SPE Models for Multi-Tier Client/Server Interactions with MQSeries and Other Middleware

Connie U. Smith
Performance Engineering Services
PO Box 2640
Santa Fe, New Mexico, 87504-2640
(505) 988-3811
http://www.perfeng.com

Abstract

This paper describes how to construct Software Performance Engineering (SPE) models of multi-tier client/server interactions with middleware such as MQSeries. It covers typical performance problems in distributed systems of this type and gives a brief review of the SPE modeling approach. A case study illustrates how to create the SPE models, detect and correct performance problems early in the life cycle when software architecture and design alternatives can lead to significant improvements.

1. Introduction

This paper addresses Software Performance Engineering (SPE) of software systems that will use MQSeries. The purpose of the SPE models is to perform an early assessment of the overall software architecture to determine whether it will meet performance objectives. Continued application of SPE techniques throughout the development process helps insure that performance goals are met. The middleware performance will affect the overall system performance, but it is only one aspect of the overall performance that is represented with the SPE models.

MQSeries is middleware that enables a company to integrate applications running on different computing systems. MQSeries delivers messages from client applications to server applications and returns the response. It provides queue structures that can be used to control sequencing, priorities, and multi-threading. It provides an option for persistent queues that can be recovered in case of a system failure. The techniques described here are not limited to MQSeries; they apply to other types of middleware that provide similar services.

The SPE performance issues that are important in this type of client/server system are:

- *What are the performance goals?* Goals typically specify the end-to-end response time or throughput for business tasks. For client/server systems, the business tasks typically span multiple interactions between client and server.
- *How do you select from the myriad of alternatives?* Middleware choices include the middleware product, the database product, whether separate middleware server(s) are required, the platform sizes, the network bandwidth(s), and middleware configuration parameter settings. Application choices include where processing should execute, remote queries versus "stored procedures," how much data to transfer, etc.
- *How do we represent and assess end-to-end performance?* End-to-end performance models represent all of the processing that occurs from the time a user initiates a business task until all work is complete. The models can be complex; SPE techniques simplify their analysis.
- *How do we identify which components are critical to performance?* Identification of the key processing steps in the overall system leads to better performance management of the evolving system.

The SPE models and their evaluation are covered in the remainder of the paper. The next section covers typical performance problems in multi-tier client/server

systems. Then section 3 gives an overview of the approach, and section 4 illustrates the SPE models with a case study.

2. Typical Performance Problems in Client/ Server Systems

This section covers problems often encountered in the study of systems of this type. The SPE performance models described later can detect these problems and provide quantitative assessment of alternatives.

- *Create streamlined transactions for most frequent workloads* - Overall performance will be better when you minimize the processing required for frequent tasks. Streamline the frequent tasks and provide other functions that do more general tasks less frequently.
- *Remote queries vs. "stored procedures"* - a client can issue individual queries to a server or can execute a "stored procedure" that executes several processing steps and perhaps multiple queries before returning results. When clients issue a series of remote queries, where the results of a query are (only) used to format another remote query (i.e., chained queries), the amount of network traffic can be reduced using stored procedures to process the set of queries and return only pertinent results.
- *Amount of data returned to client* - remote queries may return all columns in a query, or only a subset of columns. Code generation packages may default to retrieving all columns. Network traffic can be reduced by returning only necessary data. Systems often return too much data because they are unsure how much the user wants to see. This is particularly important for business tasks with very high execution frequency - create separate transactions for high-frequency tasks to retrieve only necessary data and use different transactions to retrieve all data less frequently.
- *Repeated queries for static information* - Software may be designed to retrieve static information from a database rather than hard coding it into software or screens. If changes are made to this static information, it is only necessary to change it in the database and the results are automatically propagated to all clients the next time they do a query. If the data was included in the code or on screens, new software would have to be distributed to all clients. This strategy reduces the overhead for software distribution at the expense of extra processing to retrieve static information. An alternative to reduce network traffic and processing time is to cache the results of the first query on the client and check periodically for changes.
- *Software distribution* - When software is revised it must be updated on all platforms. Broadcasting new copies of software to all clients is time consuming and resource intensive. Even though it should happen infrequently, it is important to question whether this workload should be included as one of the SPE performance scenarios. For example, if software distribution may occur as a part of the login process it is vital to determine the estimated delay for login and the impact on the overall system. Alternate distribution strategies can reduce unnecessary congestion.
- *Granularity of processing* - Systems are typically designed to process one request at a time. For expensive processing, it is usually better to collect requests into batches and process groups of requests together.
- *Polling for pending work* - In this pattern, code repeatedly checks to see if new work arrived. The interval between polling must be short enough that pending work is not delayed too long, and long enough that the system does not bog down with polling and become unable to do other work. Consider alternatives to polling that "wakes up" or interrupt a process when new work arrives.
- *Synchronous vs. asynchronous processing* - A synchronous request waits for the reply before performing other work. An asynchronous request allows processing to continue, but when the reply arrives, the requestor must either be notified of the response or it must periodically poll to see if it arrived. Asynchronous processing may be more difficult to implement than synchronous. SPE models can determine when overlap of processing is essential to meet performance goals and indicate when asynchronous processing is required.
- *Multithreading* - Server processes must handle requests from multiple clients. Multiple copies of the server process (threads) may be required to handle the combined load. It is important to identify processes that require multi-threading before they are implemented - it may be too difficult later.

