

# Extending the Message Flow Debugger for MQSI

Shuxia Tan, Eshrat Arjomandi, Richard Paige  
Department of Computer Science  
York University,  
4700 Keele St.  
Toronto, Ontario, M3J 1P3

Evan Mamas, Simon Moser, Bill O'Farrell  
IBM Toronto Laboratory  
8200 Warden Ave.  
Toronto, Ontario, M3C 1H7

## Abstract

*Integration and management of applications play key roles in today's computing. MQSeries is an asynchronous, assured application-to-application communication protocol designed to support the integration of business processes. MQSeries Integrator (MQSI) is a component of MQSeries providing support for application integration and communication. The key technology in MQSI is the notion of a message flow. A message flow is a sequence of operations on a message, performed by a series of message processing nodes. A message flow developer creates a message flow by including processing nodes and connectors between them. Errors in the development of large and complex message flows are unavoidable, hence the need for a message flow debugger.*

*A prototype debugger for message flows was developed at IBM Research Lab in Haifa. In this paper, techniques are presented that extend the prototype debugger to deal with multithreaded and nested message flows. In addition, we discuss how the performance of the debugger can be improved to meet production standards.*

## 1 Introduction

Modern applications need to cooperate in today's business processes. Common mechanisms that are used for application communication include remote procedure call (RPC), distributed object systems such as DCOM, CORBA and Enterprise Java Beans, and message queuing technology in products such as IBM's MQSeries.

MQSeries Integrator (MQSI) is a product in the MQSeries family. It provides a powerful message broker to support message

delivery, message routing and database integration. It can help to reduce costs, increase efficiency and enable more dynamic business process integration [1]. It has been widely used in financial business fields such as banking and insurance.

The technology supporting message queuing in MQSI is the notion of a *message flow*. A message flow is a sequence of operations on a message, performed by a series of message processing nodes. It can be depicted as a visual program showing the processing logic as a directed graph. A message flow developer creates a message flow by including processing nodes and connectors between them. Errors in user-developed message flows are unavoidable. Therefore a message flow debugger is essential.

A prototype debugger for message flows was developed at IBM Research Lab in Haifa, Israel. We refer to it as the **Haifa debugger**. The Haifa debugger provides a prototype design of a message flow debugger, including much of the functionality needed in debuggers, such as set breakpoints, step in, step over, and displaying message contents. However, it has limitations in debugging *multithreaded* message flows. A nested message flow provides a way to reuse message flows, analogous to function calls in programming languages. Without nested message flows, it can be very difficult to understand large message flows developed by users. Multithreaded applications are now ubiquitous and therefore must also be supported by a modern debugger.

In addition, the performance of the Haifa debugger is not acceptable for production. For large and complicated message flows with hundreds of processing nodes and multiple input and output queues, the

performance of the Haifa debugger becomes unacceptable. We have extended the Haifa debugger to a powerful and efficient debugger for all kinds of message flows. This paper provides a brief overview of extensions made to the Haifa debugger. A tool is not useful unless its perceived performance by users is acceptable. In this paper we focus on evaluating the performance of the Haifa debugger and our extensions to it.

Section 2 presents an overview of background work. Section 3 describes our extensions to the Haifa debugger. Section 4 contains performance tests for the Haifa debugger and the extended debugger. The last section, Section 5, presents concluding remarks.

## 2 Background

The introduction of enabling technologies is helpful for understanding the discussion of our approach to extending the debugger. In this section, we will introduce the basic concepts and the Haifa debugger.

### 2.1 Messages in MQSeries

MQSeries is an IBM middleware product based on the basic principles of message queuing [2]. An MQSeries message is simply a collection of data or the unit of transfer that is sent from one program to another. The message consists of two parts: control information that we call *message header* and application specific data that we call *message body*. The message header contains a message ID that identifies the message, as well as other control information such as the message format, message type, correction ID, and priority.

### 2.2 MQSeries Integrator

MQSeries Integrator (MQSI) is a powerful message broker software system, designed to help enterprises in application integration. It is also a tool kit that makes it easy to pass messages between different software applications. As a message broker software, MQSI knows how and where to deliver its messages (message routing), it can link applications by transferring the data to them

(message brokering), and it can take over the task of how to deal with each program or platform (application bridging) [3]. We briefly describe the architecture of MQSI and its components.

### MQSI's System Architecture

MQSI's system architecture is shown in Figure 1.

