JMQ

Reference Manual

Version 1.0

Author: Prasad Polukonda

Document Purpose

This document details the internals of JMQ API which would be helpful to develop Java applications that requires publishing/subscribing the data through MQ Series channel with very minimal development effort.

Contents Page

REFERENCE MANUAL **1**

Document Purpose *1*

This document details the internals of JMQ API which would be helpful to develop Java applications that requires publishing/subscribing the data through MQ Series channel with very minimal development effort. *1*

CONTENTS PAGE **1**

DOCUMENT CONTROL **1**

1 OVERVIEW **2**

1.1 KEY FEATURES 2

2 UML REPRESENTATION & CODE SNIPPETS **3**

2.1 MQPUBLISHER 3

2.1.1 MQPublishParams *3*

2.1.2 Code Snippet *4*

2.2 MQLISTENER 5

2.2.1 MQListenerParams *6*

2.2.2 Code snippet: *6*

3 SETTING UP MQ QUEUEMANAGER AND CHANNELS **7**

4 GENERAL INFORMATION ON MQ **22**

4.1 MQ MESSAGE 22

4.2 QUEUE MANAGER 23

Document Control

a. Document Ownership

b. Document History

The following table records information regarding released versions of this document and briefly describes changes between versions.

Version	Date	Author	Comment / Changes from Prior Version
1.0	27/10/2011	Prasad Polukonda	Initial Version

1 Overview

JMQ is a simple API that allows Java applications to connect to MQ Series server and to issue the calls and queries to MQ. With JMQ API, applications can seamlessly publish/subscribe data object models (POJOs - Plain Old Java Objects) to/from the MQ Channels. Application developers can also create publisher/listeners directly on messages that represent character sequence. This API abstracts the complexity of making a connection to MQ Channels, setting MQQueue OPEN options, setting GET/PUT MQMessage options, populating header information like: message Id, correlation Id, automatic conversion of application POJOs to MQ Message (default format: XML) and vice versa. However, this API provides place holders to supply your own messaging options and enables the developers to define their format and to supply their own data transformers.

1.1 Key Features

- Ease of Use/Developer Friendly: This API is very straight forward to use. Create MQ publishers/listeners on any POJO Model and start using it. It is not always necessary to deal with POJOs and the XML format. Developers can create publishers/listeners directly on a message that represents a character sequence. From the application perspective, developers can always deal with their object model and this API will take care of the low level details of message handling.

- Subscribe to only the required fields: On subscription, it is not always necessary that applications will be interested in all the fields of the MQ message. This API allows the applications to define a POJO that contains subset of fields and start listening to that. On getting the message from MQ, the listener call-back will be notified with the object of the model the application defined.

- Reduces the complexity: There are number of options available in opening a queue, setting the PUT/GET message options, setting the header information etc. It is always easy to determine which are the right options to use are and developers can spend most of the development time on exploring with different options. The default implementations provided by this API will use most appropriate options that will be suitable for most of the real-time applications. However it allows the developers to supply their own options which are suitable for their application needs.

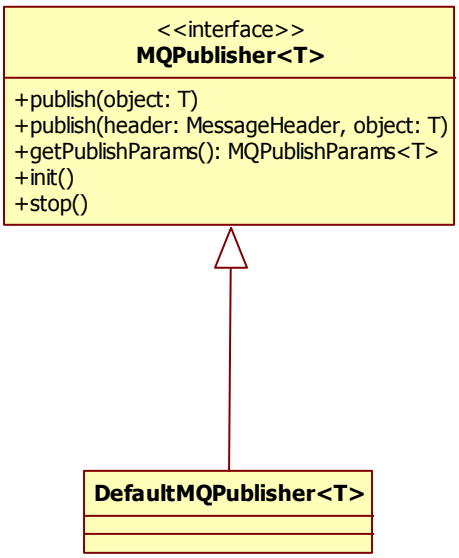
- No extra performance over-head: This API is a kind of wrapper API around MQ API and uses the capabilities of XStream for data parsing. The performance overhead on using this API is very negligible when compared to directly using the raw API. The overhead is observed as around 1 millisecond for 1 million messages.

2 UML Representation & Code Snippets

This particular section covers only the main interfaces and classes important for developers to know.

2.1 MQPublisher

An instance of MQPublisher will be used to publish the POJOs to MQ channels. The default implementation: DefaultMQPublisher will take care of transport mechanism, setting MQ Queue options, setting PUT options etc.

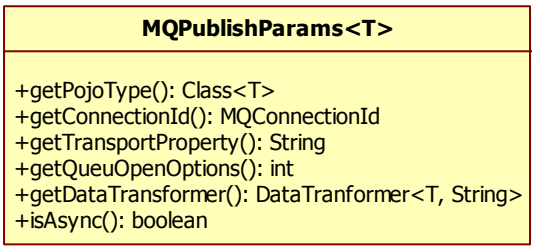


Constructor:

```
public DefaultMQPublisher(MQPublishParams<T> publishParams)
```

2.1.1 MQPublishParams

This class encapsulates all the properties that will be used by a publisher. Most of the properties are optional except PojoType and MQConnectionId



2.1.2 Code Snippet

The following code snippets demonstrating an example to send a trade message from a system called: ANVIL to a system called: FISTAR. The corresponding queue channels/queue managers need to be setup in order to run the following example or you can setup your channel by your-self (see section 4 for more details).

a) Define a POJO class called: Trade

```
public class Trade {

    private String id;
    private double qtyNominal;
    private double price;
    private Verb verb; // enum type

    public Trade() {
    }

    public Trade(String id, double qtyNominal, double price, Verb verb) {
        this.id = id;
        this.qtyNominal = qtyNominal;
        this.price = price;
        this.verb = verb;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public double getQtyNominal() {
        return qtyNominal;
    }

    public void setQtyNominal(double qtyNominal) {
        this.qtyNominal = qtyNominal;
    }

    public double getPrice() {
        return price;
    }
```

```
    public void setPrice(double price) {
        this.price = price;
    }

    public Verb getVerb() {
        return verb;
    }

    public void setVerb(Verb verb) {
        this.verb = verb;
    }
}
```

b) Publish a trade object without message header

```
MQPublishParams<Trade> publishParams = new MQPublishParams<Trade>(Trade.class,
new DefaultMQConnectionId(
                "10.132.99.190", 2229, "CLIENT.QM.ANVIL", "Q1", "qm.anvil",
null, null));

// create and initilize the publisher
MQPublisher<Trade> publisher = new DefaultMQPublisher<Trade>(publishParams);
publisher.init();

// Create a trade object and publish
Trade emp = new Trade("Anvil_RepoTrade_001", 1000000, 102.56, Verb.Sell);
publisher.publish(emp);
```

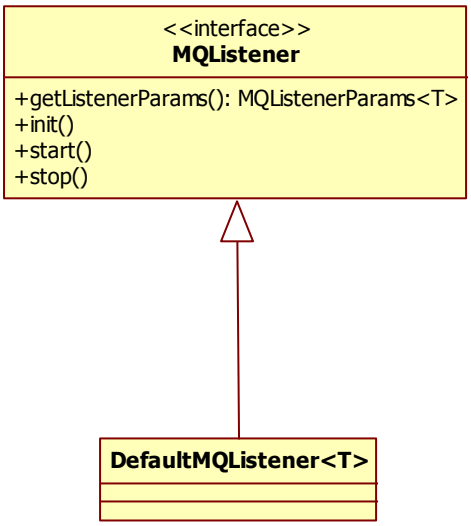
c) Publish a trade object with message header

```
// create and initilize the publisher
MQPublisher<Trade> publisher = new DefaultMQPublisher<Trade>(publishParams);
publisher.init();

// Create a trade object and publish
Trade emp = new Trade("Anvil_RepoTrade_001", 1000000, 102.56, Verb.Sell);
MessageHeader header = new DefaultMessageHeader(emp.getId(), emp.getId());
publisher.publish(header, emp);
```

2.2 MQListener

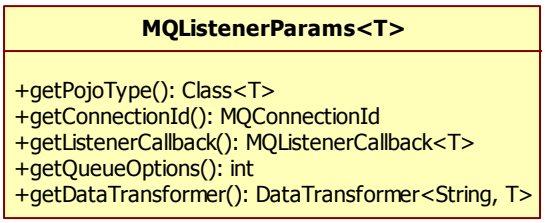
An instance of this interface will be used to listen to a particular POJO model that represents the MQ message. Your POJO doesn't need to represent the full MQ message, it can represent subset of fields.



The default implementation: DefaultMQListener will take care of setting queue open options, setting the GET message options, parsing the message header information and convert them into more readable format, sending the acknowledgements back to the reply queues, notifying application listener call-back classes with POJO objects.

2.2.1 MQListenerParams

This class encapsulates all the properties that will be used by a listener. Most of the properties are optional except PojoType, MQConnectionId and MQListenerCallback



2.2.2 Code snippet:

```
MQListenerParams<Trade> listenerParams = new
MQListenerParams<Trade>(Trade.class, new DefaultMQConnectionId(
                "10.132.99.190", 2228, "CLIENT.QM.FISTAR", "Q1", "qm.fistar",
null, null), new DemoListenerCallback());

MQListener<Trade> listener = new DefaultMQListener<Trade>(listenerParams);
listener.init();
listener.start();
```

Listener call-back class:

```
public class DemoListenerCallback implements MQListenerCallback<Trade> {

        @Override
```

```
        public void onUpdate(MessageHeader messageHeader, Trade object)
throws Exception {
            System.out.println(System.nanoTime() + " MessageHeader:"
                    + messageHeader.getMessageId()
                    + " - Trade object received:" + object);
            System.out.println(messageHeader.getMessageId().length());
        }
    }
```

For API level details, please go through the Java Docs of JMQ.

3 Setting up MQ QueueManager and Channels

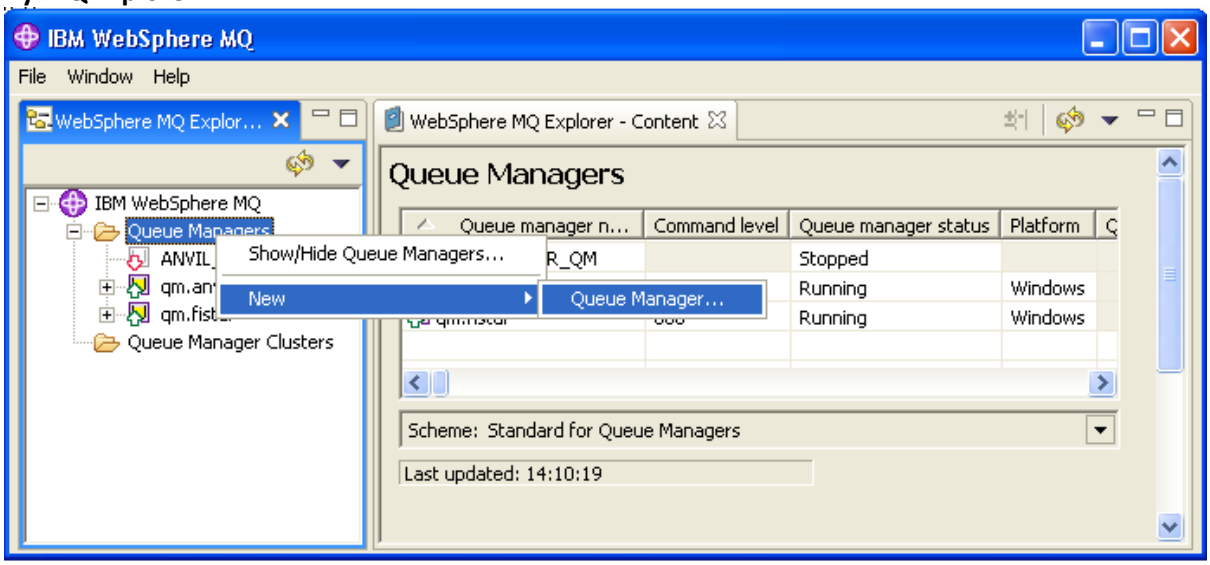
The following example demonstrates how to setup queue managers, local/remote/transmission queue definitions and channels etc. throw MQ Explorer.