These are typical problems in distributed and client/ server system architectures regardless of the implementation technology such as middleware packages or implementation language. There are many others. We wish to use SPE models to detect these and other problems early in development stages while they can be easily prevented. The case study described later illustrates some of these problems.

3. SPE Modeling Approach

This section gives a brief review of the overall approach. It is illustrated in section 4 with a case study.

1. *Select key performance scenarios:* The important performance scenarios are those that are critical to the operation of the system or which are important to responsiveness as seen by the user. Typically, this is a subset of Use Cases that are identified during system analysis [KRUC99]. The key scenarios are those which are executed frequently or those which are critical to the perceived performance of the system.

2. *Represent key performance scenarios with sequence diagrams.* A sequence diagram represents each entity or object that participates in the scenario by a vertical line or axis. The vertical axis represents relative time which increases from top to bottom. Interactions between objects (messages, events, operation invocations) are represented by horizontal arrows.

3. *Translate sequence diagrams to execution graphs:* The sequence diagram representation is translated to one or more execution graphs. We obtain estimates of the amount of processing required for each step in the execution graph. Early in the development process, these may be simply best/worst case estimates. Later, as each processing step is elaborated, the estimates become more precise [SMIT88c;SMIT90a].

4. *Add processing overhead:* The processing steps in an execution graph are typically described in terms of the software resources (e.g., database accesses or messages transmitted) used. Processing overhead maps these software resource requirements onto the amount of service they require from key devices in the hardware configuration. The case study illustrates how to determine the processing overhead for the middleware component.

5. *Solve the models:* Solving the execution graph characterizes the resource requirements of the proposed software alone. If this solution indicates problems, analysts consider design alternatives to address the problems. If not, then analysts proceed to solve the system execution model to quantify the effect of resource contention delays on response time.

The following discussion assumes some familiarity with sequence diagrams and execution graph models. They are described in more detail in [SMIT98c; SMIT97].

4. Case Study

A new distributed system will support Customer Service business tasks such as checking on the status of orders, shipments, invoices, and payments. The initial release of the system will support telephone inquiries from customers. Agents in multiple call centers will generate inquiries to provide data that will answer customer questions and initiate corrective action when necessary. A future release will support automatic generation of queries from internet-generated customer inquiries, and automated responses.

4.1 Performance Scenarios

The performance scenarios we will consider support the following Customer Service business tasks:

- *Invoice history* - Create a form with all information about all open invoices satisfying the query conditions.
- *Payment history* - Create a form with all information about all payments satisfying the query conditions.
- *Initiate inquiry* - Complete a form with information about a problem, update the database and trigger processing to handle the inquiry.

These are a subset of the business tasks (Use Cases) to be handled by the new system. They are selected to be performance scenarios that will be modeled and tracked throughout development because of their frequency and importance to the overall suitability of the system. SPE will start with this subset of tasks. Others will be added to the set of performance scenarios after the initial study of these scenarios. For example, the login process has a large amount of initialization processing. It will be evaluated in the next stage after we are sure that the overall architecture is sound.

