Getting started with MQSeries Publish/Subscribe: A Tutorial

Version 1.0 - 04/03/99

Notices

Introduction

This tutorial guides you from your first steps of creating your first MQSeries Publish/Subscribe broker to writing your own Publish/Subscribe applications.

Following this tutorial will teach you how to :

- Start and end brokers.
- Run the supplied Publish/Subscribe sample application
- Write your first simple publishing application.
- Write your first simple subscribing application.
- Create a small broker network.
- Extend the supplied Publish/Subscribe sample application
- Finally, delete brokers.

Requirements

To complete this tutorial you will require a working knowledge of MQSeries along with an understanding of the programming language C and experience in writing and compiling MQSeries applications. One of the exercises requires MQSeries channels to be defined, though this exercise can be omitted if necessary.

Before you start the tutorial check that you have the necessary level of MQSeries which supports the publish/subscribe function. You will also need to have downloaded the MQSeries Publish/Subscribe SupportPac for your platform. This can be found at **http://www.software.ibm.com/ts/mqseries/txppacs/ma0c.html**, as can the MQSeries Publish/Subscribe User’s manual which will also need to be consulted.

When you perform the tutorial you must be logged on as a user which is authorized to run MQSeries commands (a member of the *mqm* group). A C compiler will also be required.

It is assumed that the accompanying C source files with this document have been downloaded into a suitable place on the machine that you will be using for this tutorial, you will need to be able to modify these files. The directory structure of these files should be maintained, from now on we will refer to these files by there relative directory (e.g. Tutorial/Exer3/amqssub).

All supplied Publish/Subscribe samples used in this tutorial can be found under the standard directories for MQSeries samples for your installation of MQSeries (e.g. on *Windows NT* the executable samples can be found in ***mqmtop****\tools\c\samples\bin*), all non-executable Publish/Subscribe sample files will be found in or below the *pubsub* subdirectory in the C sample directory.

There is no single answer to any of the steps in the incomplete source files but a set of completed source files have been supplied under the *Tutorial\Solution* directory if required.

Exercise 1. Your first broker

First, we will setup a single MQSeries Publish/Subscribe broker.

For this we must create a queue manager, you could use the hostname of your computer as the queue manager name, or any other valid name, from now on we will call this ***QMgrName***. We always recommend that a *dead letter queue* is defined on all queue managers that will be running a broker so we will create this queue manager with one defined. From the command line type the following to create the queue manager.

```
crtmqm -u SYSTEM.DEAD.LETTER.QUEUE QMgrName
```

Start the queue manager

```
strmqm QMgrName
```

Now we can start the broker

```
strmqbrk -m QMgrName
```

To display the status of the broker we can use the *dspmqbrk* command

```
dspmqbrk -m QMgrName
```

You should have seen the following message:

```
MQSeries message broker for queue manager QMgrName running.
```

The broker is now ready to receive commands and publications from MQSeries Publish/Subscribe applications.

The first time we start the broker all the necessary default queues required by the broker are created on the queue manager, we can display these by using runmqsc

```
runmqsc QMgrName
  display qlocal(*)
  end
```

All the *SYSTEM.BROKER.** queues are those used by the broker, queue names starting with *SYSTEM.BROKER* are reserved for use by the broker. See the MQSeries Publish/Subscribe Users Guide for more details on the individual use of these queues.

To end the broker, without ending the queue manager, the *endmqbrk* command is used, end the broker now.

```
endmqbrk -m QMgrName
```

Once this has completed the broker has been ended, this can be verified by running *dspmqbrk* again.

```
dspmqbrk -m QMgrName
```

This time we should see the following message.

```
MQSeries message broker for queue manager QMgrName not active.
```

For the next exercises we will be using the broker we just created, to restart the broker rerun the *strmqbrk* command.

```
strmqbrk -m QMgrName
```

We are now ready to continue with the next exercise.

Exercise 2. Running the Sample

A sample Publish/Subscribe application is supplied with the MQSeries Publish/Subscribe package that demonstrates a number of basic publish/subscribe features. The sample is a simple implementation of a soccer match results gathering service.

Two local queues are required on the broker's queue manager before we can run the sample programs. SAMPLE.BROKER.RESULTS.STREAM, is a new stream queue; the sample does not use the supplied default stream of the broker. Also one of the sample programs is a subscriber to information being published on the sample stream, a *subscriber queue* is required to receive these publications, the queue used by the sample is *RESULTS.SERVICE.SAMPLE.QUEUE*.

To define these queues on your queue manager use the MQSC script which is supplied with the MQSeries Publish/Subscribe package. This can be found in the *pubsub* sample directory (***mqmtop****\tools\c\samples\pubsub* on Windows NT, ***mqmtop****\samp\pubsub* on UNIX). From the command line change to this directory and type the following.

```
runmqsc QMgrName < amqsresa.tst
```

We can now start the sample.

