Performance Evaluation of a Distributed Software Architecture

Connie U. Smith[†] and Lloyd G. Williams[§]

[†]Performance Engineering Services
PO Box 2640, Santa Fe, New Mexico, 87504-2640
(505) 988-3811, http://www.perfeng.com/~cusmith

[§]Software Engineering Research
264 Ridgeview Lane
Boulder, CO 80302
(303) 938-9847

Performance Evaluation of a Distributed Software Architecture

Connie U. Smith
Performance Engineering Services
PO Box 2640
Santa Fe, NM 87504
http://www.perfeng.com/~cusmith

Lloyd G. Williams
Software Engineering Research
264 Ridgeview Lane
Boulder, CO 80302

Abstract

There is growing recognition of the importance of the role of architecture in determining the quality of a software system. While a good architecture cannot guarantee attainment of quality goals, a poor architecture can prevent their achievement. It is particularly important to evaluate the performance of a distributed system architecture. Errors made early can cause excessive overhead for communication and coordination and they are far more difficult – if not impossible – to correct with tuning. This paper discusses assessment of the performance characteristics of distributed software architectures in early life cycle stages. The techniques are described and illustrated with a simple example.

1. Introduction

Architectural decisions are among the earliest made in a distributed software development project and can have the greatest impact on software quality. Thus, it is important to support assessment of quality attributes at the time these decisions are made. Our work focuses on early assessment of software architectures to ensure that they will meet non-functional, as well as functional, requirements. For this paper, we focus on techniques for the performance assessment of a distributed software architecture since many distributed systems fail to meet performance objectives when they are initially implemented.

Performance failures result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows. Moreover, "tuning" code to improve performance is likely to disrupt the original architecture, negating many of the benefits for which the architecture was selected. Finally, it is unlikely that "tuned" code will ever equal the performance of code that has been engineered for performance. In the worst case, it will be impossible to meet performance goals by tuning, necessitating a complete redesign or even cancellation of the project.

Our experience is that most performance failures are due to a lack of consideration of performance issues early in the development process, in the architectural design phase.

Poor performance is more often the result of problems in the architecture rather than in the implementation. As Clements points out:

> "Performance is largely a function of the frequency and nature of inter-component communication, in addition to the performance characteristics of the components themselves, and hence can be predicted by studying the architecture of a system." [CLEM96]

Thus it is particularly important to evaluate the performance of distributed system architectures. Errors made early can cause excessive overhead for communication and coordination and they are far more difficult – if not impossible – to correct with tuning. Our approach focuses on the generation and evaluation of design alternatives to allow assessment of the trade-offs associated with various architectural decisions.

In this paper we describe the use of software performance engineering (SPE) techniques to perform early assessment of a distributed software architecture to determine whether it will meet performance objectives. The use of SPE at the architectural design phase can help developers select a suitable architecture. Continued application of SPE techniques throughout the development process helps insure that performance goals are met.

The remainder of the paper explains the SPE approach to modeling distributed systems. and illustrates the process with a simple example.

2. SPE Process for Distributed Systems

At the architectural level of design, the SPE process for distributed systems calls for using deliberately simple models of software processing that are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. Thus our approach is to first create the software execution models that explicitly represent synchronization points among distributed processes and estimate the delay to receive results from a remote process. Later in development, more realistic models use advanced system execution model solutions to connect the distributed processes. The advanced models are described in a companion paper [SMIT98c].

The motivation for using approximate models early in development is illustrated in Figure 1 - a diagram with *N* Clients connected to a Server via a Network. Each client is modeled with a system model such as the one at the top of the diagram. User requests are processed on the Client's CPU and disks. At some point in the processing, the Client makes a request of a Server. It is transmitted via a Network (simplified to a single queue in the center section of this example), then sent to the Server. The server model at the bottom of the diagram is also a simple system model with a CPU and Disk. The request is processed on the Server's CPU and Disks, then returned to the Network. When it exits the network it returns to the Client making the request.