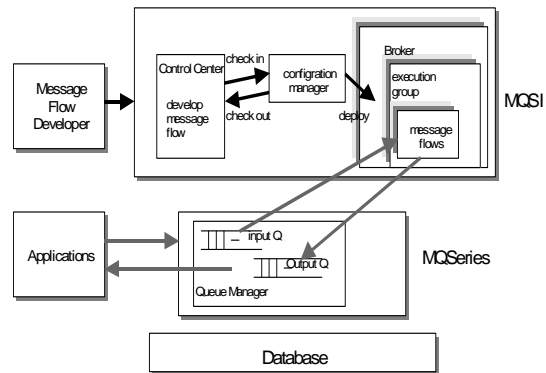


Figure 1 MQSeries Integrator System Architecture

The control center is a tool that is used to develop, modify, assign and deploy message flows. The configuration manager stores and manages the configuration data (message flows, message format definitions, etc). The broker controls the business processes and execution of message flows. The execution group is the 'container' process in which message flows are instantiated. A message flow is running when it is assigned to an execution group within a broker. Deployment assigns the configuration definitions of a message flow to a broker in order to make it run as part of the runtime system.

A user can develop a message flow by using MQSI's control center. The message flow starts running after it is checked in and deployed to the configuration manager, which will forward the message flow to the broker. As soon as a message gets into the input queues that are held by the queue manager, the message flow starts to execute actions on the message.

### Multithreaded Message Flows

In MQSI, each execution group has a thread pool. A thread takes a message from an input queue, sends it through the message flow,

and puts the message into the output queue, at which time the thread goes back to the thread pool. A message flow with more than one input node has more than one thread running simultaneously.

### Nested Message Flows

Programs written in traditional languages such as C, C++ or Java, commonly contain many function calls and procedure calls. For a message flow, users can embed one message flow within another, enabling them to reuse a particular sequence of nodes that provide a commonly required function. We call such flows *nested message flows*. An example of a nested message flow is shown in Figure 2. In this Figure, *nested\_main\_flow* has a call to *nested\_1\_flow* and *nested\_1\_flow* has a call to *nested\_2\_flow*.

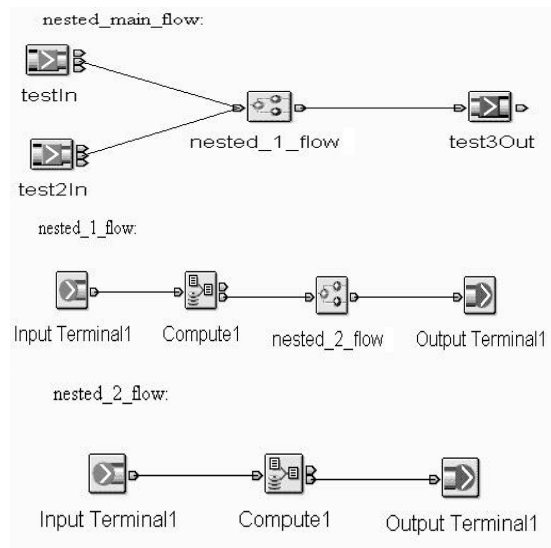


Figure 2 A Nested Message Flow Example

### 2.3 The Haifa Debugger

The Haifa debugger enables the user to inspect message content and the execution scenario by tracking a message flow while messages pass through it. The main idea behind the Haifa debugger is to leave the runtime environment unchanged and introduce special “debug instructions” into the message flow being tested, and these special “debug instructions” are implemented by *debug plug-in nodes*. A debug plug-in node is a special processing node added

between every two processing nodes in a message flow to form an *instrumented* message flow [4]. The function of the debug plug-in node is to communicate with a debug monitor that is part of the control center, passing information about the execution status, execution position, and the contents of the message, as well as propagating the new message flow to the next processing node.

Each debug plug-in node communicates with the *debug controller*. The controller’s purpose is to control the debug action and get the message content and execution information from the runtime (debug plug-in node) each time a message passes through it. Therefore, the user is able to set or delete various kinds of breakpoints, as well as carry out typical debugging functions.

### 2.4 Related Work

Like debuggers for general languages [6][7], message flow debuggers have the ability to set breakpoints, display message content (as compared with display state values in program debuggers), display execution stack contents, and debug multiple threads.