The sample has two parts, a publisher (or publishers) that publish details on a single soccer match and a subscriber that subscribes to these soccer match events published by all matches being played and displays the results for them. The publisher application is *amqsgam* (*amqsgam.exe* for *Windows NT*) and the subscriber application is *amqsres (amqsres.exe* for *Windows NT),* both these executables can be found in the standard MQSeries C sample executable directory.

The subscriber application, the *results server*, has to be running before any soccer matches are started. So, from a command line in the executable sample directory start the results server now

```
amqsres QMgrName
```

The match simulator publishes event publications, so that the results server does not miss any publications the results server has to of registered its subscription before any match simulators are started. Once a message is displayed by the results server, indicating it is okay to start the match simulator (it has successfully registered its subscription) we can start a soccer match. From another command line, think of two soccer teams (no spaces or double quotes (") allowed in either name) and start the match using these names as the first two parameters, if the results server, *amqsres*, has already finished restart it from its command line first. Now, from *another* command line in *the* same sample directory start the match simulator.

amqsgam ***Team1 Team2 QMgrName***

The soccer simulator, *amqsgam*, publishes an event publication to the *SAMPLE.BROKER.RESULTS.STREAM* of the broker's queue manager for every event that occurs in the match, when the match starts, when a goal is scored and finally when a match ends. The results service, *amqsres*, subscribes to all these event publications and so the broker sends each publication on to the results service's subscriber queue. The results service reads these publications as they arrive and processes them accordingly, displaying a new game has started, updating the current score and removing the game when it has finished.

We can now extend the simulation, restart the results server if it has ended and the first match simulator, now think of two more soccer team names and start another instance of *amqsgam*, like before, from another command window.

Now you can see both sets of match event publications are being received by the results server. You could extend the number of concurrent matches being played to as many as you would like. This demonstrates a single subscriber with multiple publishers.

A script file (a batch file for *Windows NT*) has been supplied in the tutorial directory *Tutorial\Exer2* to start up a number of soccer matches at the same time. If you are using UNIX change the permissions of the script file *amqsmany* to allow you execute permission. From a command line, change to this directory and try running this to see how the sample handles many different publishers, the results service should be started before this. Substitute the name of your queue manager for *QMgrName* in the command below. The directory containing the *amqsgam* executable sample will need to be in your path to be able to run *anqsmany*.

amqsmany ***QMgrName***

The other main feature of this sample is that it maintains a current state of all the matches being played, which allows the results server to be restarted after a possible failure. The results server does this by publishing a retained message to the broker with the latest score of each match every time the score changes. On startup of the results server it subscribes to all these retained publications and when they are received the current match state is restored as to how it was the last time the results server was running.

To exercise this feature restart the script *amqsmany* (*anqsmany.bat* for *Windows* NT) that we used above. Once it is running and a couple of goals have been scored you can change to the window running the results server, *amqsres*, and prematurely kill that process (using **Ctrl-C**). The results service can then be restarted at any time and you will see that the matches being played will be restored to their last known score and then updated by any remaining match changes that have occurred whilst the results service was stopped. If, when restarting the results service, the sample fails to open the subscriber queue for reason *2042* (*MQRC_OBJECT_IN_USE*) it is because we have killed the previous instance of the results service which had the subscriber queue opened exclusively and the queue manager is yet to learn this, wait a few seconds and try again, repeat until the results service sample starts correctly.

We will return to the sample later in Exercise 5 to explore in more detail the concepts of MQSeries Publish/Subscribe that have been implemented by it.

Exercise 3. Writing Your Own Samples

To make you more familiar with the way an MQSeries Publish/Subscribe application can be written, using the MQSeries message format *MQRFH*, the following exercises require you to complete the partial samples we have supplied. To be able to do this successfully you will need access to the MQSeries manuals (including the *MQSeries Publish/Subscribe* manual) and have some knowledge of the programming language C.

The samples you will be completing are a lot simpler in structure to the results server sample we used in *Exercise 2*. they consist of a publisher and a subscriber. The publisher will publish character strings, supplied by the user, on a particular topic. And the subscriber will register on a topic and display any publications that it receives.