This picture is a greatly over-simplified view of a particular interaction in a distributed system, however, the resulting simulation model is still complex - containing a large

number of total queue-servers and workloads when all *N* clients are included -- and the solution time is too long to be used to compare many alternatives. Early in the development process, we are not interested in the intricacies of this interaction, but rather with the feasibility and desirability of various architecture and design alternatives. Thus to keep the models at this stage simple so we can study as many alternatives as possible, we will construct each of the three system models -- clients, servers and networks -- separately and estimate the delays for external system interactions. This approximation is described in the next section.

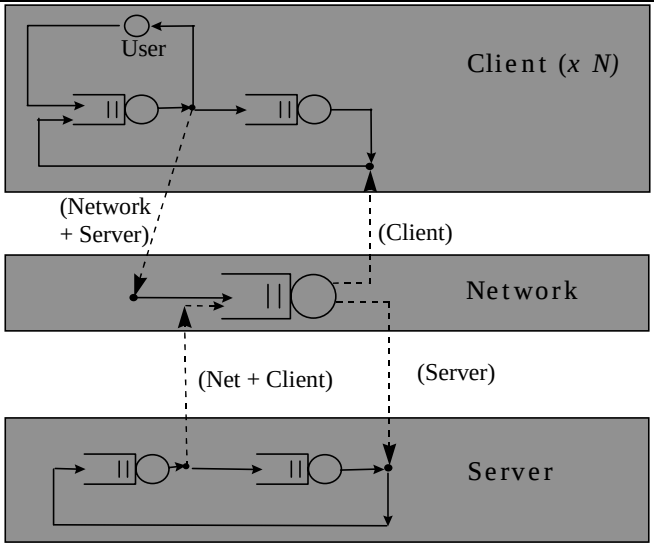

Figure 1 "Simple" Distributed System Model

3. Approximate models

This section explains the general strategy for separating the combined model into separate models. In the following sections we use terminology for synchronization and communication in distributed systems that use a CORBA compliant Object Request Broker (ORB).[1] The synchronization and communication terms are particular to CORBA distributed object technology. However these primitives are typical of most distributed systems. Even though the illustrations and discussions describe communication in terms of "client" and "server" processes, they apply to more general distributed systems. The roles of "client" and "server" refer to a particular interaction and may be reversed in subsequent interactions. In addition, the "server" process may interact with other processes, and multiple types of synchronization may be combined.

We consider three types of synchronization / coordination mechanisms between objects:

- *Synchronous invocation*: The sender invokes the request and blocks waiting for the response (a CORBA invoke call).
- *Deferred synchronous call*: The sender issues the request and continues processing. Responses are retrieved when the sender is ready (a CORBA send call, followed by a get_response call to obtain the result).

[1] CORBA is the Common Object Request Broker component of the Object Management Group's standard for middleware that facilitates development of applications in a distributed, heterogeneous environment. More information about it is in the companion paper [SMIT98c].

- *One-way (or Asynchronous) call*: The sender issues the request and continues processing; there is no response (a CORBA send_oneway call).

SPE model extensions to handle these performance issues are discussed next.

3.1 Synchronous Invocation

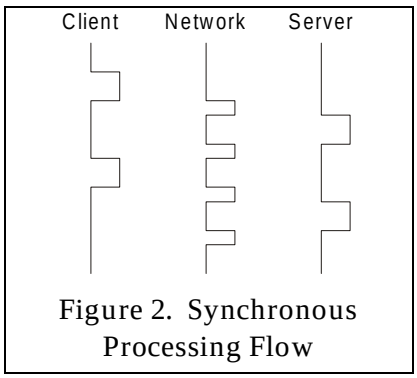

Figure 2. Synchronous Processing Flow

Figure 2 shows processing flow when a client process makes two synchronous requests to a server process. The "blips" in each column represent processing activity on the corresponding facility. The absence of a blip indicates that the corresponding facility is not processing *this request* – shared processing nodes (the Network and Server) may process other requests at those times. Processing for the first request is initiated on the Client; the synchronous request is transmitted via the Network; the Server processes the request and sends a reply; the Network processes the reply; the Client receives the reply and begins processing that will make another synchronous request. And so on.