The system will be a multi-tier client/server architecture. Business components on the client will manage the graphical user interface with the user, and submit requests via MQSeries to interact with the DB2 database on the mainframe. CICS transactions on the mainframe will receive the requests via MQSeries, execute the inquiry or update against DB2, and package response messages that will be returned to the client via MQSeries.

4.2 End-to-End Processing Flow

The processing flow for a general inquiry is in Figure 1. The only difference in the two queries is the volume of queries and the amount of data received. The general flow thus works for both queries; it is described in the following paragraphs.

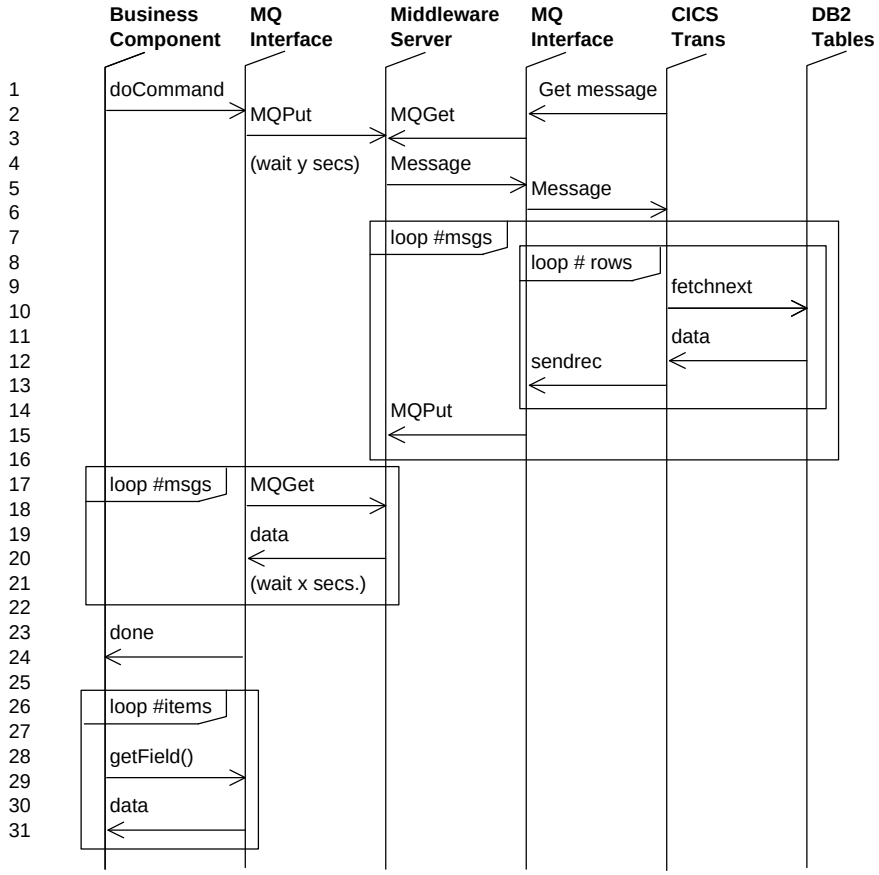


Figure 1: Inquiry Processing Flow

Client processing: The business component (Invoice history or Payment history) requests data from the MQ Interface component (lines 1-2). The MQ Interface puts the request onto the queue on the Middleware server (lines 2-3). Inquiries are to be asynchronous so the MQ Interface component can process additional requests made by the user (in another window). It waits for y seconds (line 4) then requests the reply (line 18). The value we will use for y will be the estimated time for the mainframe transaction to put the first reply message in the queue. The reply comes in messages of 4K bytes each, the final message will contain a last-message indicator. When there are additional messages for a request, the MQ interface will wait x seconds and request the next message; the value we will use for x will be the estimated time for the mainframe transaction to send the next message. The data request by the MQ Interface is asynchronous, however the business component makes a synchronous request so it waits until the data is retrieved (lines 23-24). It then makes one request per field per row of data returned by the server and inserts each field into its appropriate place in the form displayed to the user (lines 26-31).

