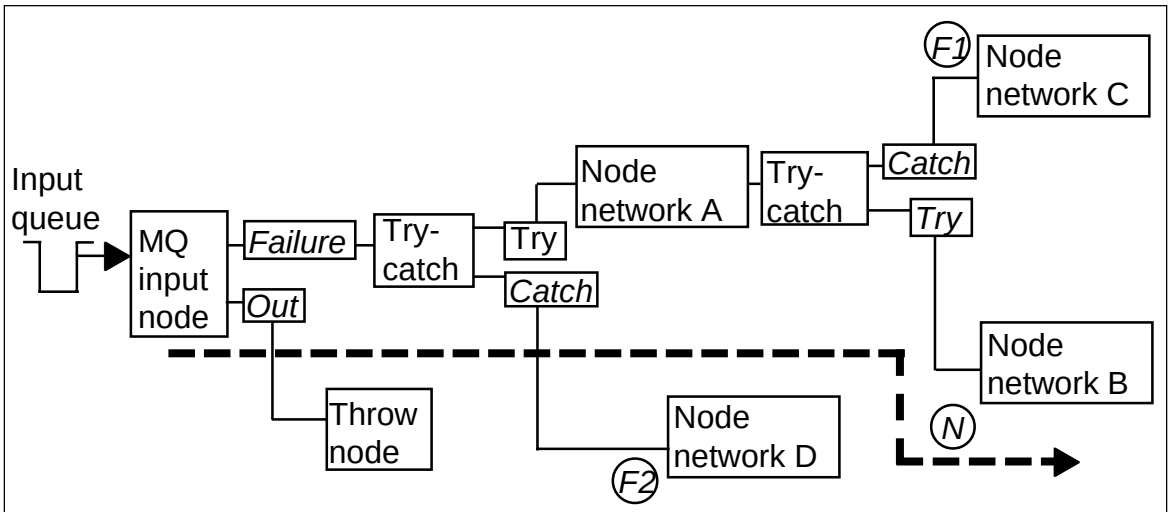


Figure 13: Using the failure path as a main path

input queue to its maximum value. This ensures that the message flow tries to process the message in its normal processing path until the processing completes successfully. The disadvantage of this strategy is that the resource consumption in the case of repetitive errors in the processing is high.

Strategy 4: use the failure path of the MQ input node as a main path

With this strategy the failure path of the MQ input node is used for the normal processing (see Figure 13). This ensures that the message flow tries to process the message even after the backout count has reached or exceeded the backout threshold. The try-catch node connected to the failure terminal provides the try-catch behaviour normally provided by the MQ input node. The disadvantage of this strategy is that the resource consumption in the case of a repetitive error in the processing is high, as in strategy 3.

Strategy 5: user-defined node to switch queue attribute 'Message GET' to inhibit

With this strategy, in the case of an error the user-defined inhibit node disables the Message GET attribute of the input queue. The next time the MQ input node tries to get a message from the inhibited input queue it will get an error and will write a single entry