To run the samples a local queue is required for the subscriber and the publisher, use *runmqsc* to define two local queues for this purpose. Call one *SUBSCRIBER_QUEUE* and the other *PUBLISHER_QUEUE*. As the samples are configured to use the default stream of the broker no extra queue is required for a new stream as in the previous sample.

```
runmqsc QMgrName
  define qlocal(SUBSCRIBER_QUEUE)
  define qlocal(PUBLISHER_QUEUE)
  end
```

The incomplete C source files have been supplied in the tutorial directories, you are required to modify these files and compile them into executables, as with standard MQSeries samples, when linking into an executable MQSeries library(s) must be linked, see the *MQSeries Application Programming Guide manual* for details on your particular platform. Each of the following samples consist of a single source file per executable and should be compiled into an executable of the same name as the directory that the source can be found in.

Publisher Project

The first sample we will complete is the publisher sample *amqspub*. The incomplete source of this sample, *amqspuba.c*, can be found in the tutorial directory *Tutorial\Exer3\amqspub*.

There are eight steps within the sample that are to be completed, work your way through the code, understanding what is being done, when you reach an incomplete step follow the instructions to add the correct lines of code. There is more help in the source file in the comments preceding each step, use this in conjunction with the help given below.

The eight steps to be completed in the code are listed below

Step 1.

To be able to put publications to the broker's stream queue we must open it first, complete the configuration of the MQOPEN command for putting messages.

Step 2.

When we publish to a broker the publication must normally start with an MQRFH structure (the definition can be found in the MQSeries Publish/Subscribe manual). All fields in this structure must have a defined value, a default definition of the MQRFH structure is supplied with MQSeries, use this to initialize the fields of the MQRFH within the message block. Some values of the MQRFH will have to be changed from their default values before we can publish the message, we will change those later, when we know what they will be.

Step 3.

Immediately after the defined MQRFH structure in the message block a NameValueString must follow, this is a character string. Point the pNameValueString pointer to the starting position of the NameValueString in the message block.

Step 4.

The contents of the NameValueString is what is used by the broker to distinguish between different Publish/Subscribe commands and how it is to be processed. The NameValueString of this message must contain all the required information for the broker to recognize it as a publication and how to process it. The Publish/Subscribe manual has information on all the recognized command messages and what name/value pairs are required/optional, use this to help in forming a continuous character string valid for our publications.

Step 5.

The StrucLength field of the MQRFH structure holds the length of the MQRFH structure and its accompanying NameValueString, this tells the broker how long the NameValueString length in this message is as the MQRFH structure is fixed in size and the NameValueString is a variable length field. It also allows any application receiving this message to identify where the NameValueString ends and the next structure (if any) starts.
Alter the StrucLength field of the MQRFH to represent the size of the structure. The broker does not require a null terminator on the end of the NameValueString but it can make it easier for any other application that should need to read the string (if a null terminator is present it is possible to perform string operations against it, e.g. strlen),

adjust the StrucLength accordingly. The code to align the end of the MQRFH and NameValueString on a 16 byte boundary has been supplied, you do not need to include your own code.

Step 6.

In this sample we are publishing a character string as user data in each publication, this is a recognized MQSeries format that can be chained from the MQRFH and NameValueString structure. The MQRFH structure must indicate that the data following this structure is a string and also what character set it is in, alter the appropriate MQRFH fields to show this.

Step 7.

The user data (in this case the string data) must be positioned directly after the MQFRH and NameValueString structure, point the pUserData pointer to the starting position of any string data we will be adding.

Step 8.

Now that a line of string data has been read from standard input we can copy it into the publication message block. Once this has been done the publication message is complete and is ready to be put to the broker's stream queue.

Once all the above steps have been completed the sample is ready to be compiled. If compilation errors are displayed at this point you may have made a mistake in one or more of your code changes, use the information in the compilation errors to try and correct any mistakes. Once the sample compiles with no errors it is ready to be run.

At this point we do not have a subscriber sample to receive your publications but we can still try running the sample to see if the broker accepts the publications that you have generated. To run the publisher sample:

amqspub ***Topic*** *PUBLISHER_QUEUE* ***QMgrName***

Where Topic can be any valid topic string (not including spaces or double quote characters (")).

You should be prompted to enter a line of text to add to the publication, type *Hello* and press enter, what happens next? Has a success message been displayed or an error returned from the broker? If an error was displayed use the error message to try and work out what has happened. One technique which might help would be to run your application while the broker isn't running. Browsing SYSTEM.DEFAULT.LOCAL.STREAM using the amqsbcg sample will show you the format of any invalid messages sent to the queue by your application. Once you can run the sample and a success message is returned we have tested the publisher sample as well as we can without a subscriber sample so it is now time to move to the next project.

Subscriber Project

Now that we have a publishing sample we need a subscribing sample to subscribe to the publisher’s topic and receive the publications. The sample we will complete is *amqssub* and, like before, the source, *amqssuba.c,* can be found in directory *Tutorial\Exer3\amqssub*.