Mainframe processing: Each business task has an associated CICS transaction on the mainframe. It begins by issuing a Get message call (lines 1-2) and waiting for the next message to be delivered by the MQ Interface routine. When a message arrives (lines 5-6), the inquiry transaction uses the query specifications received and makes the appropriate SQL query to DB2. It retrieves each row that qualifies and sends the data to the MQ Interface (lines 8-13). When enough rows are received to fill a message, the MQ Interface puts it on the appropriate Middleware server queue (lines 14-15).

Middleware server processing: The architecture calls for a set of queues (input and output) for each call center. Processing includes: receive MQPut from clients and post them to the designated queue (lines 2-3); receive MQPut from the mainframe for responses and post them to the designated queue (lines 14-15); receive MQGet requests from the client and return results from the queue (lines 17-21).

The processing for the Initiate Inquiry scenario is similar to the inquiry processing - the processing flow is not shown. The key differences are:

- the client makes a synchronous request and waits for confirmation that the update processing is complete (so there is no delay before MQGet on lines 17-18)
- the reply requires only one message (so the number of repetitions is 1 for the loops on lines 7, 8, and 17)
- there is no form to complete (so the repetitions is 1 for the loop on line 26)

This system appears to be more complex than necessary. Client/Server systems do not necessarily need middleware servers between the client and server. This system evolved this way because the developers want to reuse code on the client and server platforms, the middleware lets them connect the two without the need to develop their own software to make the connections.

4.3 Early Life Cycle Models

The early life cycle models first focus on the middleware server processing. The mainframe transactions are similar to existing legacy transactions, and simple analysis indicates that the mainframe will handle the projected load for the first release. The client supports only a single user so the local processing does not change as the workload increases. Both the mainframe and the client processing will be included in subsequent studies.

Three performance scenarios are assigned to the middleware server:

- Inbound messages: Receive requests from client and post to queue
- Mainframe results: Receive responses from mainframe and post to middleware queue
- Client polls for results: Return responses from middleware queue to client

Figure 2 shows the model for the inbound messages. A case node shows that the inbound message may be one of three types: query, update or login. Login is not included in the initial study so its execution probability is 0 and details are not shown. The query alternative has overhead to represent inserting the request in the appropriate queue, and triggers an asynchronous request on the mainframe. It does not wait for a reply. The update alternative represents the processing for the Initiate inquiry scenario. It also has overhead for inserting the request in the queue, and makes a synchronous request for processing.

Figure 3 shows the software model for mainframe results. The case node shows two alternatives: invoice history and payment history. The resource requirements will specify the size of the results for each choice The inquiry result is in a separate process because it executes asynchronously. We don't need a separate scenario for the update because it executes synchronously and we represent the processing for the reply in the doUpdate step

Figure 4 shows the processing for client polls for results. The first case node shows that if the reply is ready, the reply is sent. The next case node shows that if the last message has not been received there is a delay before the next poll. There is a submodel like Figure 4 for each type of (asynchronous) query.

These models are constructed and evaluated with the SPE modeling tool *SPE•ED*™. The model formulation is geared to *SPE•ED*, however the techniques can be adapted to any type of SPE modeling tool.

4.4 Processing overhead

We need measurements for the resource requirements for MQSeries to process our typical client and server requests. The most important requests for these scenarios are MQGet and MQPut. We will also need measurements for MQConnect, MQDisconnect, MQOpen and MQClose when we evaluate the login scenario. Thus we need to measure resource requirements for six MQSeries commands.

Many MQSeries parameters influence the resource requirements. The most important at this early stage are:

- persistent queues are saved on disk, non-persistent queues are stored in virtual memory
- the size of messages (note that MQ adds a 500 byte header to each message)
- batch size and batch timeout parameters regulate the maximum number of messages transmitted at a time. CPU requirements per message are reduced with batching, but response time may be sacrificed.

Two other factors have a great influence on the performance of the overall system:

1. The assignment of messages to queues will primarily affect queueing delays for scheduling requests (and the time to initialize queues at login).