Case: We need to feed some trade messages from system: Anvil to a system called: FISTAR. The two systems might be running on different machines and the MQ server is running on a different machine.

Steps:

1) Setup two queue managers called qm.anvil and qm.fistar

By MQ Explorer:



Enter queue manager name: qm.anvil

Follow the same step for creating the queue manager qm.fistar

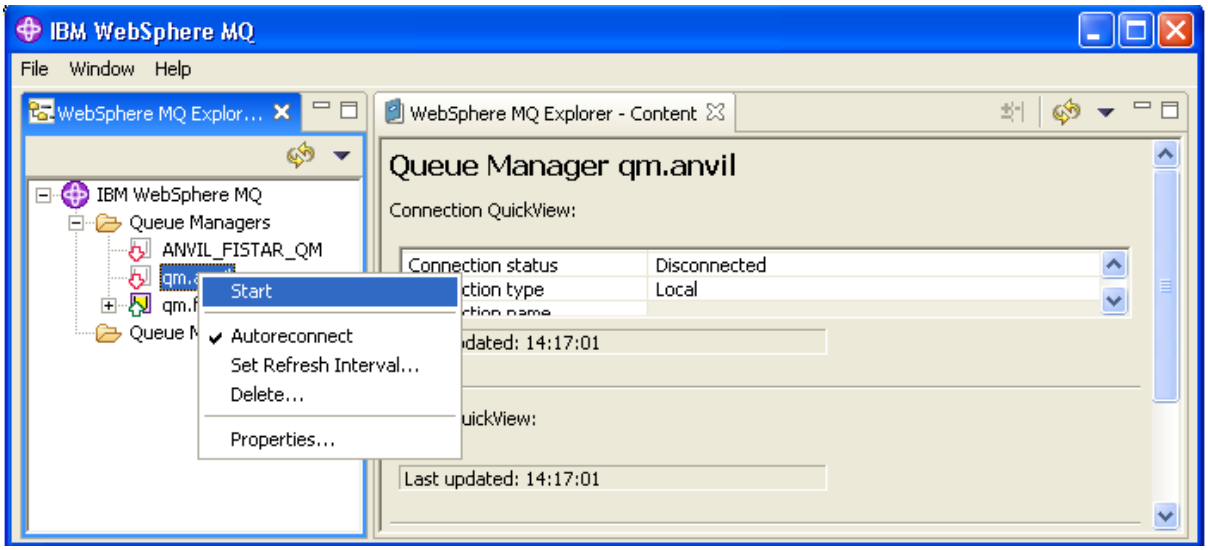
By using console commands:

crtmqm qm.anvil

crtmqm qm.fistar

2) Start the queue managers qm.anvil and qm.fistar

By MQ Explorer:



By using console commands:

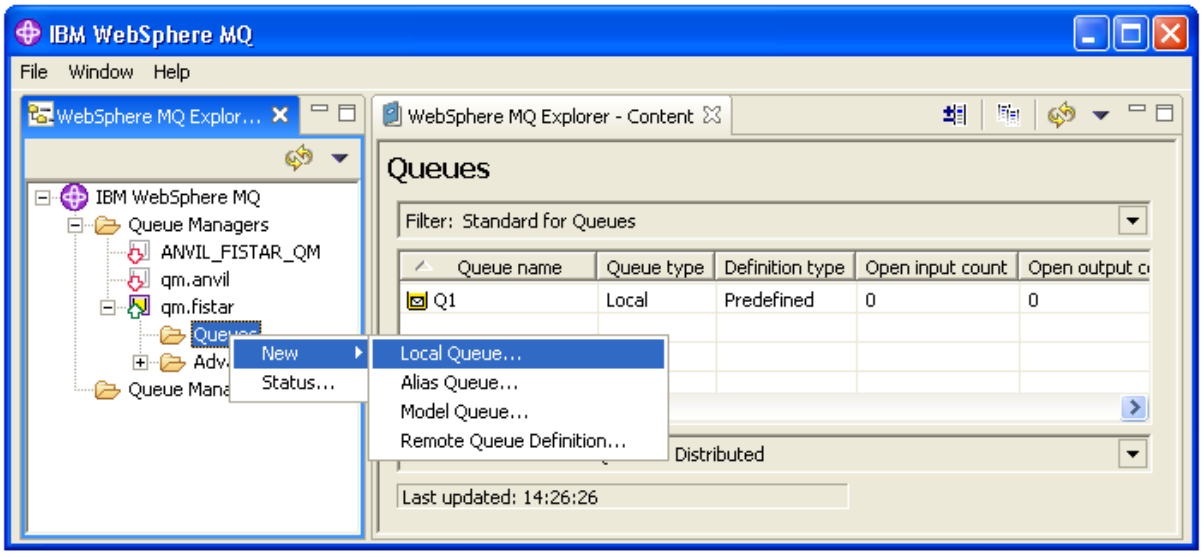
strmqm qm.anvil

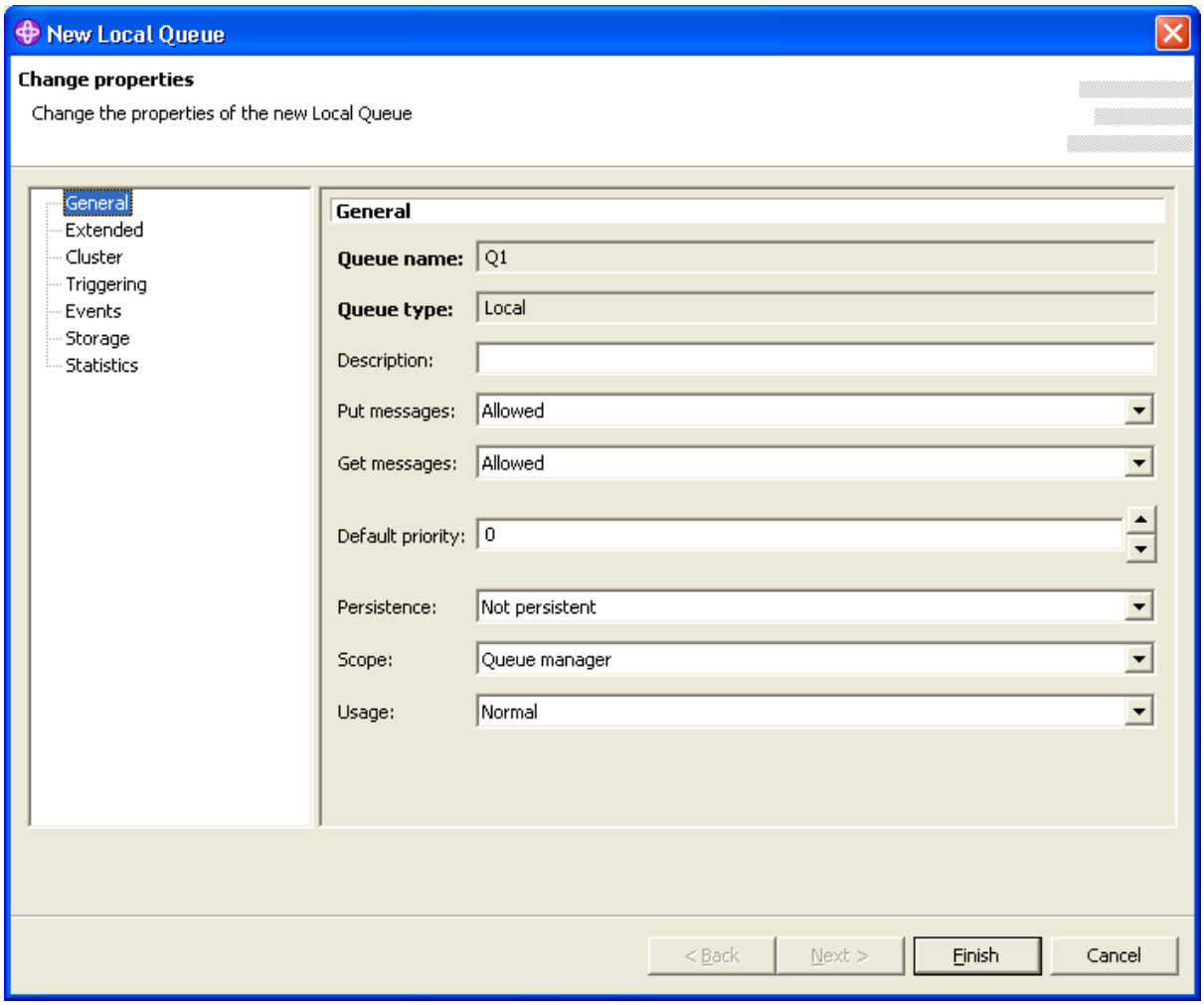
strmqm qm.fistar

3) We now have two queue managers. What we want to do now is create a queue to receive messages on qm.fistar. Remember we just need a local queue definition on qm.fistar for receiving the messages.

Make sure the queue manager : qm.fistar is running otherwise follow step:2

By MQ Explorer:





By using console commands:

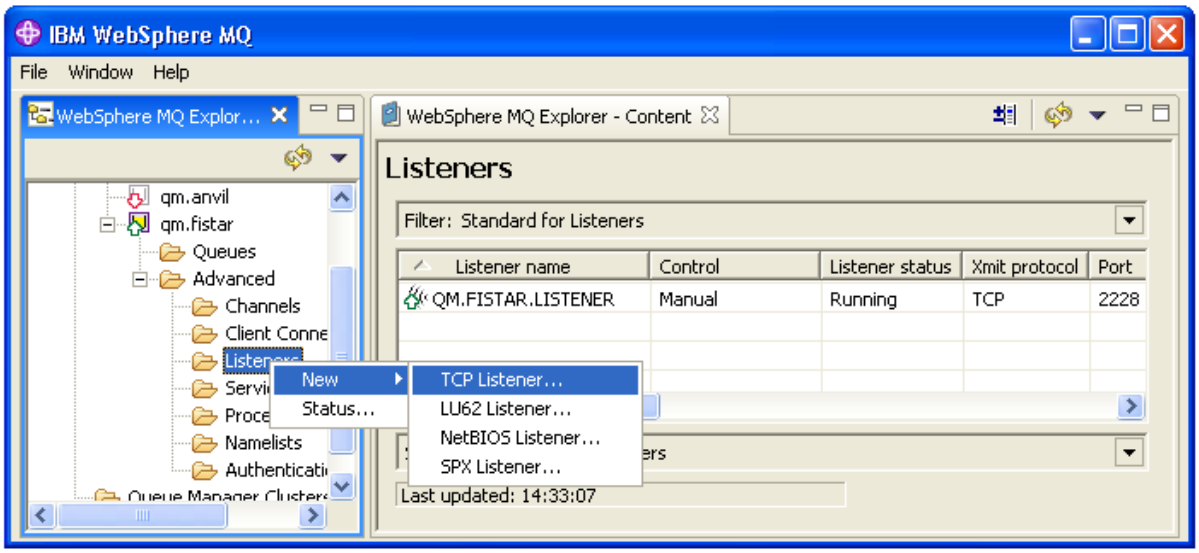
a) runmqsc qm.fistar

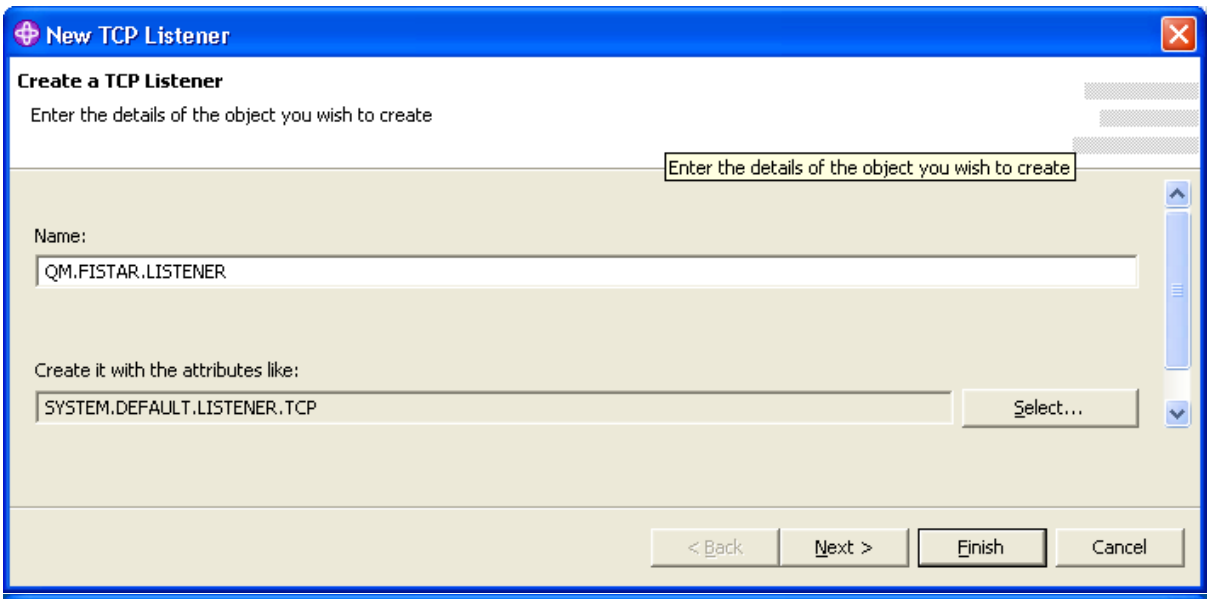
b) define qlocal(Q1)

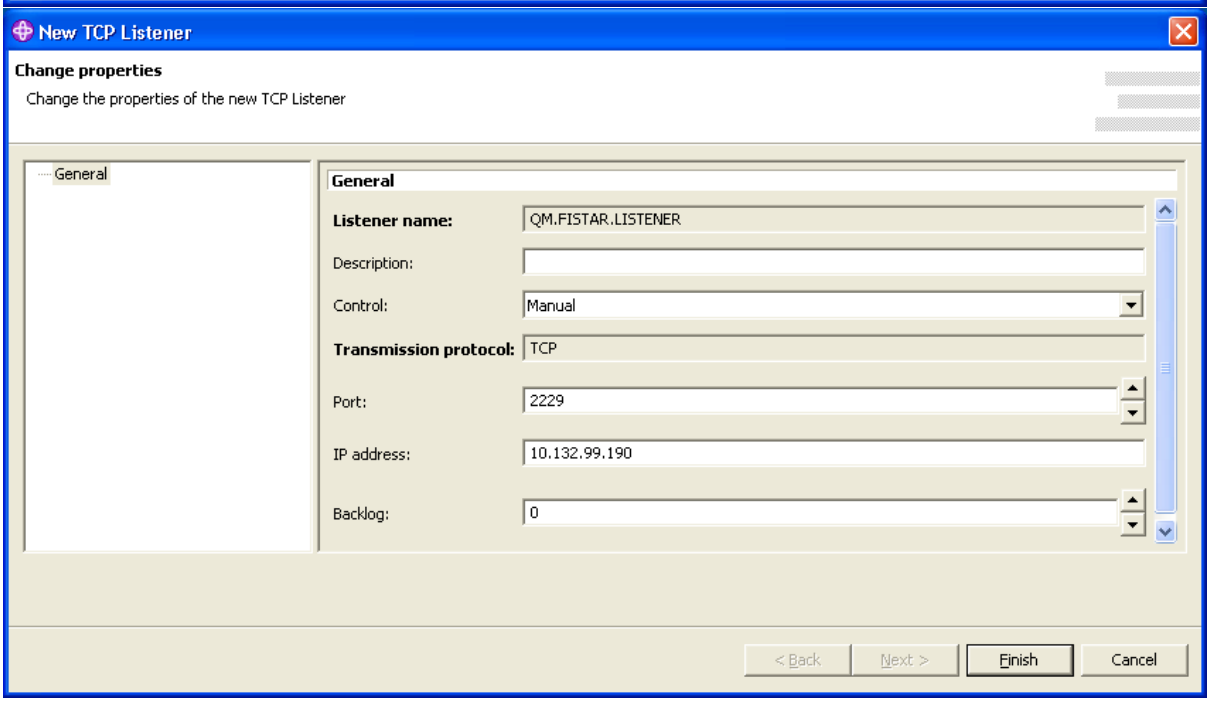
c) end

4) What we need to do now is create a listener for the qm.fistar which allows our sending queue manager: qm.anvil to connect via channel. We are going to make the listener use port 2229.

By MQ Explorer:


IBM WebSphere MQ
File Window Help
WebSphere MQ Explor...
WebSphere MQ Explorer - Content
Listeners
Filter: Standard for Listeners
Listener name
Control
Listener status
Xmit protocol
Port
QM.FISTAR.LISTENER
Manual
Running
TCP
2228
qm.anvil
qm.fistar
Queues
Advanced
Channels
Client Conne
Namelists
New
Status...
TCP Listener...
LU62 Listener...
NetBIOS Listener...
SPX Listener...
Last updated: 14:33:07


New TCP Listener
Create a TCP Listener
Enter the details of the object you wish to create
Enter the details of the object you wish to create
Name:
QM.FISTAR.LISTENER
Create it with the attributes like:
SYSTEM.DEFAULT.LISTENER.TCP
Select...
< Back
Next >
Finish
Cancel


New TCP Listener
Change properties
Change the properties of the new TCP Listener
General
General
Listener name:
QM.FISTAR.LISTENER
Description:
Control:
Manual
Transmission protocol:
TCP
Port:
2229
IP address:
10.132.99.190
Backlog:
0
< Back
Next >
Finish
Cancel

After finish, start the listener by right click on the created listener.

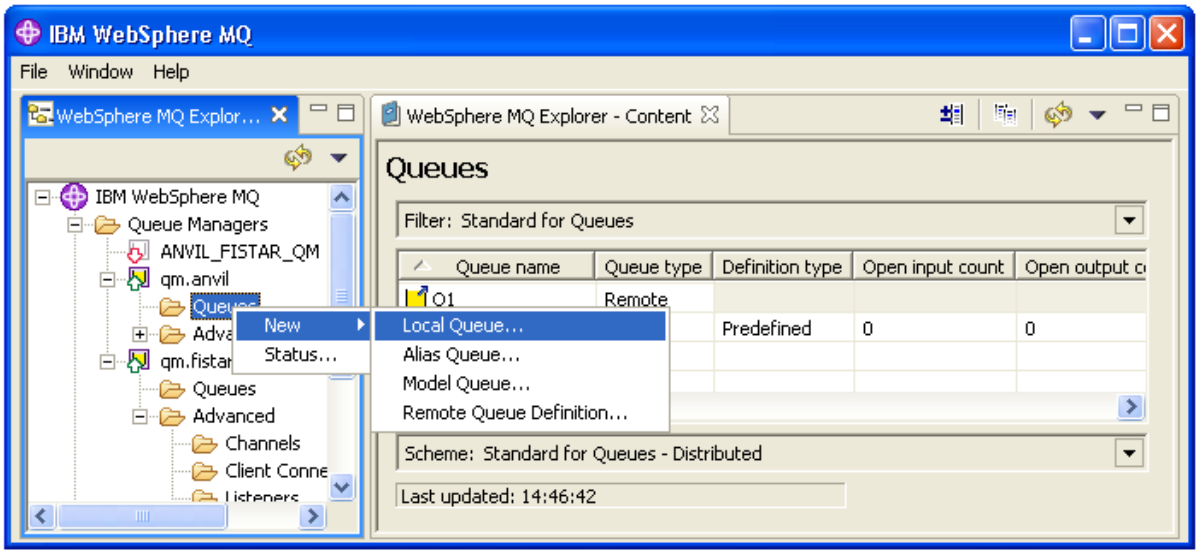
By console commands:

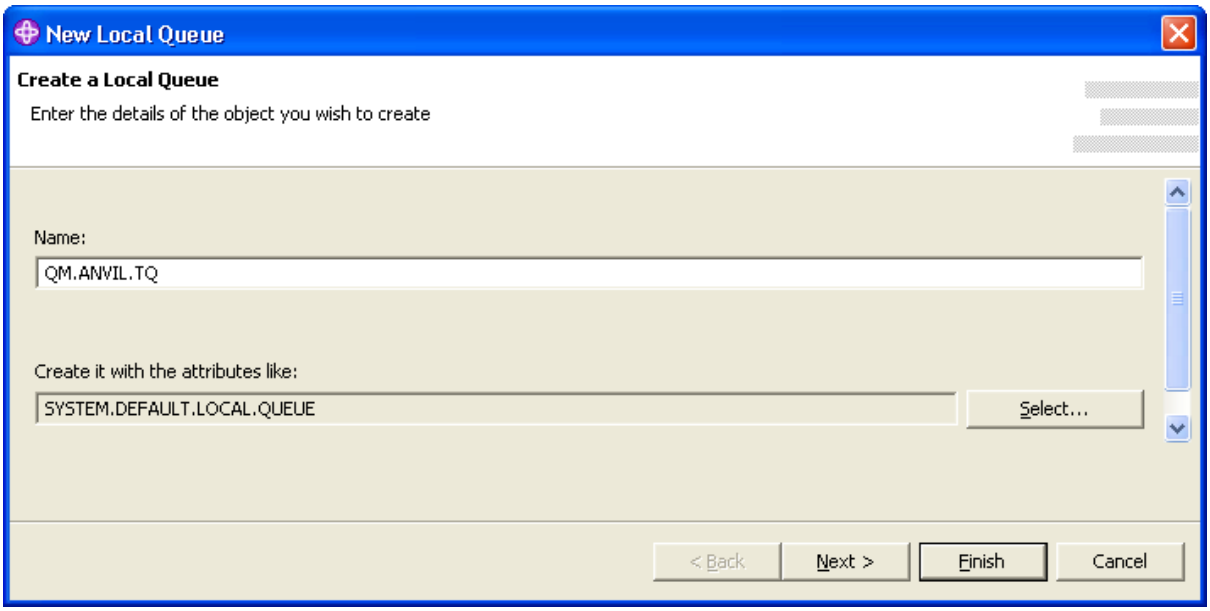
a) runmqsc qm.fistar
b) define listener(QM.FISTAR.LISTENER) trptype(TCP) port(2229)
c) start listener(QM.FISTAR.LISTENER)
d) end