There are five steps within the sample that are to be completed, work your way through the code, understanding what has been done, when you reach an incomplete step follow the instructions to add the correct lines of code. There is more help in the source file preceding each step, use this in conjunction with the help given below.

The five steps to complete in the code are listed below

Step 1.

To be able to put the command messages to the broker we must first open the appropriate queue, complete the configuration of the MQOPEN command for putting messages to this queue.

Step 2.

In this sample we must first register our interest in a topic, so we must send a subscription registration to the broker. Because we will be sending different commands to the broker in this sample the function for generating a command message and sending it to the broker has been taken out of the main line of code and put into the function *SendBrokerCommand*, One of the arguments of this function is the command string to put into the NameValueString of the command message, add the appropriate command string for a subscription registration. (The function *SendBrokerCommand* should be similar to the section of code in amqspuba.c that builds the publication message).

Step 3.

Once we have registered as a subscriber we can wait for publications to arrive on our subscriber queue, as they arrive we read them in using MQGET. When we have a message we need to check that it is in the format we were expecting, not a message put to this queue from a different sort of application. So the first thing we must to do is check that the message is an MQRFH format message.

Step 4.

Now that we recognize the message as an MQRFH message we can locate the portion of the message that we, as a subscribing application, are interested in. In our sample we do not need to look at the NameValueString of the message, all we are interested in is the user data that follows it, the character string. Point the character string pointer pUserData to the start of the character string that follows the NameValueString and print it to the screen, if the null terminator was included by the publisher the *printf* function can be used.

Step 5.

We have now finished running the sample so we need to deregister our subscription that we made earlier, as in *Step 2* the command string has been left out of the call to SendBrokerCommand, add the appropriate string for a subscription deregistration.

Once all the above steps have been completed the sample is ready to be compiled. If compilation errors are displayed at this point you may have made a mistake in one or more of your code changes, use the information in the compilation errors to try and correct any mistakes. Once the sample compiles with no errors it is ready to be run.

To run this subscriber sample:

amqssub ***Topic*** *SUBSCRIBER_QUEUE* ***QMgrName***

Where Topic can be any valid topic string (not including spaces or double quote characters (“)).

This should display a message informing that we have successfully registered with the broker and we are waiting for a publication to arrive. As before, if you do not see this you should see an error message instead, use the error message to try and work out what has gone wrong and correct the sample appropriately.

Once the sample has registered successfully we can test the receiving of publications. Whilst *amqssub* is still running (restart it is necessary) return to the publisher project and start *amqspub* as before, make sure the topic string you supply matches that of the one you started *amqssub* with. Now enter text when prompted, this should be displayed in the command window running *amqssub*, as the subscriber is sent the publication you sent to the broker from *amqspub*. If this does not happen an error has occurred and you need to try and see where this has happened, please ask for assistance if you require it.

Once you are successfully receiving publications at your subscriber sample you can experiment with running multiple publishers and multiple subscribers in different command windows (Each subscriber will require a *different* subscriber queue, these will need to be defined as before using *runmqsc*, e.g. ‘SUBSCRIBER_QUEUE_2’, etc.). Try running with different topics on different publishers and subscribers, you can even try subscribing to a *wildcard* topic and publishing on different topics that match the wildcard.

Exercise 4. Setting up a Broker Network

So far we have only been running using a single broker. It is also possible to connect multiple brokers, each hosted by a different queue managers, together to form a *broker network*. Brokers are connected together to form a hierarchy, in this exercise our network will consist of two brokers running at two queue manage rs on the same machine. If you have never connected queue managers together using channels then consider just reading through this exercise and proceeding with Exercise 5 instead.

Before we create new queue managers and brokers we will end the broker and queue manager that we have so far been using, first end the broker and then the queue manager.

> *endmqbrk -m* ***QMgrName***
> *endmqm* ***QMgrName***

Now we will create a two broker hierarchy on our machine, for this we will create two new queue managers to host the brokers. A two broker hierarchy has a parent/child relationship, we will name the two queue managers *parent* and *child*.

As in section 1, to create and start the queue managers from the command line type

```
crtmqm -u SYSTEM.DEAD.LETTER.QUEUE parent
strmqm parent

crtmqm -u SYSTEM.DEAD.LETTER.QUEUE child
strmqm child
```

Neighboring brokers require their queue managers to communicate both ways between each other. To enable this we must define a channel from *parent* to *child* and from *child* to *parent*. The simplest method of achieving this is for each sender channel to reference a transmission queue which has the same name as its remote queue manager. The supplied channel definitions for these channels do this and assume that the TCP/IP protocol will be used. For more information about channels refer to the *MQSeries Intercommunication* guide..