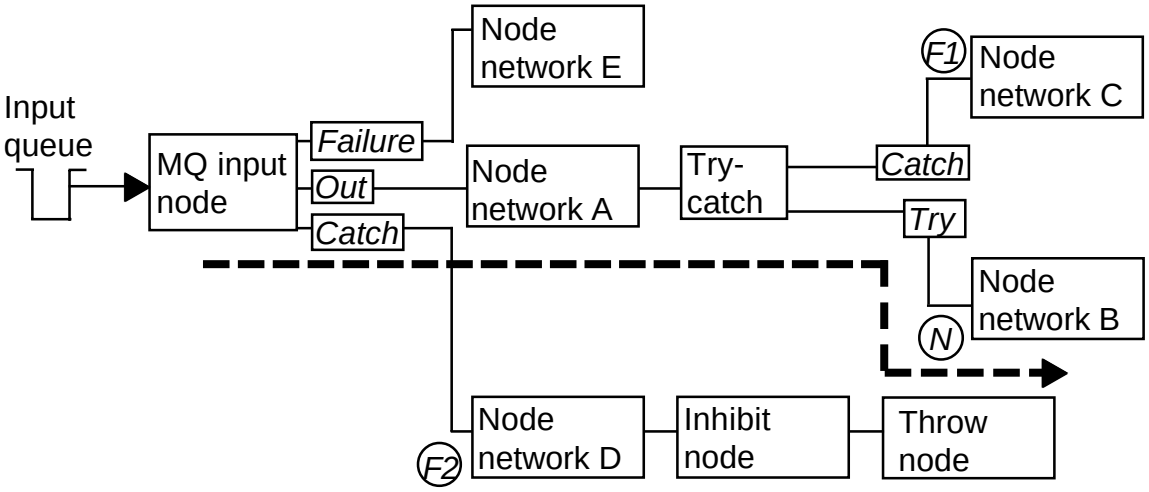


Figure 14: Using a user-defined node to disable Message Get

into the error log. After that the MQ input node tries to get a message from the input queue, without issuing any further error messages to the error log until the Message GET attribute is enabled again.

Figure 14 shows a message flow containing a user-defined node to disable the Message GET attribute of the input queue in the catch path. The disadvantages of this strategy are:

- The broker needs specific access rights to the queue to change the value for the Message GET attribute.
- The queue attribute has to be enabled again by either a WMQ administrator or another message flow containing a user-defined node that enables the Message GET attribute.
- An error entry is written to the system error log each time the Message GET attribute of the queue is set to inhibit.
- The backout threshold of the input queue should be set to a high value to ensure the processing of the failing message when the root problem is removed.

