IBM Rational Performance Tester and IBM WebSphere MQ: Performing MQ data correlation through sockets

Francisco Sambade
Test engineer, RTTS
15 Sep 2004

from The Rational Edge: You can use IBM Rational Performance Tester -- which emulates real users and verifies that an application will provide acceptable response times and functionality under load -- to implement performance testing solutions in heterogeneous environments that utilize IBM WebSphere MQ applications. This article explains how to do this, based on a hypothetical enterprise scenario.

Today's enterprise computing is rapidly converting from the client-server model to Web applications. To compete effectively, businesses need to move information between systems in real time, and typical enterprise solutions include a variety of entangled technologies that create, parse, and ultimately present processed data to the end-user. As the applications in these solutions typically come from multiple vendors, getting them to interact effectively is often difficult.

IBM WebSphere® MQ (formerly MQ Series) is a product that helps you achieve effective application interoperability. It allows you to easily exchange information across different platforms (integrating new and existing business applications), using messages and queues. By utilizing a queue manager to handle messages, it eliminates cross-platform dependencies, and program communication becomes asynchronous. This brings definite development benefits. However, it also adds complexity to writing a performance testing solution with a tool that doesn't natively support the MQ protocol.

Fortunately, you can use IBM Rational® Performance Tester -- which emulates real users and verifies that an application will provide acceptable response times and functionality under load -- to implement performance testing solutions in heterogeneous environments that utilize IBM WebSphere MQ applications. This article explains how to do this, based on a hypothetical enterprise scenario.

Before we introduce that scenario, let's start with some basics.

About performance testing

Performance testing allows you to evaluate whether a system provides acceptable response times and expected behaviors when serving a large number of concurrent clients. Performance testing tools can help you identify the bottlenecks in your system and determine how to fix the problems. To set up a performance test, you do the following:

1. Use an automated testing tool (such as IBM Rational® Robot) to record a sequence of actions between a typical user and the application. This sequence becomes a single test.
2. Starting with the test you created in Step 1, create similar tests that use different data typical of a user population. For example, for a site that allows users to purchase items, vary the number of items or the method of payment.
3. Create a schedule of user activities. For example, at any given time, you might want to test what will happen if, simultaneously, twenty users purchase a single item, ten users purchase multiple items, and seventy users search for an item.
4. Use "virtual users" to run the tests and identify performance problems.

About IBM Rational Performance Tester

IBM Rational Performance Tester validates that an application, under multi-user loads, performs as needed against specific response criteria. Solving performance problems in the test lab, when the cost of discovering and repairing defects is lowest, is essential to the success of your users and customers. IBM Rational Performance Tester contains the following tools:

- **IBM Rational Performance Tester Pack of Virtual Testers** allows you to simulate multiple users simultaneously working with your application.
- **IBM Rational TestManager** manages all aspects of performance testing across multiple platforms, configurations, and user loads.
- **IBM Rational Performance Tester Extension for SQL** provides performance testing for relational databases.

An enterprise testing scenario

Suppose you are chosen to build a suite of performance scripts for the newest application in the company, which is called ERMS, or Excellent Resource Management System. The infrastructure for ERMS is composed of a clustered set of application servers, a load balancer, a database server, and an IBM WebSphere MQ server. For the most part, ERMS talks directly to the application server through the HTTP protocol. However, some functions require data from an external feed, forcing ERMS to establish a direct connection to the MQ Manager residing in a specialized server. After recording an initial set of transactions, you notice that the scripts involved in the searches don't contain any emulation commands. You look at the list of available protocols and decide to add the Socket protocol to the mix in an attempt to capture end-to-end communication at a lower level. For this, you navigate to **Session Record Options** under the **Tools** menu in IBM Rational Robot, bring up the **Generator Filtering** tab, and move **Socket** from the **Available protocols** list into the **Selected protocols** list. After regenerating the scripts from the session, you notice `socket send` and `socket nrecv` emulation commands with associated mixed hex and ASCII data. A typical socket send call looks similar to the following:

```
sock_send
"TSH `00000050018c3000000000000000000000000111033300000000`(P`0000000"
"00000000000000007000000010000000100002800`SPQU`0000000100000"
"00c`SPQI`000000010000000c`";
```