A system related to MQSeries Integrator is NEON e-Biz Integrator [5]. It does not provide a convenient tool to debug message flows. To our knowledge, there are no academic or industry groups working on the development of message flow debuggers. The Haifa debugger for message flows is the first debugger prototype for message flow systems.

### 3. Extensions to the Haifa Debugger

We have extended the Haifa debugger to support multithreaded message flows and nested message flows efficiently. Our extended debugger provides two modes: blocking and non-blocking. Blocking mode itself supports **dynamic**, **static** and **enhanced** modes. In terms of functionality, the relationship among these modes can be described by the UML (Unified Modeling Language) class diagram shown in Figure 3.

The **dynamic blocking mode** is our first extension: it inherits all the functionality of the Haifa debugger, and adds new features,

particularly the ability to debug nested message flows completely. However, dynamic blocking mode is only an intermediate product, it is not intended to be used by users due to its poor performance. The **static blocking mode** has all the functionality of the dynamic blocking mode except that it does not allow breakpoints to be set on the fly and provides an execution stack for nested message flows to be displayed. In addition it produces a substantial improvement in performance. The **enhanced blocking mode** solves the problems that exist in the static blocking mode without sacrificing the performance of the static blocking mode. Finally, the **non-blocking mode** has all the functionality of other modes, but it provides true multithreading and superior performance. The enhanced blocking mode and the non-blocking mode are products that can be used by end-users.

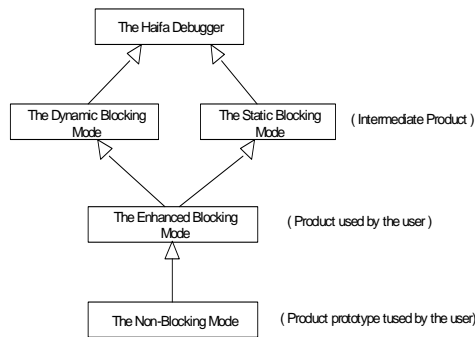


Figure 3 The Relationship of All Modes of the Message Flow Debugger

### 3.1 Dynamic Blocking Mode

In the Haifa debugger, multiple threads can cause a debug session execution to end abnormally. For example, as soon as the first thread finishes the execution in the last debug node, the execution engine will send a “debug session end” message to the debug controller, which will cause the termination of debugging in the controller and in the GUI.

Dynamic blocking mode addresses this problem; the solution will also work with multiple threads. We use **Mutex** objects in debug plug-in nodes to block threads until the first thread goes through the whole message

flow and return to the first debug node. From the view of the whole message flow, the propagating algorithms in debug plug-in nodes and in the processing nodes call each other and composed of a recursive call chain. Each debug node needs to communicate with the debug controller both in calls and in return. We refer to the socket operation in calls as the forward socket, and socket operation in returns as the backward socket. The forward socket is used to ask the debug controller if the debug node has a breakpoint, whereas the backward socket is used to ask if the debug session has ended.

Another extension in this mode is to make the debug plug-in node wise. By using the variable “recursive level”, the debug plug-in node knows when a chain of recursive calls is back at the first call. At this point, the “debug session ended” information is sent to the controller when the recursive call is finished. Thus, when the recursive calls return, the socket operation is executed only once instead of once per processing node.

### 3.2 Static Blocking Mode

Although in the dynamic blocking mode we can debug multithreaded message flows, end a debug session normally, and reduce the cost of open/close socket operations, there are still limitations to its performance. Static blocking mode has therefore been designed to improve the performance.

The purpose of the communication between the debug node and the debug controller is to send a message to the debug controller when the debug node has a breakpoint set up by the user. In the Haifa debugger and the dynamic blocking mode debugger, each debug node has to communicate with the debug controller in the forward (call) direction of a “recursive” call, whether or not it has a breakpoint. This inefficiency is negligible in the case of small message flows, but becomes much more significant when a message flow is large and has just a few breakpoints.

The key point in the design of static blocking mode is to make the debug plug-in node smarter and avoid unnecessary communications. We present this design by first comparing the dynamic and static

blocking modes. In addition, the methods and existing problems with the static blocking mode are discussed.