Strategy 6: user-defined input node instead of WMQI MQ input node

With this strategy (see Figure 15) a user-defined controlled input

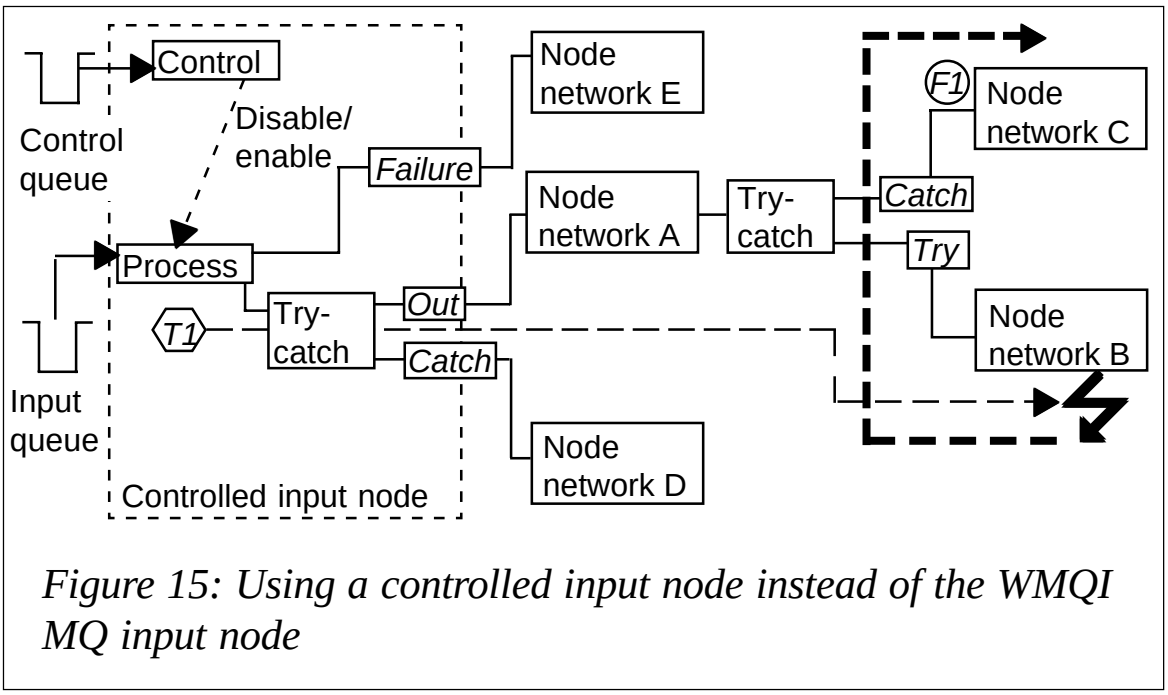


Figure 15: Using a controlled input node instead of the WMQI MQ input node

node is used instead of the WMQI MQ input node. It provides the same functionality as the WMQI MQ input but can be controlled via command messages on a second input queue, the control queue. If the backout count reaches the backout threshold this input node disables its normal processing until a restart command message is received on the control queue. It then enables the processing again and the failing message is retried.

The failure terminal is used only for messages containing syntax errors but the broker also needs specific access rights to the output queues in the message flow to set the security and application context for the message. It is not possible to pass the context of the message to the output queues because the queue handle of the input queue is not known to the WMQI MQ output node. The advantages are:

- There is no resource consumption in the case of a continuous error.
- There is no limitation for retrying the processing of a message.
- The message flow can be disabled and enabled by control messages.

CONCLUSION

In the case of a repeating error in a WMQI message flow there are three final states a message can have:

- The message is moved to a backout or failure queue.
- The message is processed until the transaction completes successfully.
- The message remains on the input queue and is no longer processed by the message flow and must be handled by another application.

In message flows where database and WMQ operations are performed there are several strategies for handling exceptions and preserving a consistent processing state. It is sometimes necessary to process the messages in the order they have arrived on the input queue but in the case of repeating errors it is not desirable to try repeatedly to process the message until the processing completes successfully. For these cases, strategies 5 and 6 are preferable. The other strategies either do not preserve the message order from the input queue (strategies 3 and 4), or result in an unnecessary resource consumption because of repeatedly trying to process the message (strategies 5 and 6).

Christian Herrmann
IBM (Germany)

MQSoftware has announced an updated version of its DataFlow Studio application integration toolset along with a new Q Gateway component that will provide secure file store and forward capabilities via a Web-based user interface.

The DataFlow Studio package enables the transport, transformation, and orchestration of data critical for application integration.

The idea is to provide a cost-effective and secure means for optimizing the exchange of data between a company's secure internal computer server and external users and customers. Users are promised assured delivery of files and centralized administration and management of all enterprise file transfers from one location.

For more information contact:
MQSoftware,1660 South Highway 100, Suite 400, Minneapolis, Minnesota 55416, USA.
Tel: +1 952 345 8720.
Fax: +1 952 345 8721.
Web: http://www.mqsoftware.com

MQSoftware, Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey, GU2 7YG, UK.
Tel: +44 1483 295400.
Fax: +44 1483 573704.

* * *

ATG, a provider of software applications for commerce and customer self-service, and IBM have formed a relationship that enables ATG to OEM IBM's WebSphere Internet infrastructure software as part of its packaged solution offerings. Under the agreement, ATG's products will be enhanced to pre-integrate with WebSphere Application Server (WAS), WebSphere Studio development tools, and WebSphere MQ application integration software.

In a separate statement IBM recently announced the release of WebSphere MQ Extended Security Edition V5.3, which IBM says is optimzied for regulated industries that require a high level of security (such as securities, banking, insurance, and government). The company claims it enables organizations to implement an end-to-end, application-level data protection model and lets administrators perform enterprise-wide, remote management of security policies on queues.

The produce combines WebSphere MQ and Tivoli Access Manager for Business Integration. According to IBM, the offering provides customers with a secure messaging environment that exchanges information across multiple platforms via data messages.

For more information contact your local IBM representative.

* * *