Because MQ is not supported as a scriptable protocol, the functions connecting directly to the IBM WebSphere MQ server are being recorded as low-level socket calls. You attempt to play back the script with five users, and discover that test execution fails. Exploring TestManager's log, you notice a `«Timeout»` error message waiting for a response from the server on a given `socket nrecv` command. Previous experience leads you to believe that there is uncorrelated data, dynamic information from a server-supplied response that you need to use in subsequent requests, forcing you to analyze the suite of scripts to find out which calls you need to modify.

Solving the problem

The first step in solving this problem is to become familiar with the IBM WebSphere MQ protocol. Understanding the logical flow of events greatly reduces the time it takes to "decode" the MQ messages, since you will primarily focus on the calls that retrieve and use dynamic data. To perform successful correlation, you need to know what bytes are changing whenever a function gets called, so that you can put parameters in place to grab these bytes from the response and use them to generate a new set of requests. A common way to do this is to record the same transaction multiple times, supplying the same data, and determining where dynamic identifiers are used (session IDs, object handles, etc.), by comparing the set of requests in the newly recorded scripts. Depending on the complexity of the transaction, analyzing the network traffic can become a tedious, time-consuming process that can easily lead to script failure if you overlook minor details, such as differences in byte counts after changes are introduced.

Understanding the IBM WebSphere MQ message structure

As illustrated in Figure 1, messaging is facilitated through functions of a common API known as the Message Queue Interface (MQI). A connection needs to be established to the queue manager with the MQConn call. If this call succeeds, a unique connection handle is returned. Before an application can "put" and "get" messages to and from the queue, it first needs to issue an MQOpen call. If successful, a unique object handle is returned, which together with the connection handle, is used in all subsequent calls to place and retrieve messages to and from the queue.

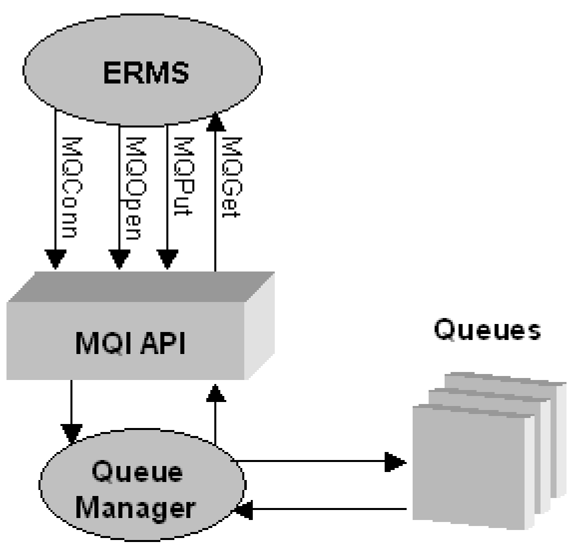

Figure 1: The IBM WebSphere MQ messaging sequence, which flows from upper left to lower right in this diagram

Here are some typical, frequently used MQI calls:

MQConn: Requests a connection to the queue manager.
MQOpen: Establishes access to objects (queues, queue managers, etc.).
MQInq: Queries/requests attributes on MQ objects.
MQSet: Changes the attributes of a queue object.
MQPut: Places a message in an opened queue.
MQPut1: Places single messages in a queue. This call does not need the queue to be opened prior to its execution. After it completes, it performs an MQClose.
MQGet: Retrieves a message from an opened queue.
MQClose: Closes a queue object.