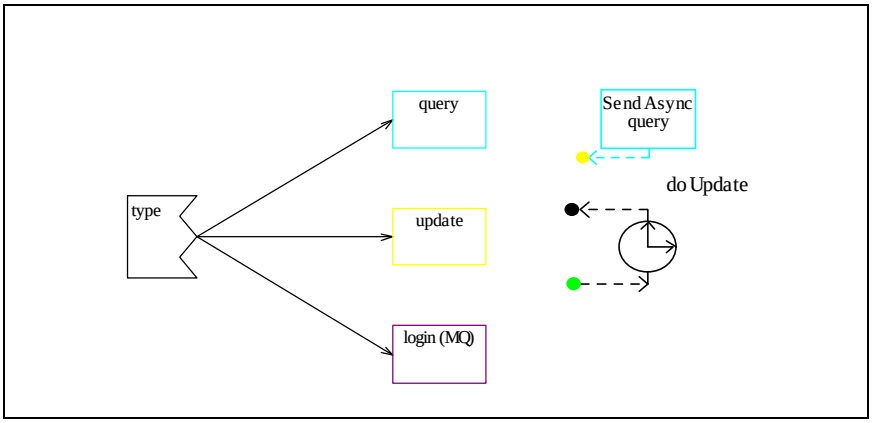


Figure 2: Middleware server: inbound message

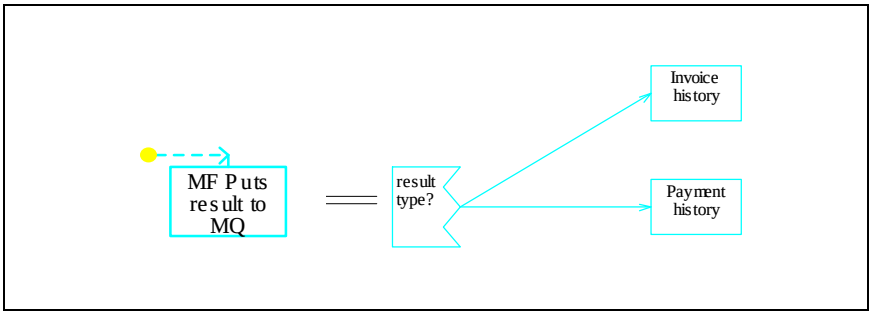


Figure 3: Mainframe sends results.