#### ***The difference between the static and dynamic modes***

The main difference between the static and dynamic modes is that the debugger in the static blocking mode is smarter than the dynamic blocking mode. In the dynamic blocking mode, the execution engine doesn't know which debug nodes have a breakpoint: the debug controller owns the user's breakpoint table. Therefore, when a thread encounters a debug node, it has to communicate with the debug controller to check whether or not the debug node has a breakpoint. In the static blocking mode, the user's breakpoint table is moved from the debug controller to the execution engine. In this way, communication is performed only by those debug nodes that have breakpoints.

#### ***The Breakpoint Attribute***

Now that the general design has been explained, the details of how to move the user's breakpoint table to the execution engine can be discussed. All processing nodes, including the debug plug-in nodes, have attributes such as a node's UUID. These attributes are set up when the processing node is created during the message flow development. The attributes are sent to the broker when the message flow is deployed. Therefore at runtime, the broker knows the UUID of each processing node.

In order to have the broker know at runtime whether a debug node has a breakpoint, a breakpoint attribute is added to the node. If the user sets up a breakpoint in this debug node, its breakpoint attribute is set to true, otherwise, it is set to false. After the message flow has been deployed, the broker will know the breakpoint attribute value of each debug node at runtime. Therefore, in effect, we can say that the broker holds the user's breakpoint table.

#### ***Problems with Static Blocking Mode***

Although we made many improvements in the static blocking mode, there are still a few problems as follows, which are addressed in enhanced blocking and non-blocking modes.

1. As with the dynamic blocking mode debugger, the static blocking mode debugger is not a truly multithreaded debugger because only one thread is executed at a time.
2. For nested message flows, the execution stack cannot display the path of the nested message flow. This is because only UUIDs of debug nodes that have breakpoints are sent to the debug controller. These UUIDs did not provide the capability for the debug controller of getting all message flow names that the message has passed through, considering not all paths in a message flow have breakpoints.
3. The user cannot set a breakpoint on the fly. After the message flow is deployed, the value of the breakpoint attribute for each node has been sent to the broker, and users' changes to breakpoint settings after the message flow is deployed cannot be passed to the broker.

### **3.3 Enhanced blocking mode**

Since the static blocking mode has several limitations, it is considered as an intermediate product. However, performance tests (Section 4) show that the static blocking mode has good performance. Enhanced blocking mode addresses some of the limitations of static blocking mode. It uses a *breakpoint* attribute in the debug plug-in node and keeps the user's breakpoint lists in the execution engine. Enhanced blocking mode requires solving the problems listed in Section 3.2 without affecting the performance of static blocking mode. We use techniques such as **thread local storage**, **run-time stack** and **daemon threads** to solve these problems. Detailed descriptions can be found in [8].

### **3.3 Non-Blocking Mode**

Although the dynamic, static and enhanced blocking mode debuggers can be used to debug many message flows, there are still a few problems with the enhanced blocking mode.

1. **The performance issue.** For small message flows, a blocking mode is a good choice, since it does not require much time for a message to be transferred from an input queue to an output queue. However, for large message flows with two or more input nodes, or with complicated paths from input nodes to output nodes, and only two or three breakpoints, it is possible that some paths from the input node to the output node do not meet any breakpoints. The messages passing through such paths are supposed to be transferred at once. However, in the blocking mode, the threads run one at a time, therefore message flows described above are not handled efficiently in the presence of many messages.
2. **Time-sensitive message flows.** In a blocking mode, the debugger can deal with most message flows. However, time-sensitive message flows will not be debugged correctly; the Haifa debugger also has this problem. For example, consider Figure 5, which describes a message flow with a database node. The

first thread from MQInput1 will deposit data into the database, and the second thread from MQInput2 will retrieve the data from the database after MQInput1's deposit. When the user debugs the message flow in blocking mode, it is impossible to know which thread gets to the database node first, so it is not possible to detect timing bugs.

In view of the problems above, we developed and implemented the non-blocking mode, in which all threads are unblocked to run at the same time. All threads connect to the debug controller through sockets at the same time.

The non-blocking mode debugger inherits all functionality from the static debug mode, hence debug nodes have a breakpoint attribute, and the debug controller does not possess the user's breakpoint table any more. We now briefly describe how all threads can connect to the debug controller simultaneously, how the user can monitor the threads from the screen, and how the user can control the execution of the threads.