Data correlation
An initial set of MQI calls is sent to the queue manager to establish a connection:

```
sock send   // Initial Data
"TSH `0000008201013100000000000000000000000011103330000`ID  `ff250000000000"
"3200007ffe064000003b9ac9ff`NAME NAMEN. NAMENAM  E`0003`3                  "
"`000000010000`";
```

If successful, a connection is established, and the server responds with the connection handle that needs to be supplied in subsequent requests. This does not need to be manually correlated, since the TCP stack on the operating system supplies the right connection handle automatically whenever a new request is made. However, data does need to be correlated right after the MQOpen call. The server responds with an object handle that uniquely identifies the object in use. The correlation process involves parsing the response for the object handle, and using it in subsequent MQI calls as follows:

```
// MQ Open
***************************************************************************
sock send
"TSH `000000d80183300000000000000000000000001110333000000000d8000000000000"
"000000000000`OD  `0000000100000005`                                     "
"                                                            AMQ.*       "
"                                                 `000000` ";
//*************************************************************************
```

The response to this request is stored in the read-only `_response` variable. A simple printout of its contents reveals:

```
// MQ Open_Reply
 ***************************************************************************
TSH `000000d801933000000000000000000000000011103330000000000d80000000000000000
400c3fa0'OD
 `0000000100000005`
 AMQ.*                                                           `000000`
 TSH
//*************************************************************************
```

You can easily identify the object handle through multiple function captures, while providing the same data, and observing the set of bytes that are changing among the captures. The italicized text in the previous printout represents the object handle that needs to be correlated. The first four bytes preceding the object descriptor "`OD`" form the object handle "`400c3fa0`".

The IBM Rational Performance Tester programming language is a subset of C, with powerful string parsing functions that you can use to grab this handle from the response. A sample function could be:

```
string func getMQHandle(searchMe)
string searchMe;
{

    string handle = “”;
    string temp = “”;
    string chars = “”;
    int    end = 0;

    end = strstr (searchMe,"`OD");          // Grab Object Handle
    if (end > 0)
        handle = substr(searchMe,end-8,8);  // Get the previous 8 chars (4 bytes)
    else // There are embedded characters between “’” and “OD”
    {
        temp = substr(searchMe, 6, strlen(searchMe)); // Remove the Header (TSH ')
        end = strstr (temp,"OD"); // Find index for OD
        if (end > 0)
        {
            posQuote = strstr(temp, "`");   // Find index for single quote
            counter = end-(posQuote+1);     // Number of characters to convert to Hex
            grab = 8 - (counter*2);         // Number of characters to grab

            handle = substr(temp,posQuote-grab,grab);  // Get the previous chars
            chars = substr(temp,posQuote+1,counter);
            chars = mixed2hexstring(chars);            // convert to hex
            chars = substr(chars,2,strlen(chars)-2);   // Remove "`" from the string
            handle = handle+chars;
        }
        else
        {
            printf("ERROR Retrieving MQ Handle");
            script exit("Error Retrieving MQ Handle at "+tod());
        }
    }
    return handle;
}
```

Once you obtain the handle, you can insert it back into the recorded calls as follows:

```
// MQ INQ
**************************************************************************//
sock send
"TSH `0000003c01893000000000000000000000000111033300000000006d000000000000"
"0000" + objHandle + "00000001000000000000030000007df`";
```

Similarly, the server also returns the MQ Object name, which needs to be provided in an MQOpen call in order to successfully establish access to a given object.

```

// Response containing the dynamic object name
// MQOpen Reply Obj = AMQ.40CC53750001E462
"TSH `000000d801933000000000000000000000000111033300000000000d8000000000000000"
"0400c4010`OD  `0000000100000001`AMQ.40CC53750001E462                       "
"NAME NAMENAMEN NA
" " AMQ.*"

// Modified MQOpen call with the previously captured object name
sock send
    "TSH `000000d801833000000000000000000000000111033300000000000d8000000000000"
        "000000000000`OD  `0000000100000001`"+objName+"                 "
            "                                                          AMQ.*"
                "                                                `000000`)";
```

Note that this response also contains an object handle, which is used in subsequent MQInq calls to identify the opened object. Issuing an MQClose MQI call follows the same process. After a connection has been opened and an object handle is retrieved, the handle for that object needs to be supplied to MQClose in order to terminate access to it.