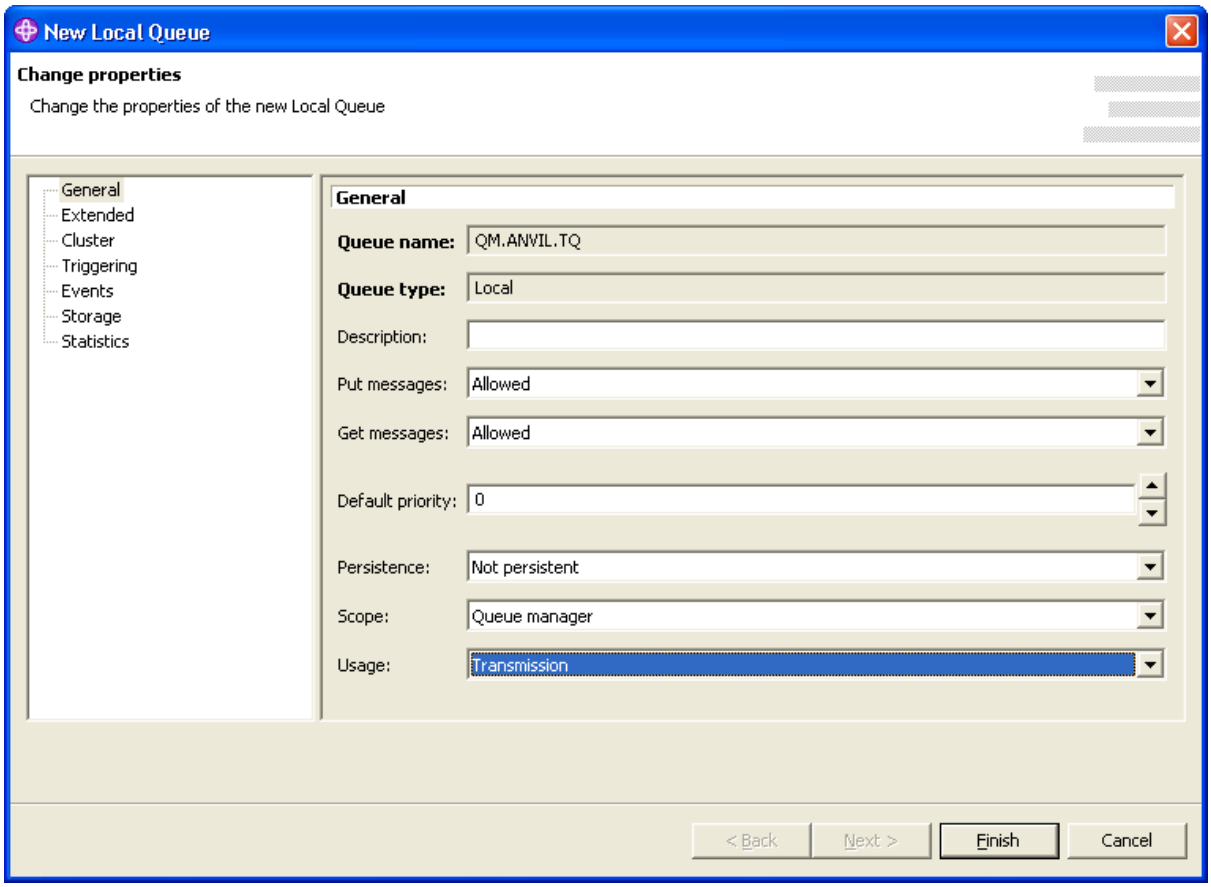
NOTE: Remember firewalls will need to be configured if you have them to allow communication on these ports.

5) What we need to do now is create a transmission queue on qm.anvil. We require a transmission queue to be able to talk to a channel object. We use a channel to allow qm.anvil to talk to qm.fistar.

By MQ Explorer:





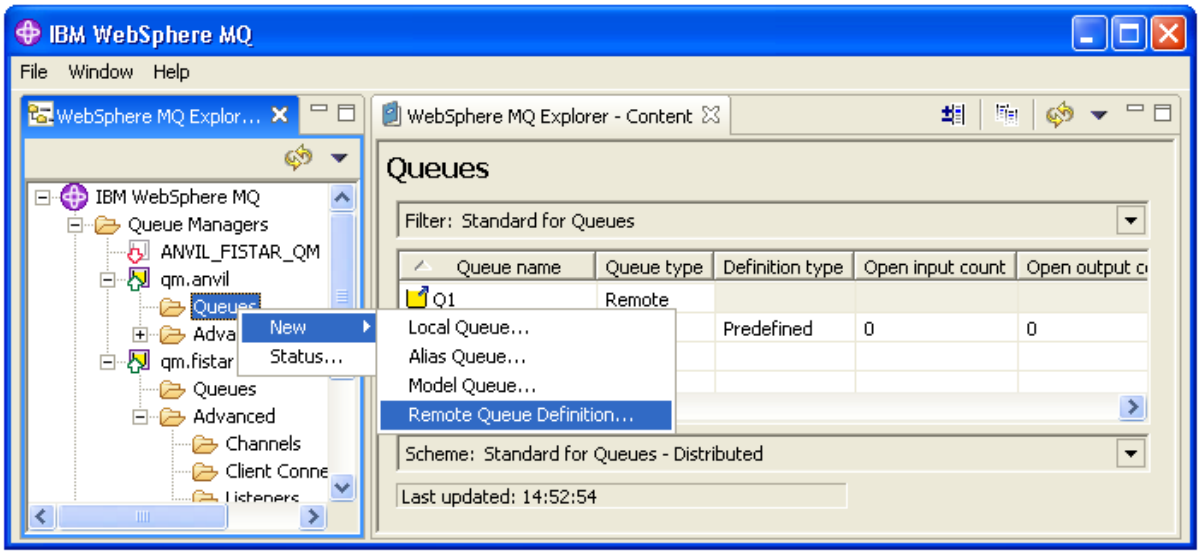


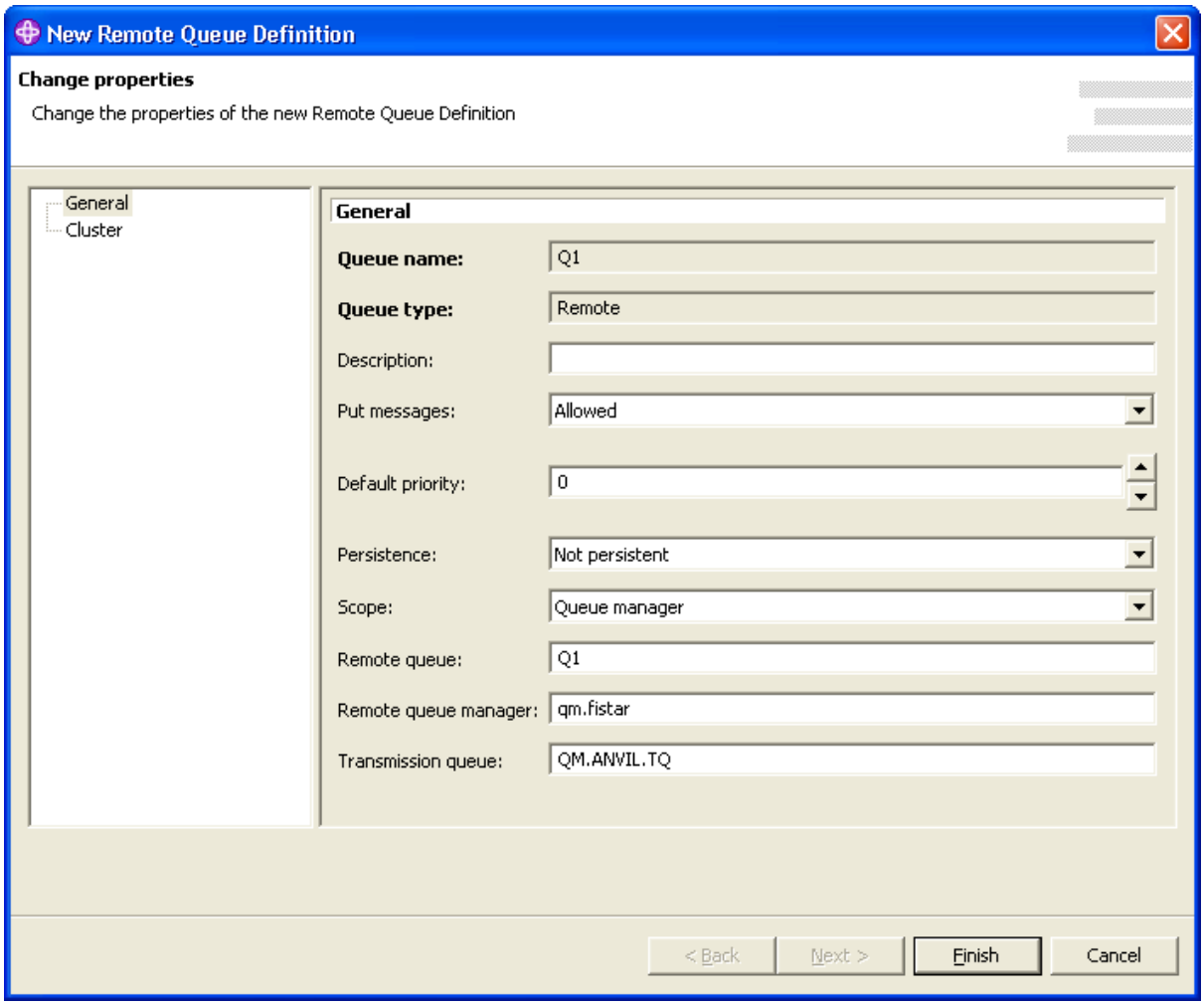
By console commands:

a) runmqsc qm.anvil
b) define qlocal(QM.ANVIL.TQ) usage(xmitq)
c) end

6) We will also need to create a remote queue definition which will represent our remote queue Q1 on qm.fistar which we want to send messages to.

By MQ Explorer:



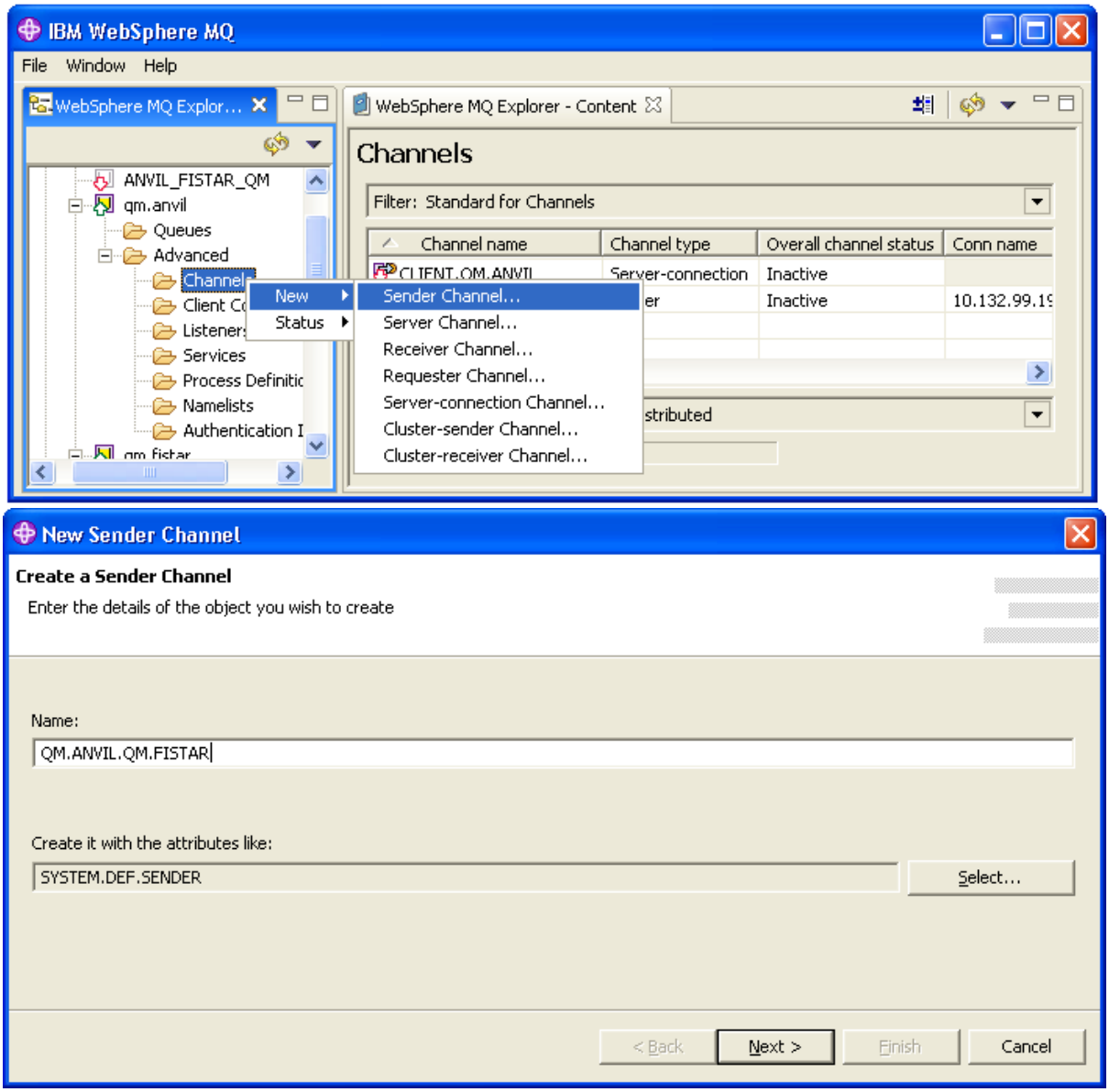


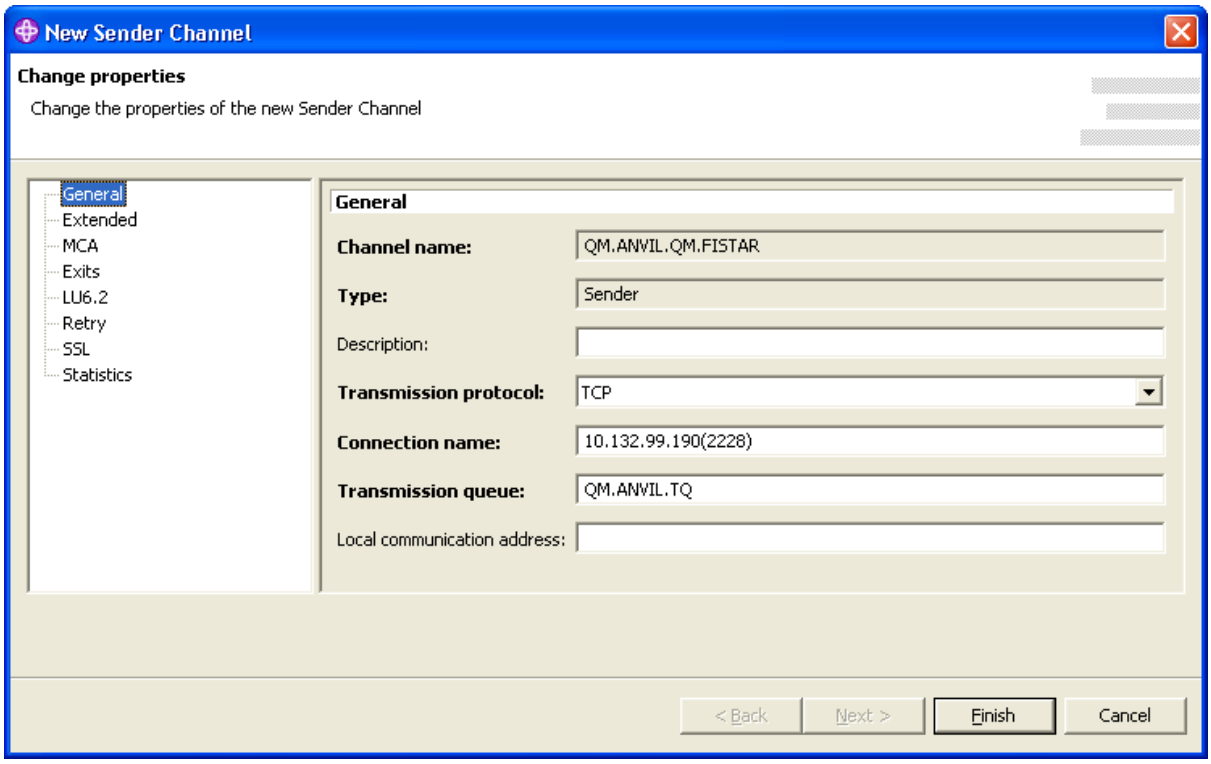
By console commands:

a) runmqsc qm.anvil
b) define qremote(Q1) rname(Q1) rqmname(qm.fistar) xmitq(QM.ANVIL.TQ)
c) end

7) We have created the transmission queue (QM.ANVIL.TQ) and remote queue definition (Q1), we still need to create the channel objects to allow queue managers to talk each other. On qm.anvil, we need to create a sender channel.

By MQExplorer:


IBM WebSphere MQ
File Window Help
WebSphere MQ Explor...
WebSphere MQ Explorer - Content
Channels
Filter: Standard for Channels
Channel name
Channel type
Overall channel status
Conn name
ANVIL_FISTAR_QM
qm.anvil
Queues
Advanced
Channel
Client C
Listener
Services
Process Definitic
Namelists
Authentication I
qm.fistar
New
Status
Sender Channel...
Server Channel...
Receiver Channel...
Requester Channel...
Server-connection Channel...
Cluster-sender Channel...
Cluster-receiver Channel...
Inactive
Inactive
10.132.99.19
New Sender Channel
Create a Sender Channel
Enter the details of the object you wish to create
Name:
QM.ANVIL.QM.FISTAR
Create it with the attributes like:
SYSTEM.DEF.SENDER
Select...
< Back
Next >
Finish
Cancel



By using console commands:

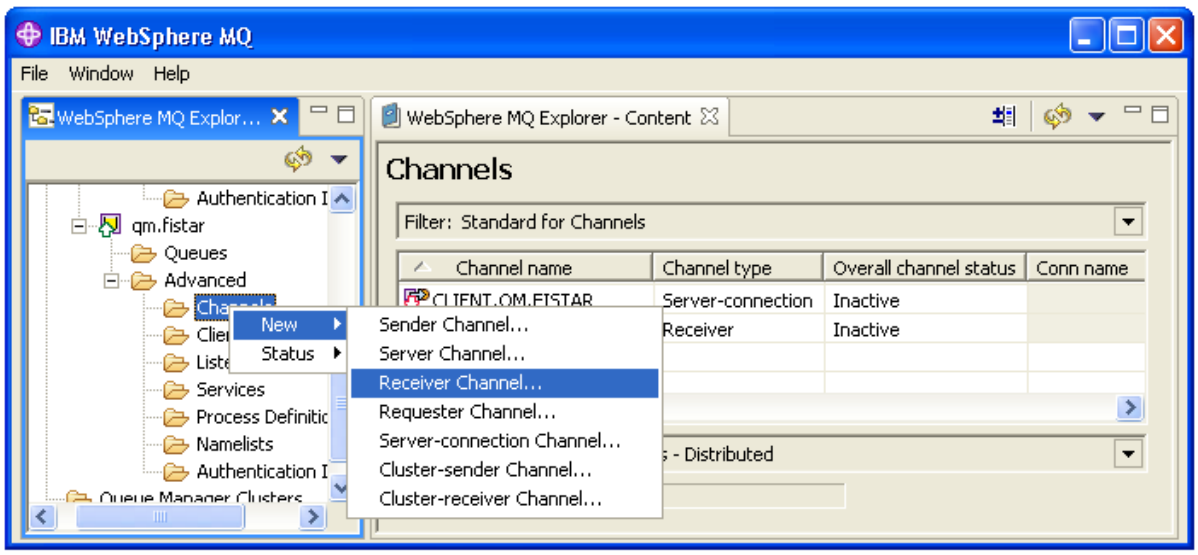
a) runmqsc qm.anvil

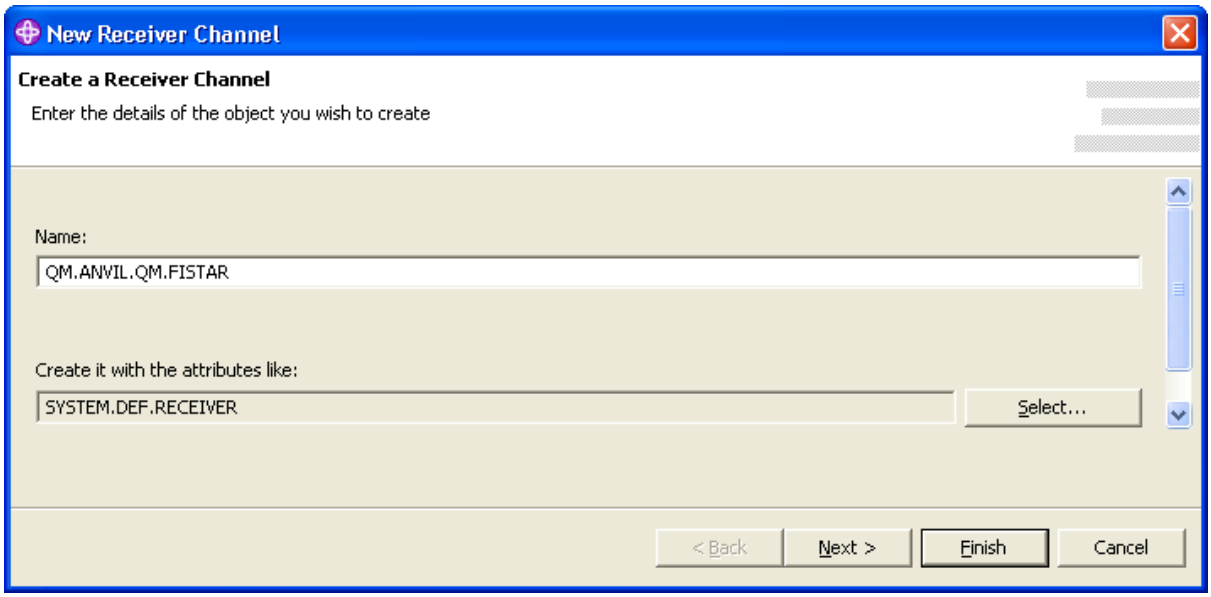
b) define channel(QM.ANVIL.QM.FISTAR) chltype(SDR) conname('10.132.99.190(2228)') xmitq(QM.ANVIL.TQ) trptype(TCP)