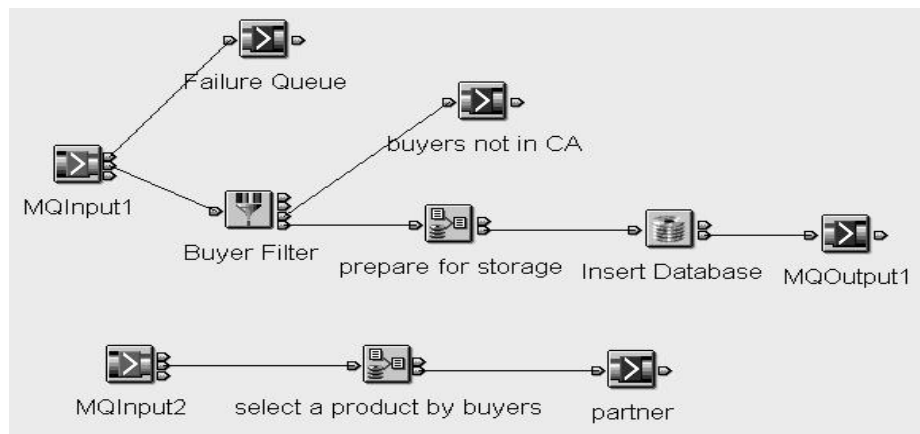


Figure 5 A Message Flow with a Database Node

### ***The Multithreaded Socket Server in the Debug Controller***

In non-blocking mode, all threads are running, therefore, many threads may need to connect to the debug controller through sockets simultaneously. In the Haifa debugger, the socket client in the execution

engine tries to connect to the debug controller. The socket server is in the debug controller. In the design of the non-blocking mode, we designed a multithreaded socket server (shown in Figure 6) to communicate with socket clients. By using the multithreaded socket server in the debug controller, all executing threads can send

messages to the debug controller simultaneously instead of sending messages one thread at a time. Since socket operations are expensive, the multithreaded socket server can reduce the expense, thereby improving performance.

As soon as a thread in the execution engine tries to connect with the debug controller, the socket server in the debug controller will pick up a socket thread from the thread pool to build up the connection with the execution threads.

There are two objects used to transfer data between the socket server and socket clients. One is the *SocketServer* class, which is responsible for building up the socket connection and creating a thread. The other is the *SocketThread* class, which will receive the data sent by a thread in a debug plug-in node through the socket connection.

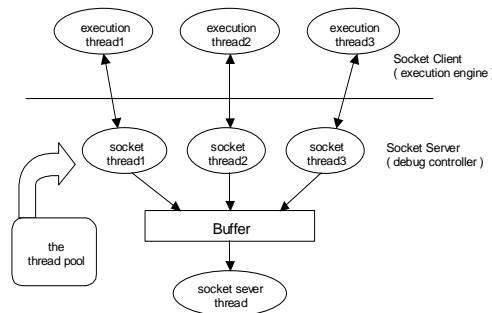


Figure 6 The Multithreaded Socket Server

The debug controller with the socket server and the GUI are developed in Java. The GUI code is not thread safe: only one thread is in the GUI side at all times, so all socket threads cannot communicate with the GUI thread at the same time. A buffer, as shown in Figure 6, is used as the communicating bridge. Every socket thread puts data into the buffer and waits for the return data. The SocketServer thread will send the buffer to the GUI. However, there is a problem: not all execution threads connect with the debug controller at the same time, therefore this raises the issue of when the buffer data should be sent to the GUI.

One solution to this problem is to send the buffer content to the GUI as soon as it gets to the buffer. Alternatively a special thread can

be used to monitor the buffer, and after a time unit, the data in the buffer can be sent to the GUI by this special thread. In our implementation, we use the first the solution.

### User Interface

In non-blocking mode, the debug controller can obtain data sent from more than one thread. So, there is a user interface problem with respect to how to display all the messages and all the threads' information without confusing the user. A user interface, as shown in Figure 7, has been designed. A new window named *ThreadListView* is used to display information about the threads that have sent messages to the debug controller. In the *ThreadListView* window of Figure 7, there is information about two threads that have sent messages to the debug controller. When the user clicks "thread from test2In" (test2In is the input queue associated with MQInput1 node), the *Execution Stack* window will display the name of the message flow that thread has passed, the *Message Contents* area will display the message header and message contents that the thread sends, and the *Message Flow* area will show an icon to indicate where the execution of the flow stops. By clicking the items in the *ThreadListView* window, the information in other three windows/areas is updated.