This is the basic process for MQ correlation. However, you should consider other things when making changes in the scripts. The byte count for a given call needs to be accommodated if different size parameters are introduced. Suppose you have an MQPut message with dynamic body contents. For example, the message might contain a search term that needs to change during runtime in order to measure how ERMS would perform when concurrent users are executing *different* searches. You'll notice the search term embedded in a socket call that was captured during initial recording. If this term changes to a bigger or smaller size, so does the byte count for the message. Depending on its structure, this count might be specified in one or multiple places (including a byte count associated with the search term in question). A new byte count needs to be calculated based on the initial count +/- the bytes introduced by this term, and then inserted into the call.

Subsequently, not all search terms will return the same number of results, so the byte count for the Socket *nrecv* also needs to be correlated. The trick is to determine how many bytes the server will return for a specified call, since the only place this information is displayed is in the response itself. If you were to perform a `sockect_nrecv` for "n" bytes, and the response contained « n bytes, then command execution would time out waiting to retrieve the remaining bytes. In contrast, if you were retrieving fewer bytes than the number returned by the server, the remaining bytes would carry over to the next Socket `nrecv` emulation command and eventually cause synchronization problems. To resolve this issue and receive the correct number of bytes every time, you need to parse the byte length from the response itself. Because the response contains the byte length in the four bytes following the "`TSH`" `structid` (notice the space after "`TSH`"), you can grab the first eight bytes from the response, ignore "`TSH`", and set the remaining bytes, which represent the byte count, into a variable. Because this length is represented in hex, it needs to be converted to decimal first, so that the remainder byte length can be supplied to a new `sock_nrecv` command as follows:

```

...
{
string totalBytes = "";
int    totalBytesDec = 0;
int    remainingBytes = 0;
}
sock nrecv 4;                              // Bypass the first 4 bytes "TSH "
sock nrecv 4;                              // Get 4 byte length "`000000d8`"
totalBytes = mixed2hexstring( response); // Convert to hex is needed
totalBytesDec = hexToDec(totalBytes);    // Convert a hex string to decimal (Custom function)
remainingBytes = totalBytesDec-8;        // Calculate remaining Bytes
sock nrecv remainingBytes;                  // Get the rest of the response
...
```

Final note

The techniques shown in this article allow for manual MQ data correlation through the Socket protocol, as captured by IBM Rational's Performance Tester. For further MQ analysis, and an easier understanding of MQI messages, you can use third-party packet sniffers/protocol analyzers to identify key attributes in the application. For instance, Ethereal, an open-source protocol analyzer can capture and decipher recorded MQ packets, identifying the type of request sent to the server, the location of object handles, packet segment lengths, or even the name for the Queue Manager in use.

References

IBM WebSphere MQ Call Routines:
http://support.sas.com/rnd/itech/doc9/dev_guide/messageq/ibmmq/mqfuncs.html

IBM WebSphere MQ Application Message Interface:
http://publibfp.boulder.ibm.com/epubs/html/amtyak08/amtyak08tfrm.htm

IBM WebSphere MQ Application Programming Guide:
http://publibfp.boulder.ibm.com/epubs/html/csqzal09/csqzal09tfrm.htm

IBM WebSphere MQ Application Programming Reference:
http://publibfp.boulder.ibm.com/epubs/html/csqzak09/csqzak09tfrm.htm

IBM WebSphere MQ Messages:
http://publibfp.boulder.ibm.com/epubs/html/amqzao04/amqzao04tfrm.htm

About RTTS

RTTS (www.rttsweb.com) is a professional services organization that specializes in the testing of IT applications and architectures. Using best-of breed products, expert test engineers, and proven methodology to provide the foremost end-to-end solution, RTTS helps organizations successfully implement quality business applications. An IBM Premier Business Partner, RTTS offers full outsourcing of testing needs, individual test tool expertise, expert mentoring and education services, and a proven plan for providing knowledge and skills transfer.

About the author

Francisco Sambade, a test engineer with RTTS, has executed performance and functional testing for major corporations. His experience has exposed him to unique testing obstacles that led him to develop the approach described in this article. A specialist with IBM Rational, Mercury Interactive, and Compuware performance and functional testing tools, he also holds CompTIA A+ and CompTIA Network+ certifications and a computer science degree from the State University of New York at Stony Brook.