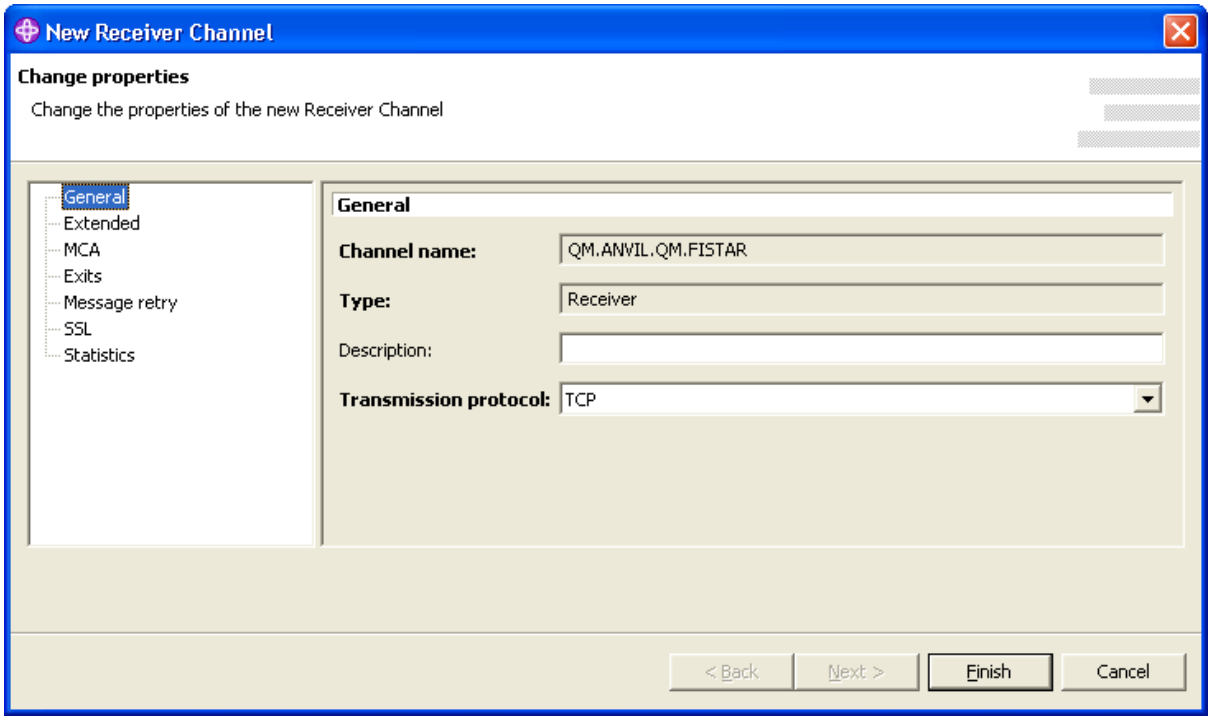
c) end

8) What we need to do now is create a receiving channel object on qm.fistar

By MQ Explorer:







By using console commands:

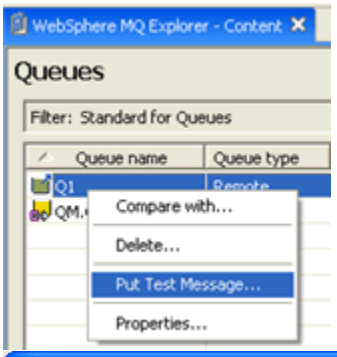
a) runmqsc qm.fistar
b) define channel(QM.ANVIL.QM.FISTAR) chltype(RCVR) trptype(TCP)
c) end

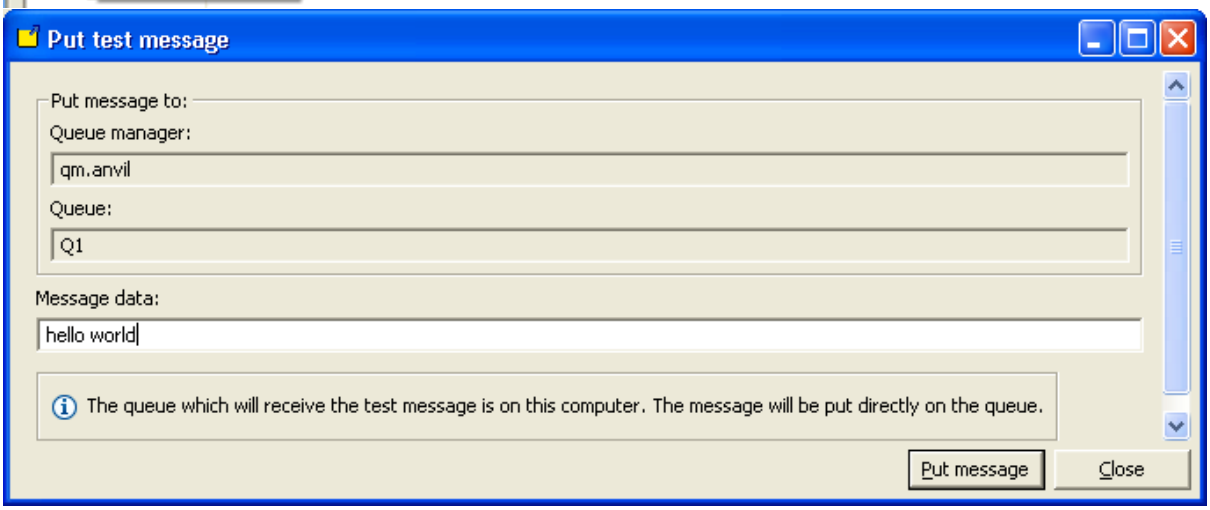
9) The idea is simple. MQ must always use a local queue to put a message. So we put a message on the remote queue definition (which is setup locally on qm.anvil) and it transfers to the channel via the local transmission queue.

10) Start the channels (QM.ANVIL.QM.FISTAR) on qm.anvil and qm.fistar

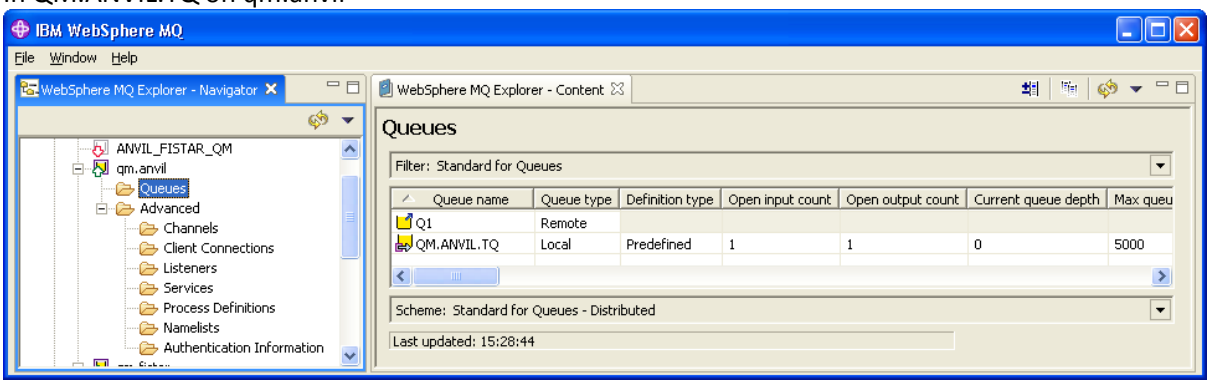
11) Lets test the message flow now using MQ explorer. What we are going to do now is put a message on the remote queue definition and see if it arrives on the remote queue manager.