### User's Control of the Threads

One of the features of the debugger for multithreaded applications is that the user can control the execution of the threads. The user can choose one thread to run in order to observe this thread; choose one thread to run until completion if they are not interested in this thread; choose all threads to run one step in order to observe the execution of the threads; or choose all threads to run until completion if they do not want to debug any threads.

When execution of the flow stops by meeting a breakpoint, the user can choose one thread in order to display message contents, execution stack, or to change message contents in the GUI. Then, the user can choose one of the above options to continue the execution of the message flow.

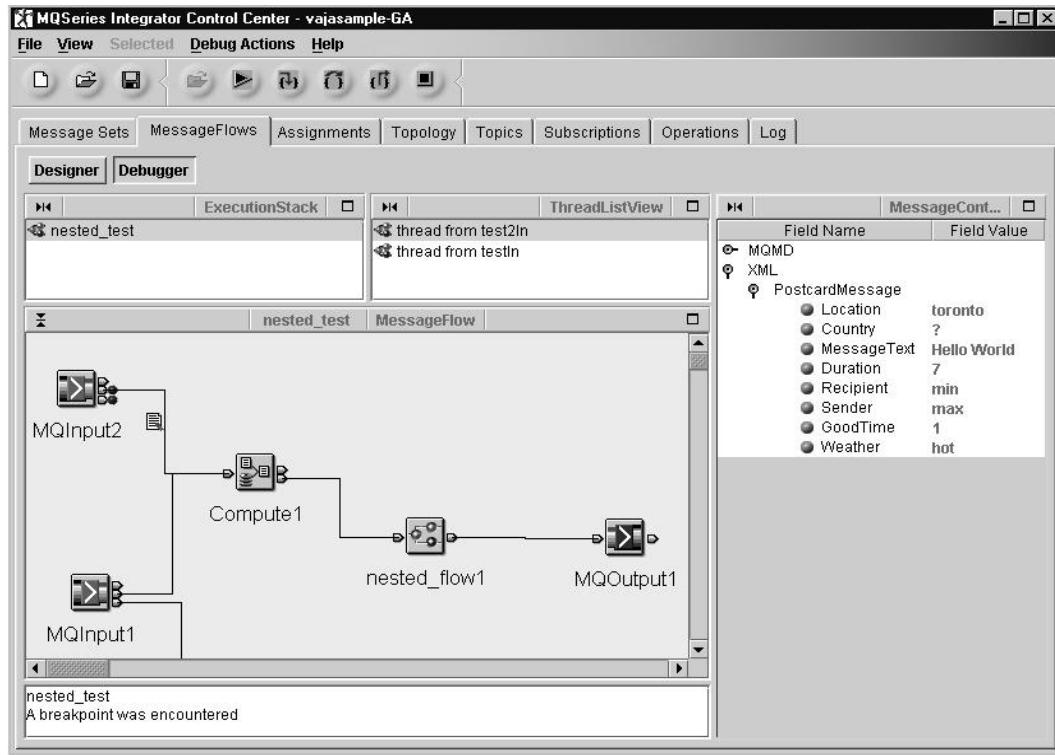


Figure 7 User Interface for Non-blocking Mode Debugger

#### 4 Performance analysis

Any system or subsystem has both functionality and performance requirements. Generally, performance requirements in a distributed system are assessed in terms of CPU time, the used memory, used disk space, the number of transactions, and connect times to the server (such as in socket operations). MQSI has a complicated runtime system and deployment operations require significant CPU time. Therefore, the usage of CPU time for the debugger is more important than other factors. In this paper, we focus on evaluating the usage of CPU time. The aim of our analysis is to identify and eliminate causes of unexpected and long response times, thereby improving user-perceived performance.

The message flow debugger is an interactive subsystem. It is hard to measure the entire system's efficiency, because the performance tester is not in control of how

fast test subjects respond. Therefore, we will focus on measuring the CPU time of threads.

We have noticed that debugging a message flow may have poor performance in the following situations:

- When the message flow is large, with hundreds of processing nodes.
- When the message flow is complicated with many input nodes and there are many branches in the flow graph.
- When the message flow is not complicated, but input nodes can launch more than one thread at a time.
- When many messages are in the input queues while debugging is taking place.