The two *MQSC* script files, *parent.tst* and *child.tst*, can be found in *Tutorial\Exer4*, these script files contain the definitions to create the channels and necessary queues for the two brokers. They need to be altered first to configure them for your particular computer. Edit the two files and replace the word ***hostname*** with the hostname of your computer. The *MQSC* scripts have used *TCPIP* ports *1414* (for queue manager *child*) and *1415* (for queue manager *parent*), you may wish to change these to match your system.

Ensure that a *listener* for each queue manager has been started, then start the two channels, *parent.to.child* and *child.to.parent*.

Once we have the parent and child queue managers communicating with each other we can start the brokers. The broker on *child* will naturally be the child of the broker on *parent*. As in *Exercise 1*, to start the brokers type the following commands. It is recommended that you allow a delay of a few seconds between starting related brokers for the first time as this allows the parent broker to be running before the child registers with it.

```
strmqbrk -m parent
strmqbrk -m child - p parent
```

The ***-p*** option tells the *child* broker that *parent* is its parent.

We now have a broker hierarchy.

At this point we will introduce another sample that is supplied with MQSeries Publish/Subscribe, this sample, *amqspsd,* is a sample system management tool. It makes use of the administration and metatopic messages that each broker itself publishes to provide information on the current configuration and use of a broker.

The system management sample requires a subscriber queue for it to be defined before it is run. The *MQSC* script for this queue can be found in the *pubsub\admin* directory under the MQSeries C sample directory. The script file is *amqspsda.tst*.

Run this script for each queue manager :

```
runmqsc parent < amqspsda.tst
runmqsc child < amqspsda.tst
```

The sample accepts many different options for obtaining different levels of information from the broker, we will use it with its default settings at this point.

We will use this sample to first show if the two brokers recognize each other as relations. Run the sample against the parent queue manager, the executable can be found in the usual MQSeries sample directory,

```
amqspsd -m parent
```

You should see output like the one below, note how the child broker should be listed under the broker’s children heading, if it is not the broker does not recognize the second broker, there could be a problem with the channels, look for messages in the MQSeries error logs.

```
MQSeries Message Broker Dumper
    Start time: Sat Dec 12 17:13:04 1998

    Broker Relations
    ----------------
    QMgrName:
      parent
    Parent :
      None
    Children:
      child

    Streams supported
    -----------------
    SYSTEM.BROKER.DEFAULT.STREAM
    SYSTEM.BROKER.ADMIN.STREAM

    Publishers
    ----------
    StreamName: SYSTEM.BROKER.ADMIN.STREAM
      None
    StreamName: SYSTEM.BROKER.DEFAULT.STREAM
      None

    Subscribers
    -----------
    StreamName: SYSTEM.BROKER.ADMIN.STREAM
      Topic: MQ/parent                    /StreamSupport
        BrokerCount: 0
        ApplCount: 2
        AnonymousCount: 0
        RegistrationQMgrName: child
        RegistrationQName: SYSTEM.BROKER.INTER.BROKER.COMMUNICATIONS
        RegistrationCorrelId: 414D515701010000000000000000000000000000000000
        RegistrationUserIdentifier: mqm
        RegistrationOptions: 1 : MQREGO_CORREL_ID_AS_IDENTITY
        RegistrationTime: 1998121217112523
        RegistrationQMgrName: parent
        RegistrationQName: SYSTEM.BROKER.INTER.BROKER.COMMUNICATIONS
        RegistrationCorrelId: 414D515901010000000000000000000000000000000000
        RegistrationUserIdentifier: mqm
        RegistrationOptions: 17 : MQREGO_CORREL_ID_AS_IDENTITY
    MQREGO_NEW_PUBLICATIONS_ONLY
        RegistrationTime: 1998121217111404
      Topic: MQ/S/parent                   /Subscribers/Identities/*
        BrokerCount: 0
        ApplCount: 1
        AnonymousCount: 1
    StreamName: SYSTEM.BROKER.DEFAULT.STREAM
      Topic: MQ/S/parent                   /Subscribers/Identities/*
        BrokerCount: 0
        ApplCount: 1
        AnonymousCount: 1
```

Run the sample again, this time against the *child* queue manager. This time you should see the name of the parent queue manager under the parent heading, if you do not see this the broker does not recognize the parent broker as its parent, then again look for messages in the MQSeries error logs.

As you can see, other information is also displayed under the subscriber heading. The subscriptions we can see to ***MQ/...*** topics are those subscriptions currently registered by the broker and its neighbor, or the system management sample itself (these will be deregistered once the system management sample completes, the sample has registered as anonymous so the details are not visible under normal circumstances).