a) On qm.anvil, right mouse click on the remote queue definition: Q1

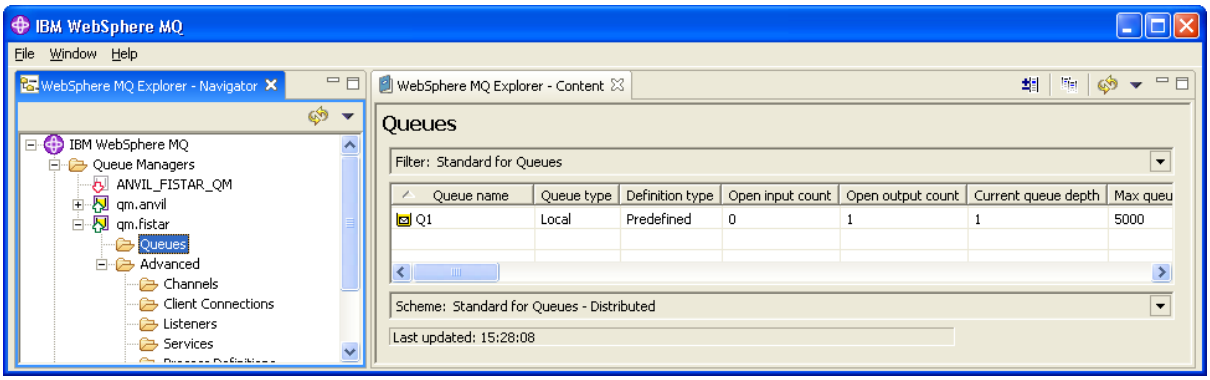




In QM.ANVIL.TQ on qm.anvil



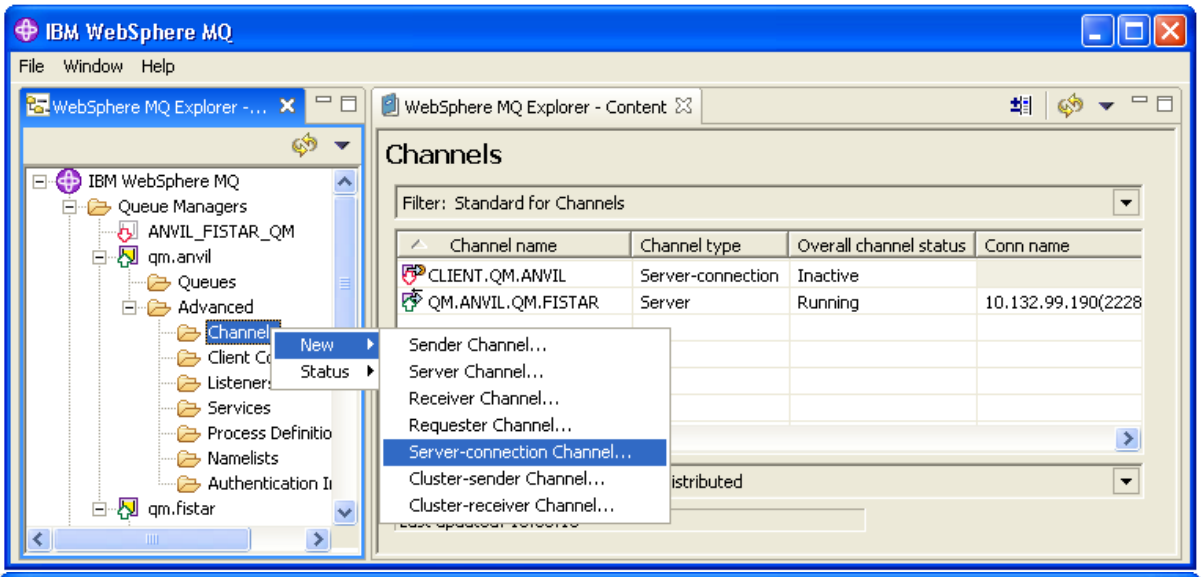
In Q1 on qm.fistar

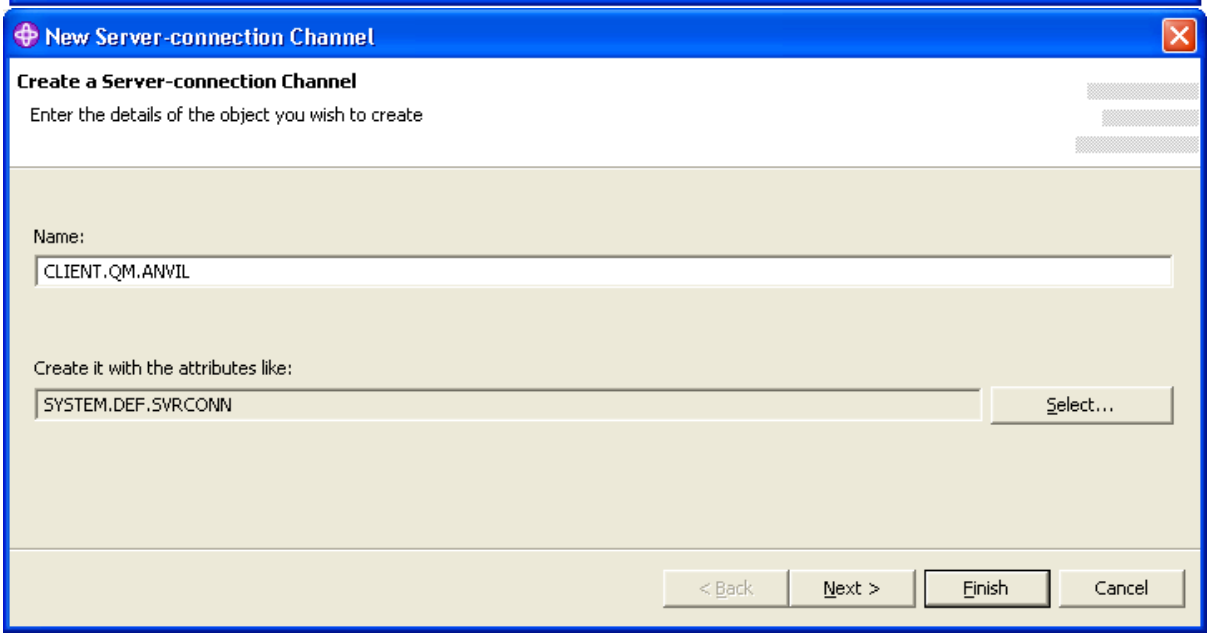


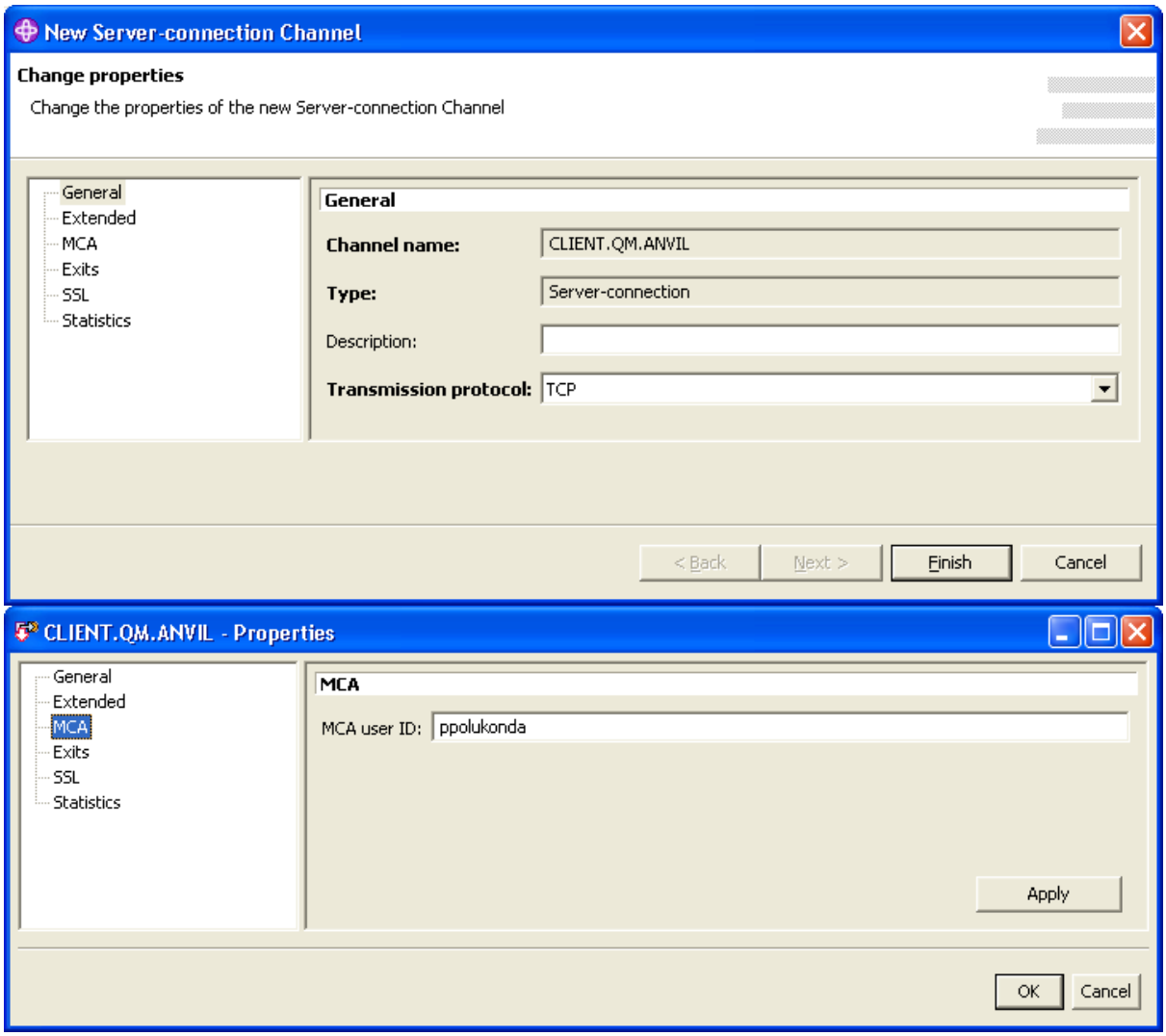
This proves that our setup is working correctly.

12) Now what we have to do is to make our client applications to connect to qm.anvil (to publish) and qm.fistar(to consume) queue managers.

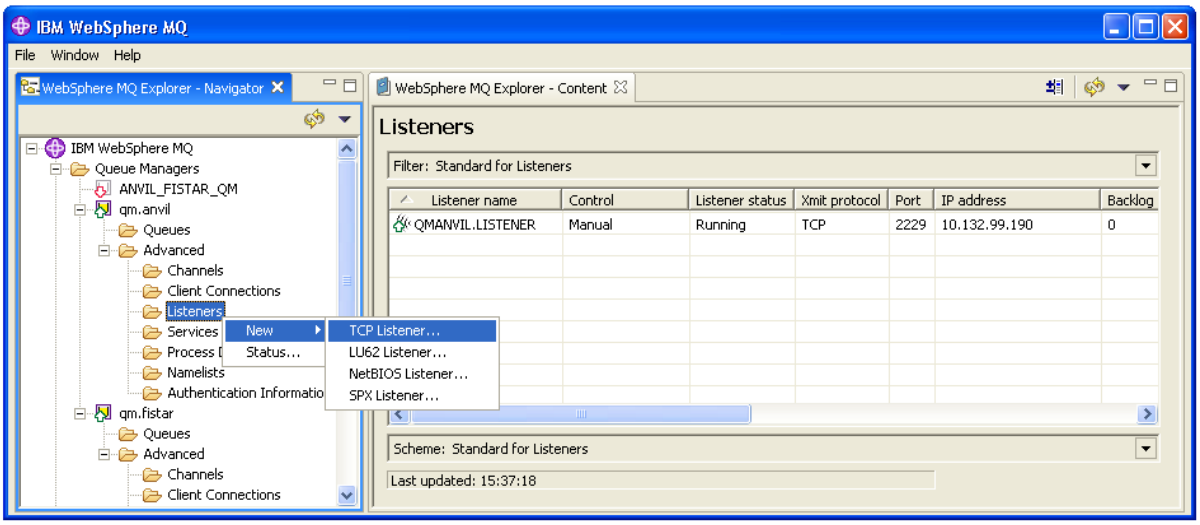
For this we need to setup server-connection channel and a corresponding TCP listener on both the queue managers:

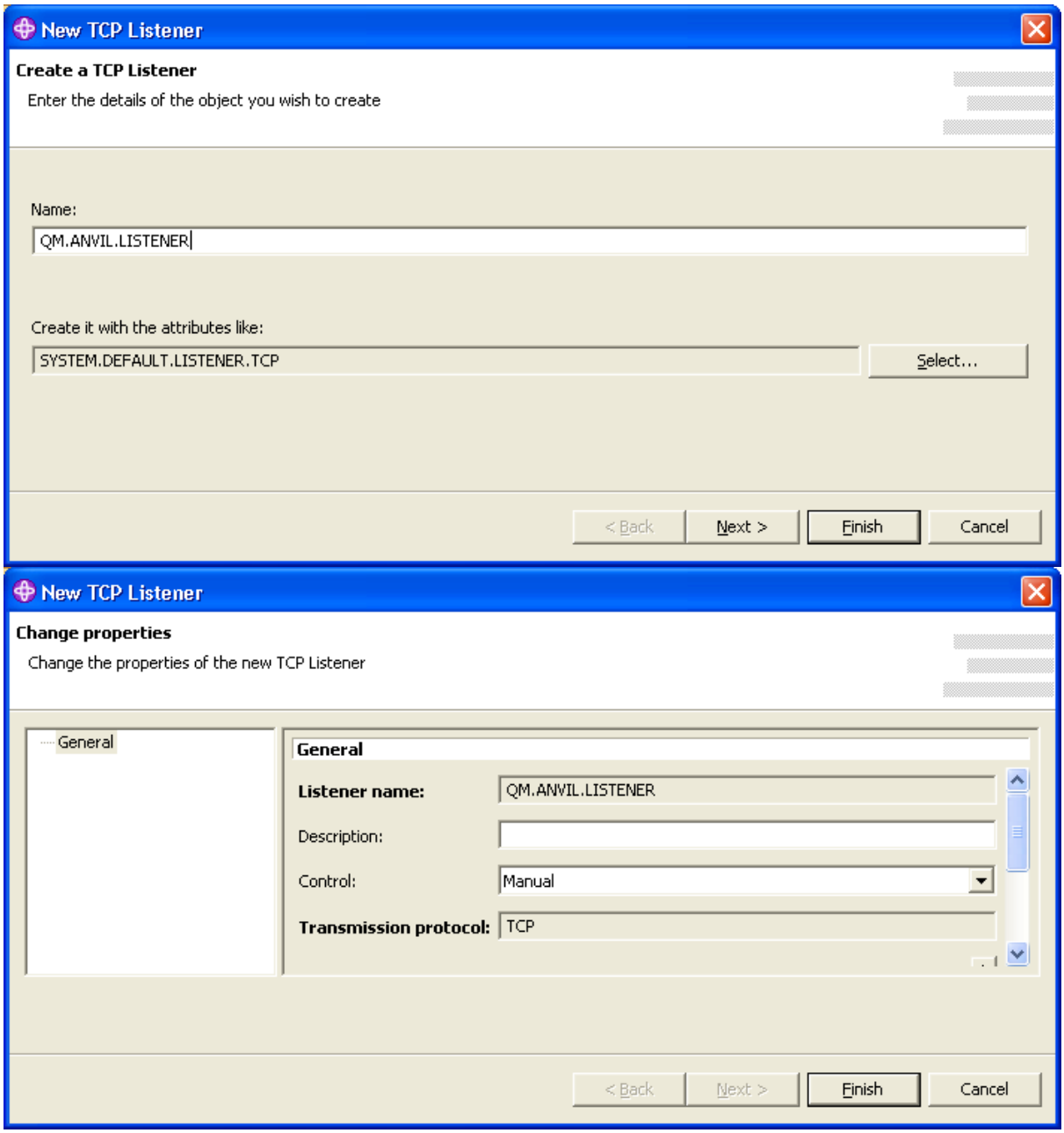


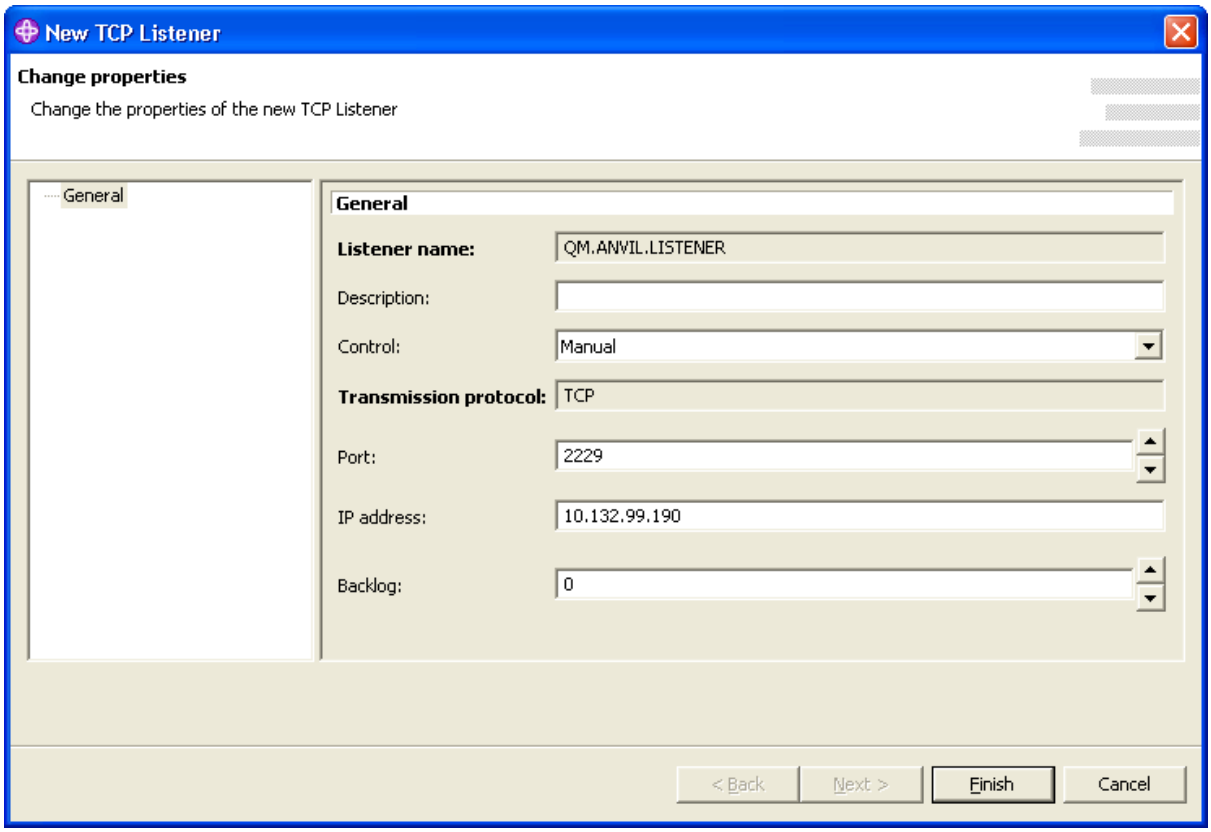




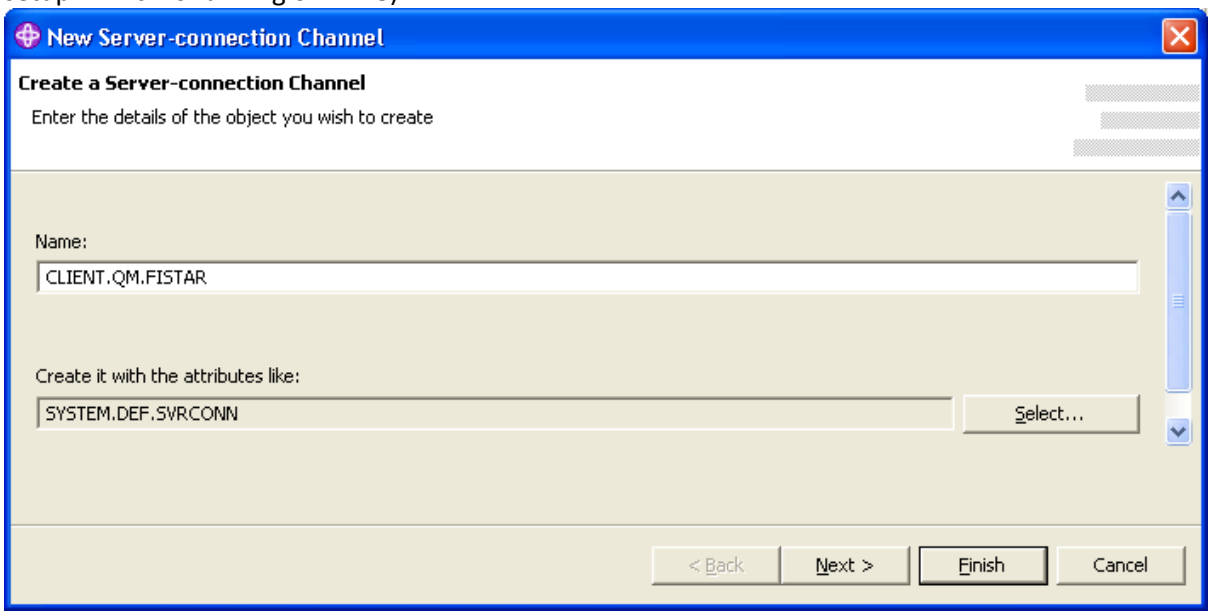
Now setup a corresponding TCP listener:

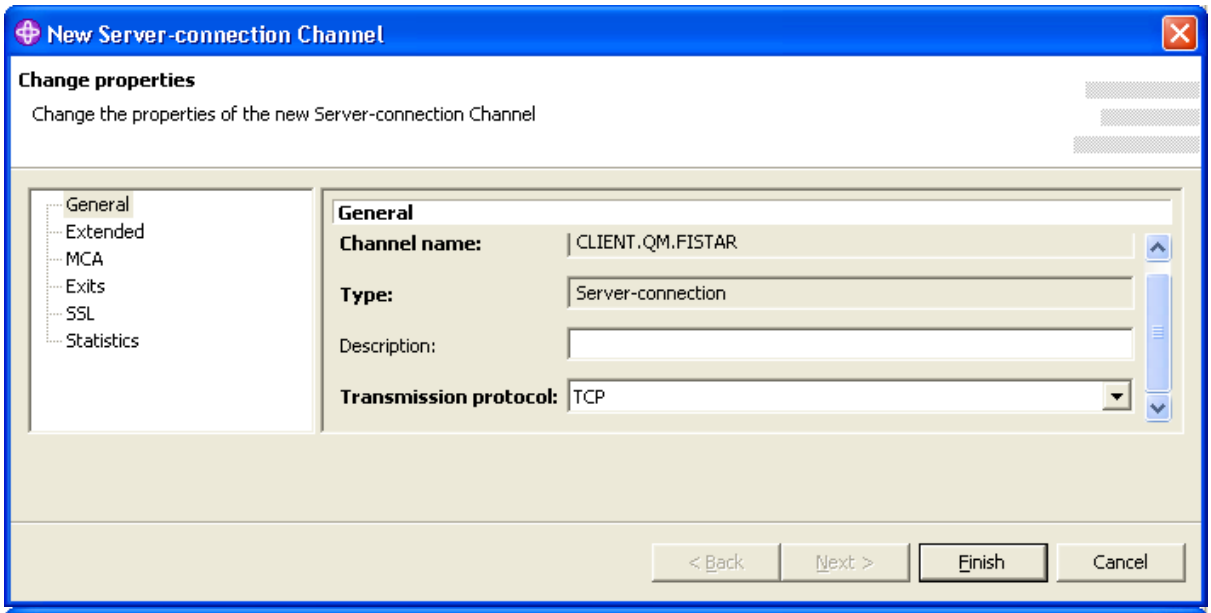


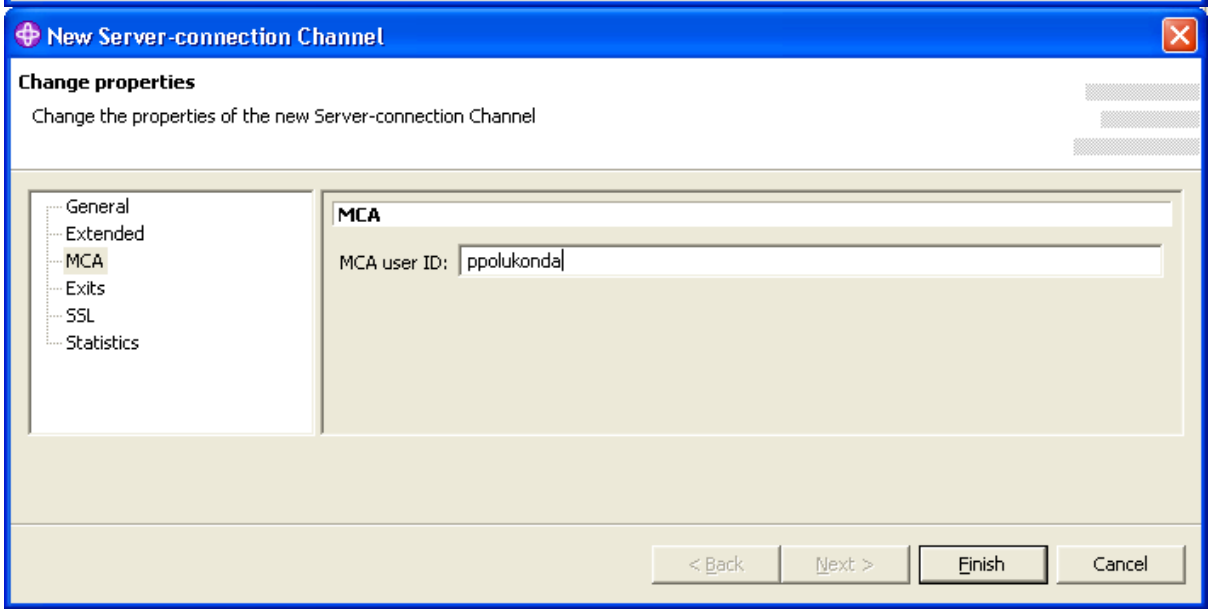

New TCP Listener
Create a TCP Listener
Enter the details of the object you wish to create
Name:
QM.ANVIL.LISTENER
Create it with the attributes like:
SYSTEM.DEFAULT.LISTENER.TCP
Select...
< Back
Next >
Finish
Cancel
New TCP Listener
Change properties
Change the properties of the new TCP Listener
General
General
Listener name:
QM.ANVIL.LISTENER
Description:
Control:
Manual
Transmission protocol:
TCP
< Back
Next >
Finish
Cancel



Follow the same step for qm.fistar (except listener because listener is already setup as part of qm setup…which is running on 2228):







4 General Information on MQ

4.1 MQ Message

- Made up of two parts: Message descriptor, Application Data
- Default MaxMsgLength is 100mb
- You can access message control information using the MQMD structure, which defines the message descriptor
- There are 4 types of messages defined by MQ: Datagram, Request, Reply, Report
- Each type of message is identified by an MQMT_* value.
- Use a datagram when you do not require a reply from the application that receives the message

- Use a request message when you want a reply from the application that receives the message
- Use a reply message when you reply to another message
- Persistent messages are written to logs and queue data files. If a queue manager is restarted after a failure, it recovers these persistent messages necessary from the logged data. The performance of your application is affected when you use persistent messages.

4.2 Queue Manager

- Each computer that hosts queues requires a queue manager. Each queue manager has a unique name, and administers the queues that have been created on it (these are known as local queues).
- For a program to send messages to a remote queue (a queue on a different queue manager), the following should be available:
  1) A message channel (connects the local and remote queue managers)
  2) A transmission queue (used to store messages if the message channel is unavailable)
  3) The queue manager puts messages onto the transmission queue, which are then sent to the associated remote queue manager
  4) A program can send messages to a remote queue without needing to know its location, if a remote queue definition has been defined locally
  5) The remote queue definition holds the target queue and transmission queue names. The local queue manager uses this information to determine where to send the message
  6) By using remote queue definitions, message channels and transmission queues, any program can send messages to any queue without needing to know where that queue is located
  7) A server connection channel must be defined on the QueueManager machine for every corresponding client channel definition.