In general, the function of setting breakpoints is the most common operation for users of the debugger. When the user sets breakpoints, the message flow that the user is

debugging can interact with the user. In this situation, no matter how many messages are in the input queue, the execution time for the debugger is generally not important because the time that the user takes to check the information is generally longer than its execution time. However, since the message flow is multithreaded, for message flows with the above conditions, debugging will appear slow to the user. For example, for a message flow with many input nodes and many branches, it is possible that some threads will never meet a breakpoint when they pass through the message flow. It is shown in the following tests that such a flow is inefficient.

Our analysis shows that this inefficiency is due to the socket communications between the debug plug-in node and the debug controller. The specific reasons for this are:

- For each forward communication (from the debug plug-in node to the debug controller), the thread in the debug plug-in node has to wait for the debug controller's reply.
- The socket communication itself is expensive.

Our performance analysis consists of three tests. Test 1 compares the execution time of each debug plug-in node in different modes. Test 2 is designed to determine which part of the code in the debug plug-in node is the bottleneck.

### Test Environment

These tests are executed on IBM Pentium 3 700 with 384 RAM. The operating system is Windows NT 4.0, on which the message flow debugger is developed originally. Since we only test the efficiency of the debug plug-in code, the platform does not affect our test results too much. Except for the operating system and the software that is required by the operating system, there are the following jobs running: MQSeries 5.1, DB2 EEE 6.1, MQSeries Integrator 2.0.

### Test Method

In these tests, performance is measured statistically. Since a single message takes very little time to flow through the test

message flow, we send many messages and get their overall time, then average the time for one message or one debug plug-in node. Each averaged datum in Table 1 and Table 2 is obtained from at least three replications of the same measurement process.

There exists variance in measurement. The measurement was done manually by using stop watching, which creates some human errors, however, since measurement was done once for 2000 messages, the error for one single message is negligible.

### Test 1

The purpose of this test is to compare the efficiency (measured in terms of CPU time) of debug nodes in all modes.

- 1 **Hypothesis:** static blocking and non-blocking modes are more efficient than dynamic blocking mode. Socket communication is the likely bottleneck.
- 2 **Test message flow:** the flow is shown in Figure 8.
- 3 **Test method:** send a large number of messages to the input queue, and run the message flow in the five modes as summarized in Table 1. We calculate the time between the first message exiting the input queue and the last message getting into the output queue. All modes have different numbers of socket operations, so we can determine in this way whether the socket operations affect the execution time of the flow.

In Table 1, the data in column 4 (we refer to it C4) is obtained by measurement. C5 is from the calculation of  $C3/2000$ , C6 from the calculation of  $(C5-0.0305)/10$ .

From the last column of Table 1, we can see that the execution time of the static blocking mode debugger is almost 14 times more efficient than the Haifa debugger, and 42 times more efficient than the dynamic blocking mode debugger. The non-blocking mode is almost 140 times than the Haifa debugger. However, the dynamic blocking mode debugger takes more time than the Haifa debugger. This is because the Haifa debugger does not block any threads,



Finally from Table 2 we can calculate the time for each backward socket as:

$$(25.2 - 16) / (200 * 1 \text{ backward socket}) = .0046 \text{ sec} = 46 \text{ ms}$$

Considering the test condition we have and the method employed, the above results are fairly good. For example, in the same test, 10 sockets in forward and 10 sockets in backward will take:

$$(150 \text{ ms} + 46 \text{ ms}) * 200 * 10 = 392 \text{ sec}$$

CEFS in the tested message flow for one message is:

$$5.15 \text{ ms} * 200 * 10 \text{ nodes} = 10.3 \text{ sec.}$$

The calculated total time (392sec+10.3sec) is very close to the direct measurement result (420sec).

Figure 9 presents the timing for various components of executing a debug plug-in node. Hence it can be seen that each forward socket operation takes 150ms; this is a large part of the time in the execution of a message flow. Therefore, the performance can be improved by reducing the number of socket operations.

This measurement shows that our hypothesis is correct.