Define the necessary queues required to run the samples you completed in *Exercise 3* on the two new queue managers, start by defining the subscriber queue on the child queue manager and the publisher on the parent queue manager. Now start the subscribe sample, *amqssub*, on the child queue manager and the publisher sample, *amqspub*, on the parent queue manager. You should see publications arriving at the subscriber sample exactly as before in a single broker network. Extend this test by adding publishers and subscribers on either of the two queue managers, they should all perform exactly as you would expect for a single broker network (remember that each subscriber will require its own unique queue and a publisher queue will be needed on each broker that a publisher runs on).

Whilst these samples are still running run the system management sample on each of the brokers, you should now see more subscriptions registered on the default stream.

Exercise 5. Understanding and Extending the Results Server Sample

We have already seen the results server samples running, but the way in which they work was only briefly mentioned, we will now discuss these samples in more detail and how they implement a number of different features of the MQSeries Publish/Subscribe function. Once we understand how these samples work we can extend their functionality.

If we look at the two parts of the results server sample independently we can compare them against the simple publisher and subscriber that we completed earlier. The source for these samples can be found under the MQSeries C sample directory in the subdirectory *pubsub*.

Match Simulator, amqsgam

Source: *pubsub\amqsgama.c*

The match simulator is just a simple publisher like our *amqspub* sample, with a little logic to simply simulate a soccer match (random goal scoring within a fixed period). The sample publishes event publications (not *retained*) on three different topics*, Sport/Soccer/Event/MatchStarted, Sport/Soccer/Event/MatchEnded* and *Sport/Soccer/Event/ScoreUpdate*, depending on what event has occurred. The structure of the user data (character string format in all cases, *MQFMT_STRING*) varies according to the event being published, if the match is starting or ending the names of both teams are required by the results service, if a goal has been scored the only required data is the name of the team scoring, the results service can deduce from this which match the team is playing in and adjust the score accordingly (we can only ever increment a teams score by one).

In one respect *amqsgam* is simpler than *amqspub*, *amqsgam* sends all publications as *datagram* messages (does not request a response from the broker), this is purely for simplicity, we would not recommend an application did this (did not request a response from the broker on receiving a command or publication), especially in a test environment where more errors are likely to occur. If a broker rejects a publication or is unable to process it correctly the publisher would not be informed. The preferred method of sending publications is to send them as datagrams with a report option of *MQRO_NAN*, that is *negative replies only*. In this case a reply will only be generated and returned by the broker if a problem occurs in processing a publication. This way we do not have the overhead of one reply per publication but we do have the ability to know when a problem occurs. A separate thread or process should be used to wait for error responses as their arrival will be infrequent, if at all, and this would impede performance of the publishing application if it was to wait for a message after each publish. The use of *MQRO_NAN* has been implemented in *amqsres* when publishing *LatestScore* publications, the negative responses (if any) are read in by the main *MQGET* loop processing arriving messages (normally event publications).

When running multiple match simulators we have multiple publishers publishing on the same set of topics simultaneously, these publications are *event* publications (not *retained*) and it is perfectly valid to have more than one publisher of these. We do not recommend that more than one publisher publishes *state* publications (retained) simultaneously as it is then difficult to determine which publication has been retained, and there is a possibility of different publications being retained on the same topic on different brokers at the same time.

Another feature of publishing event publications is that any subscribers wishing to receive these publications must be registered as a subscriber before the publication is published, it must also be noted that the subscription must have had time to propagate across the broker network to the broker(s) from which the publications are being published. In this example we are publishing on a new stream, if the match simulator sample (publisher) was to start before a subscription had arrived the broker would have no knowledge of the stream and the publications arriving on the stream queue would not be processed until the broker was informed of the stream (in this case by a *register subscriber* command).

Results Server, amqsres

Source: *pubsub\amqsresa.c*

The results server is actually a subscriber and a publisher, as was mentioned when we first saw the sample. It subscribes to all event publications, *Sport/Soccer/Event/**, and publishes and subscribes to the state publications, *Sport/Soccer/State/LatestScore/** (not simultaneously).

Under normal conditions the sample has an active subscription to *Sport/Soccer/Event/**, all publications from a match simulator are then sent to the results server. On receiving an event publication the results server updates the current state of the appropriate match in memory and also publishes a retained publication on the topic *"Sport/Soccer/State/LatestScore/Team1 Team2"*, where *Team1* and *Team2* are the two teams playing in the match, the user data of the publication is the current score of the match. This retained publication replaces any existing one for this match. The topic is unique to the match (as it includes the team names) and therefore, there will be one retained publication on the broker for each match currently being played. Once a match finishes (a *Sport/Soccer/Event/MatchEnded* publication is received for the match) the match state is removed from memory and the results server issues a *Delete Publication* command on the "*Sport/Soccer/State/LatestScore/Team1 Team2*" topic to remove the retained publication held for this topic. This means that under normal circumstances when running the results service any retained publications put to the broker as the result of a match will be removed before the results server has ended. There are two cases when this is not true. One is when a *Sport/Soccer/Event/MatchEnded* publication is not received for a match it is currently reporting on (perhaps the match simulator was prematurely stopped or a publication has been put to a dead letter queue). The other is when the results server is prematurely stopped, this is the specific case why we publish on "*Sport/Soccer/State/LatestScore/Team1 Team2*".