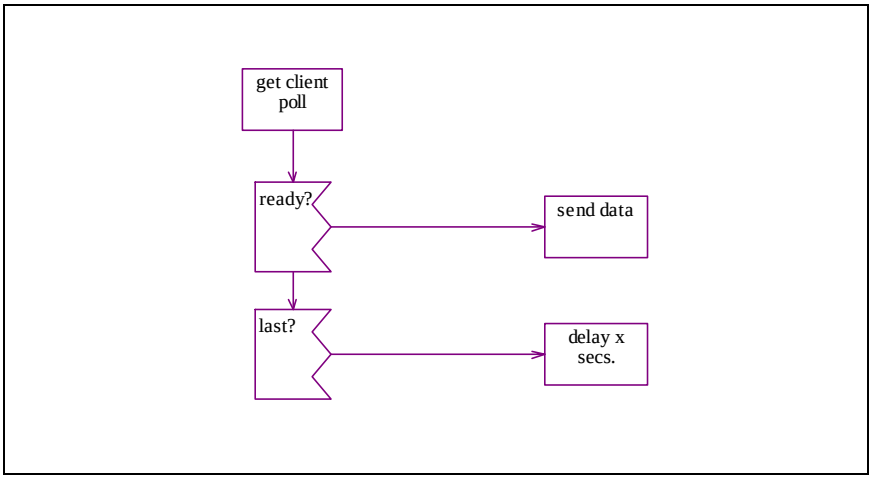


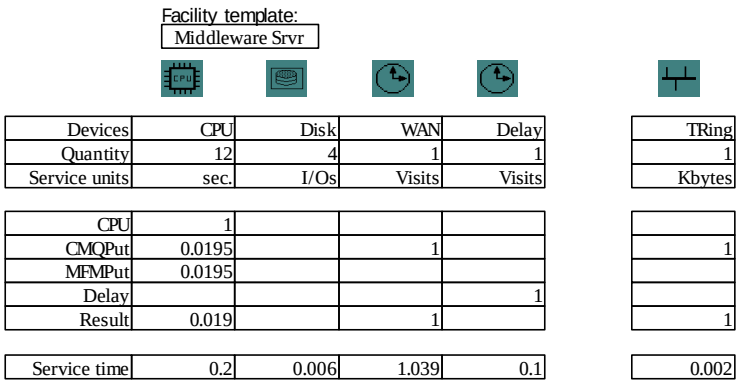
Figure 4: Client polls for results.

2. The volume of message arrivals affects the total resource requirements for queue processing and the queueing delays for scheduling requests.

These factors can dramatically affect the overall performance of the middleware server, but they have a minor affect on the resource requirements for an individual MQGet and MQPut. We need the resource requirements to create the SPE models; the model solution can provide estimates of total processing requirements and queueing delays for a given queue assignment strategy.

To gather the resource requirements for the six MQSeries commands you will either need to conduct your own benchmark study, or use the results of a study conducted with workloads and MQ parameter settings that match your future environment. Procedures for conducting benchmark studies may be found in numerous CMG papers.

After we collect measurements for MQGet and MQPut, the next step is to specify the types of software resource requirements and their processing overhead. The software resources we use for this case study are:



Devices	CPU	Disk	WAN	Delay	TRing
Quantity	12	4	1	1	1
Service units	sec.	I/Os	Visits	Visits	Kbytes
CPU	1				
CMQPut	0.0195		1		1
MFMPut	0.0195				
Delay				1	
Result	0.019		1		1
Service time	0.2	0.006	1.039	0.1	0.002

Figure 5: Processing Overhead

1. CPU - an estimate of the CPU time required for miscellaneous processing

2. CMQPut - the number of times a client processing step issues an MQPut command

3. MFPut - the number of times a Mainframe processing step issues an MQPut command

4. Delay - the relative delay before the client makes the next request, or for a synchronous request the estimated time before the mainframe sends the response.

5. Result - the number of result messages returned to the client (includes the MQGet command).

Analysts specify values for these five software resource requirements for processing steps in each performance scenario. The computer resource requirements for each software resource request are specified in an overhead matrix. This matrix is used for all software models that execute in that hardware/software environment. Figure 5 shows the overhead matrix with the processing overhead for this case study. The five software resources are in the left column of the matrix; the devices in the facility are in the other columns. The values in the matrix describe the device characteristics. The contents of an overhead matrix are described in more detail in [SMIT97]. This paper focuses on the processing overhead for this case study.

The values in the center section of the matrix define the connection between software resource requests and computer device usage. The values in the CPU column are the measured values for MQGet and MQPut. (The other four MQSeries commands are used by only a few steps, so their CPU values are not specified in the overhead matrix, they are specified for those steps, and the 1 in the CPU row causes the resulting demand to be attributed to the CPU device.) The service time for the CPU is 0.2 because the measurements were made on a slower processor; the actual system will be deployed on a processor 5 times faster. There are no values in the disk column for this alternative because the queues are non-persistent.

The ones in the WAN column mean that each 'CMQPut' and 'Result' specified in the software model causes one visit to the WAN representing either one input message or one output message between the clients and middleware server. Similarly, the ones in the TRing column represent either one input message or one output message transmitted on the LAN between the middleware server and the mainframe. The service times for the WAN and the LAN represent the time for a 4K message on the respective bandwidths of the two networks. The WAN is a delay server because this early life cycle model does not model contention on the WAN. The configuration is such that call centers do not interfere with one another, the link is already sized to handle the load for the legacy system, and we assume that the capacity is sufficient for the new system. The network configuration issues will be examined later after the architecture issues are resolved.

Thus each value specified for a processing step in the software model generates demand for service from one or more devices in the middleware server. The overhead matrix defines the devices used and the amount of service needed from each device. The demand is the product of the software model value times the value in the overhead matrix cell times the service time for the column.

.