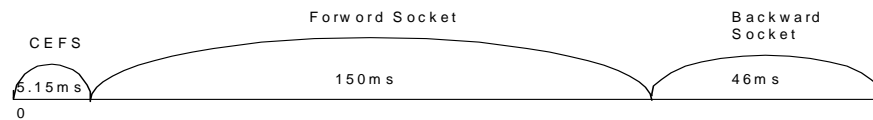


Figure 9 The Time for Executing a Debug Plug-In Node

The number of the calls and return through the sockets for each message	The number of sockets operation for 200 messages	Time (sec) for all messages to get into the output queue
0 socket in forward and in backward ( Without debugger )	0	5.7 ± 12.3%
10 sockets in forward and in backward ( The Haifa debugger )	200 * 20	436 ± 0.1%
10 sockets in forward and in backward, ( The dynamic blocking mode)	200 * 20	420 ± 0.1%
0 forward , 0 backward ( The static blocking mode )	200 * 0	16 ± 1.88%
1 socket in forward with sending "a", 0 socket in backward ( The static blocking mode )	200 * 1	46 ± 1%
0 socket in forward, 1 sockets in backward, ( The static blocking mode )	200 * 1	25.2 ± 7.1%

Table 2 Execution Time of Each Section of the Code in the Debug Plug-In Node (DLL)

### Discussion

In the Haifa debugger, each TCP/IP communication that sends a message to the controller and waits for a reply from the controller takes approximately 150

milliseconds. This expense is caused by the fact that the debug nodes have to communicate with the controller even if there are no breakpoints on any nodes of a path from the input node to the output node.

Testing has also shown that without this communication, a debug node adds no more than a few hundred milliseconds per node to the message transit time. Therefore, based on these tests, we reduce socket communication in the extended debugger. In dynamic blocking mode, backward socket communication is reduced by using mutex objects in the debug plug-in node. In static blocking mode and non-blocking mode, forward socket communication is reduced by using breakpoint attributes in the debug plug-in nodes.

The tests show that the results of poor performance are due to the cost of TCP/IP transactions. Since the message flow model is multithreaded, it is possible for a message to go through a message flow avoiding all breakpoints, hence no socket communications are necessary. In the static blocking mode and non-blocking mode, we have used thread local storage and daemon threads to reduce the expense of communication. The tests show that performance improvements can be achieved by reducing the number of socket operations and by using thread local storage, a runtime stack, and daemon threads.

## 5 Conclusions

The Haifa debugger is designed to handle simple message flows with one input node. However, it has limitations in debugging nested and multithreaded message flows. As well, its performance is not acceptable for production. In this paper we presented a few extensions to the Haifa debugger which significantly overcomes the restrictions of the Haifa debugger. Some of the features of the extended debugger include:

- It allows ending a debug session correctly without wrapping debug output with the next debug session;
- It allows users to set breakpoints on the fly and display the message flow names in the execution stack windows.
- At the same time, performance tests show that the extended debugger improves performance on single

machines and on networked computers.

We extended the Haifa debugger step by step. First, we developed the dynamic blocking mode debugger to support blocking threads in debug plug-in nodes. We then extended the debugger to support ending a debug session correctly.

Secondly, in order to improve the performance of the debugger, we developed the static blocking mode debugger by adding breakpoint attributes and UUIs of message flows to debug nodes in order to make them smarter. This reduces the cost of socket communications. Meanwhile, we used **thread local storage**, a **run-time stack**, and **daemon threads** algorithms [8] to implement enhanced blocking mode debugger. Thirdly, in order to allow multithreaded message flows to work efficiently, we developed the non-blocking mode debugger.

## References

- [1] *MQSeries Integrator*, [http://www3.ibm.com/software/swprod/swprod.nsf/key?SearchView&Query=\[title\]=MQSeries+Integrator+or+\[MetaKeywords\]=MQSeries+Integrator](http://www3.ibm.com/software/swprod/swprod.nsf/key?SearchView&Query=[title]=MQSeries+Integrator+or+[MetaKeywords]=MQSeries+Integrator)
- [2] *IBM MQSeries Primie*, IBM Documentation, IBM Toronto Library.
- [3] Frank Leymann, Dieter Roller, *Production Workflow - Concepts and Techniques*, Prentice Hall, Inc., 2000
- [4] A.R. Neta, *Message Flow Debugger Architecture*, May 2000, Haifa, Israel.
- [5] *NEON Technology: e-Biz Integrator™*, <http://www.neonsoft.com/products/e-Biz.html>
- [6] Jonathan B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley & Sons Inc., 1996
- [7] Peter A. Buhr, Martin Karsten and Jun Shih, *KDB: A Multithreaded debugger for Multithreaded Applications*, Proc. SPDT '96, Philadelphia PA, USA, 1996.
- [8] Shuxia Tan, *Extending a Message Flow Debugger for MQSI*, MSc Thesis, Department of Computer Science, York University, 2001