In the case when the results server was prematurely stopped we would like to be able to restart the results server and continue with the results from the state in which they were left. So every time we start the results server, before subscribing to *Sport/Soccer/Event/**, we subscribe to *Sport/Soccer/State/LatestScore/**, on receiving the retained publications that exist (if any) we restore the state of the matches, as represented by the *Sport/Soccer/State/LatestScore/** publications. Once this has been done we can deregister our subscription from the *Sport/Soccer/State/LatestScore/** topic and then register the subscription to *Sport/Soccer/Event/** and start processing the event publications that arrive. This gives us the ability to stop and start the results server at any time whilst matches are being played. One MQSeries feature that has not been used in this sample that would improve the reliability of the restart would be to use *units-of-work*, these would start with the getting of the event publication and complete at the point that the *LatestScore* publication is put. This would mean that if the results server was halted between these two operations (the MQGET and the MQPUT) the original MQGET would be backed out and on restart the event publication would still be available to update the restored state.

If the results server is ended prematurely the subscription to *Sport/Soccer/Event/** is still active (no deregister subscriber message has been sent) and event publications will continue to be sent to the results server's subscription queue. Therefore, on startup we wish to read in all the *LatestScore* retained publications *before* we start to process any event publications, to achieve this we could have used a separate queue as our subscriber queue for the *Sport/Soccer/State/LatestScore/** subscription, but this would require yet another queue to be defined. A cleaner method is to subscribe with a different identity, the same queue name and queue manager name but a different *CorrelId*. By specifying a CorrelId as part of our subscriber identity the broker will always put the publications for this subscriber on the subscriber queue with the specified CorrelId. We can therefore, use the *get message options* of *MQGET* to only get the messages that match this CorrelId. So we first get all the messages that match the CorrelId of the *Sport/Soccer/State/LatestScore/** subscription, once we have restored the state we can subscribe to *Sport/Soccer/Event/** with a different CorrelId and get all messages that match this CorrelId, which will include those that arrived on the subscriber queue whilst the results server was stopped.

To minimize the amount of user data that is added to the publication message, and to avoid repeating data, the results server actually uses information contained in the *NameValueString* of the publication, specifically the *MQPSTopic* value. As we subscribe to more than one topic (by using wildcards) we do not automatically know what topic the publication we are looking at is for, so we must find the topic value in the *NameValueString* and process the publication appropriately. The results server sample includes a function that tokenizes the *NameValueString* into name/value pairs. We could have used a very simple method based on the fact that the tokens in the string are space delimited but there is the possibility that a value contains spaces (if the value is enclosed by quotes), as is the case with the *“Sport/Soccer/State/LatestScore/Team1 Team2*” topic. The tokenizer function used in *amsresa.c* is called *GetNextFunction()* and can be used in other applications requiring this functionality.

Extending the Results Service Sample

To further familiarize yourself with writing MQSeries Publish/Subscribe applications this exercise will allow you to extend the functionality of the results server sample that we have already seen.

The results server maintains the state of matches currently being played (by using retained publications), this state is removed once a match ends. We will extend the results server sample, *amqsresa.c*, to publish a retained publication on a new topic once a match ends, this publication will include the same details as in the *LatestScore* publications, the names of the teams playing and the score at the end of play. These publications will remain on the broker and not be deleted. Remember, only one publication is retained on each topic so each match must have a unique topic (as currently implemented for the *LatestScore* publications). One characteristic of this new function to note is that only the last match played between two teams is held as a retained publication, if the same match is replayed later the original FinalScore publication holding the first match result will be overwritten.

Now we require a third sample application, very similar to the subscriber *amqssub.exe* that we wrote earlier. This sample will subscribe to the topics that we publish the final scores on, when a publication is received the match details are in the publication's user data and displayed to the screen.

Results Service Project, amqsres2

A modifiable version of the results server can be found in *Tutorial\Exer5\amqsres2*. Follow these steps to complete a modified results service. Compile the source file into the executable *amqsres2* (*amqsres2.exe* for *Windows NT*) once the steps below have been completed.