Table 1: Sample Resource Requirements

	Query	Update	Invoice history	Payment history	Delay first	Send data	Delay next
Probability	0.75	0.25	G1	G3		G2	G2 - 1
CMQPut	0.125	1					
MFPut		0.1	1	1			
Delay		2			5		3
Result		0.1				1	

Given these values for processing overhead, we specify values for each processing step in the software models for the number of software resource requirements. Values for some of the processing steps are shown in Table 1. The G1, G2 and G3 are parameters whose value can be changed to study sensitivity, scalability, and other values.

Specifications for the arrival rate of each of the three performance scenarios, and the number of results expected for each type of interaction complete the model.

The solution to this model gives the response time for the individual scenarios and the utilization of the network and middleware server. The end-to-end time for the performance scenarios and other results are described in the next section.

4.5 Early Life Cycle Results

The results are in Figure 6. Part (a) shows the results for the inbound message; part (b) shows the results for the mainframe to send results; and (c) shows the results for the client to poll for results. Each set of results shows the overall residence time in seconds, the time for each processing step, and in the "Resource Usage" section, the total time at the CPU, Disk, WAN, Delay, and Tring.

The most interesting set of results is (c) client polls for results. Of its 9 seconds response time, 6.8 seconds is for the WAN to deliver the messages to the client, and 2.2 is the estimated delay for the messages to arrive from the mainframe. The total end-to-end response time for queries is the sum of the times for the 3 parts of the processing. The total end-to-end time for the update is 0.34 seconds because it is a synchronous request. The total time for the login is 0 because its details are not included in this model.

Thus the primary problem is the amount of data transferred over a relatively slow WAN. Three solution alternatives were identified:

1. Create a slim inquiry that returns only the data that is used most frequently (narrower rows). The slim inquiry would be used most of the time, the fat inquiry only in special circumstances.

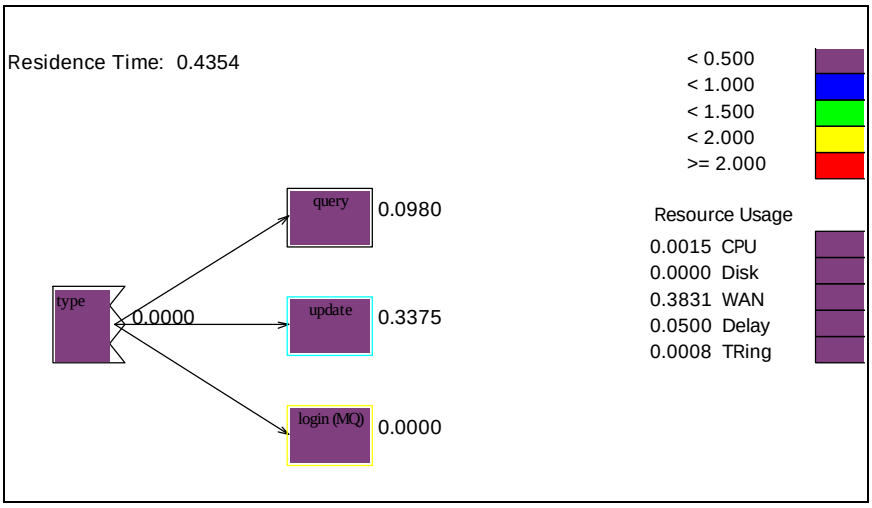
2. Increase WAN bandwidth

3. Revise response time objectives. The fat inquiry will take longer to complete than transactions in the previous system.

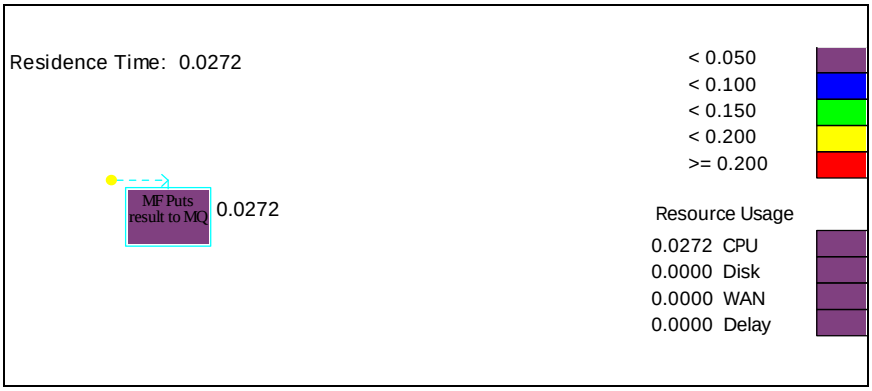
In addition to identifying the above problem in the software architecture, models also provided data for the following:

- Determined appropriate delay between polls
- Sized Middleware server(s) to support all client traffic
- Identified server configuration issues, and possible need for dynamic reconfiguration because of MQSeries queue structures

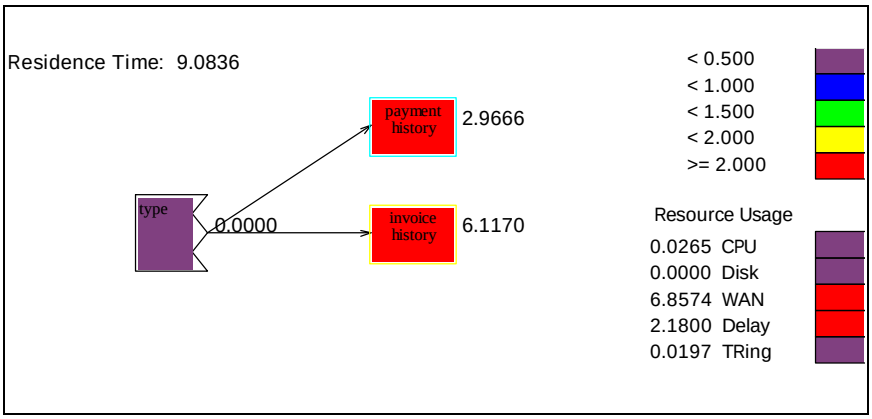
Note that these simple models provide sufficient information to identify and correct problems Later, more advanced models such as those in [SMIT98c] will investigate potential synchronization problems and queueing delays due to the planned queue configurations. They will represent processing on the mainframe as well as the client and make the connections at the designated synchronization points.



(a) Inbound message



(b) Mainframe puts results



(c) Client polls for results

Figure 6: Early Life Cycle Results

5. Summary

This paper presents an approach to modeling the end-to-end performance of multi-tier client/server systems that use middleware between the clients and servers. It describes some typical problems that occur in systems of this type. They are the types of problems that can be detected with SPE models. The models also quantify the performance of design alternatives.

The paper describes the SPE modeling approach for distributed and multi-tier client/server systems, begin-

ning with the identification of performance scenarios, representing the processing with sequence diagrams of the end-to-end processing, the translation of sequence diagrams into execution graph models, defining resource requirements and processing overhead, and evaluating the SPE model results. The approach was illustrated with a case study.

It is important to model systems with complex interactions early in development. Problems detected then have more options for correction - after implementation it is more difficult perhaps impossible to correct fundamental problems with the architecture and design. The SPE models can detect and prevent the typical problems; they can also identify other performance problems that may be less common.

References

[KRUC99] Philippe Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley, Reading, MA, 1999.

[SMIT88c] Connie U. Smith, "How to Obtain Data for Software Performance Engineering Studies," Proceedings Computer Measurement Group Conference 88, Dallas, TX, 1988,321-329.

[SMIT90a] Connie U. Smith, Performance Engineering of Software Systems, Addison-Wesley, Reading, MA, 1990.

[SMIT97] Connie U. Smith and Lloyd G. Williams, "Performance Engineering of Object-Oriented Systems," Proc. Computer Measurement Group, Orlando FL, December 1997.

[SMIT98c] Connie U. Smith and Lloyd G. Williams, "Performance Engineering Models of CORBA-based Distributed Object Systems," Proc. Computer Measurement Group, Anaheim, 1998.

[SMIT98c] Connie U. Smith and Lloyd G. Williams, "Performance Models of Distributed System Architectures," Proc. Computer Measurement Group, Anaheim, Dec. 1998.