The following steps extend the *UpdateLatestScorePub()* function to publish on the *FinalScore* topic once a match has ended. Read the code that has been added to this function and understand how it works.

Step 1.

Build the topic string that we will be publishing on, this is unique to each match being played, it is of the form:

```
Sport/Soccer/State/FinalScore/Team1 Team2
```

Step 2.

We call the function *BuildMQRFHeader()* to build the *MQRFH* structure and *NameValueString* for the publication, two of the arguments required have been left out of your source, the command string needed for a publication message and the publication options for this publication (in *decimal* form (using the defined MQPUBO_ options) not a string), use the Publish/Subscribe manual to find the MQ constants required. Complete this function call.

Step 3.

Finally we must add the user data to the publication, for this publication we generate a string that will be displayed by the final score sample, the string must contain all the details of the matches final score.

Final Score Project, amqsfin

To create the final score sample modify the source file in *Tutorial\Exer5\amqsfin*. This is basically a slightly modified version of the subscriber sample from *Exercise 3*, this time the source file is called *amqsfina.c* but it is very similar to *amqssuba.c*, follow the next step to complete the final score sample. The only differences to the simple subscriber sample is that the topic we subscribe to is defined (not a user argument) and we are not using the default stream.

Step 1.

In the function *SendBrokerCommand()* build the *NameValueString* of the command messages sent to the broker's control queue. This is a similar *NameValueString* between the subscriber registration and deregister, the only difference being the command string which is passed into the function as an argument. The *NameValueString* must contain all the name/value pairs required for a register/deregister of a subscriber, see the Publish/Subscribe manual for details of these commands, remember that we are not subscribing to a topic on the default stream so the stream name is required.

Once both of the above projects compile cleanly we can run the new samples. We will use the single broker which we created in Exercise 1. If this has been ended then restart it.

To run the final score sample, *amqsfin*, we require another queue in addition to the two previously defined in the *MQSC* script amqsresa.tst, define *FINAL.SCORE.SAMPLE.QUEUE* as a local queue on ***QMgrName,*** this will be the subscriber queue used by *amqsfin*

```
runmqsc QMgrName
    define qlocal(FINAL.SCORE.SAMPLE.QUEUE)
```

Unlike the results service it is not important when the final score sample is started as it will receive any existing *FinalScore* retained publications when it is started. This time we will start the final result sample, *amqsfin*, first. From the command line in the tutorial directory *Tutorial\Exer5\amqsfin*, run this at the ***QMgrName*** broker.

amqsfin ***QMgrName***

Now start the new results service from another command line in *Tutorial\Exer5\amqsres2*, run this also at the ***QMgrName*** *broker.*

amqsres2 ***QMgrName***

Finally start up one or more of the original match simulators, *amqsgam*, or use the multiple match script file *amqsmany* (from *Exercise 2*) to start three at once.

The match simulator(s) and results service should perform exactly as before, once a match ends you should see the final match score being displayed by the final score sample.

Exercise 6. Deleting a Broker Hierarchy

Now that we have completed the exercises on the two-broker hierarchy we created in *Exercise 4* we can delete them.

We recommend that a broker hierarchy is deleted using a *bottom-up* approach, i.e. start by ending and deleting the brokers on the lowest levels of the hierarchy, once these are removed continue up the hierarchy until the root broker can be deleted.

To delete a broker it must first be ended, but *not* the parent broker as this will be sent a message from the child informing it that the child is deleting, it can then remove its knowledge of this child, a broker cannot be deleted until it has no registered children.

```
endmqbrk -m child
```

When a broker is deleted the queues defined when the broker is first started are removed, it is possible that some of these may be held open by the parent broker via a channel, whilst this is true the queue cannot be deleted and thus, the broker cannot complete the deletion. The parent's hold on these queues will eventually time-out and the child broker can then be deleted, to speed this process up we can break the parent's hold by stopping and starting the parent-to-child channel, this can be done from *runmqsc*.

```
runmqsc parent
   stop channel('parent.to.child')
   start channel('parent.to.child')
   end
```

Once this has been done we can delete the child broker.

```
dltmqbrk -m child
```

If you now use the system management sample *amqspsd* to display the configuration of the parent broker you will see that the parent has no children registered.

```
amqspsd -m parent
```

To delete the parent broker we follow the same method as used for the child, remembering to stop and start the child's channel to the parent.

End the broker running on *parent.*

```
endmqbrk -m parent
```

Stop and start the channel from *child* to *parent*.

```
runmqsc child
   stop channel('child.to.parent')
   start channel('child.to.parent')
   end
```

Delete the broker on *parent.*

```
dltmqbrk -m parent
```

We have now removed both brokers from the child and parent queue managers.

This completes the exercises for the tutorial.