Note that we do not explicitly represent overhead (CORBA - ORB) processing that might occur for run-time binding of the Client and Server processes. This illustration assumes that the processes are bound earlier. Analysts could model the overhead explicitly as another column in these diagrams, or implicitly by adding processing overhead for each remote request.

This profile is the basis for the first approximation. We create a separate performance scenario for each process, specify resource requirements corresponding to the "blips," and estimate the delay between "blips." The models may be iterative - the solution of each independent performance scenario quantifies its processing time. The processing time for the "dependent blips" can be used to refine the estimate of the delay between blips. In these illustrations we arbitrarily show the client and server objects on separate processors and the invocations must be transmitted through the network. If they reside on the same processor there is no delay for the network "blips."

This model is first solved without contention to determine if this optimistic model meets performance objectives. After correcting any problems, the system execution model solution quantifies additional delays due to contention from other work.

3.2 Deferred Synchronous Communication

Figure 3 shows one possibility for the processing flow when the client sends a deferred synchronous request to the server. The client process issues the request and continues processing. The processing on the Network and Server processing nodes is the same as before. At some point in the processing the Client needs the result from the server before it may proceed. The dashed line below the first Client "blip" represents the time that the Client must wait for the request to be completed and returned. The second possibility is that the deferred synchronous request is completed before the Client needs

the result to proceed. A figure representing this possibility would omit the dashed line and show a continuous blip in the Client column.

The model for deferred synchronous communication is similar to the previous one. We create a separate performance scenario for each process, specify resource requirements corresponding to the "blips," estimate the delay between "blips," and estimate the delay corresponding to the dashed line. If the second possibility holds, the delay estimate is zero. These models may also be iterative with solutions from one model refining the delay estimates for the next solution.

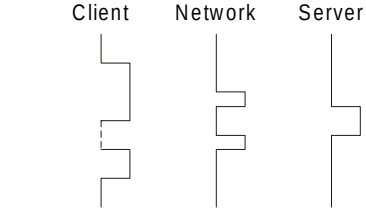

Figure 3. Deferred Synchronous Processing Flow

3.3 Asynchronous Communication

Figure 4 shows one possibility for the processing flow when the client makes an asynchronous request from the server. The client process makes the request and continues processing. The processing on the Network and Server processing nodes for transmitting and processing the request is the same as before, but there is no reply.

The model for asynchronous communication also has a separate performance scenario for each process. Analysts specify resource requirements corresponding to the "blips." There is no need to estimate Client delay because there is no response to the asynchronous invocation. If the Server model is an open model, we estimate the arrival rate of asynchronous requests; if they are closed models, we estimate the time between requests. Both estimates come from the processing time of the Client process; iterative solutions may be used to refine estimates.

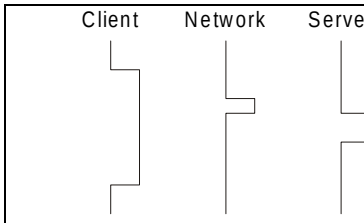

Figure 4. Asynchronous Processing Flow

3.4 Approximate Software Execution Models

We use the following steps to create an approximate model of software synchronization:

1. Create a separate performance scenario for the key processes in a distributed system
2. Insert special nodes into the execution graph to represent the particular type of synchronization / communication [2]
3. Estimate the delay for the called process to respond to the request
4. Solve the models.

[2] This discussion assumes familiarity with execution graphs to represent performance scenarios. Details may be found in [SMIT90a].

Later, the advanced system model connects the processes on the various facilities to quantify the synchronization costs and delays.

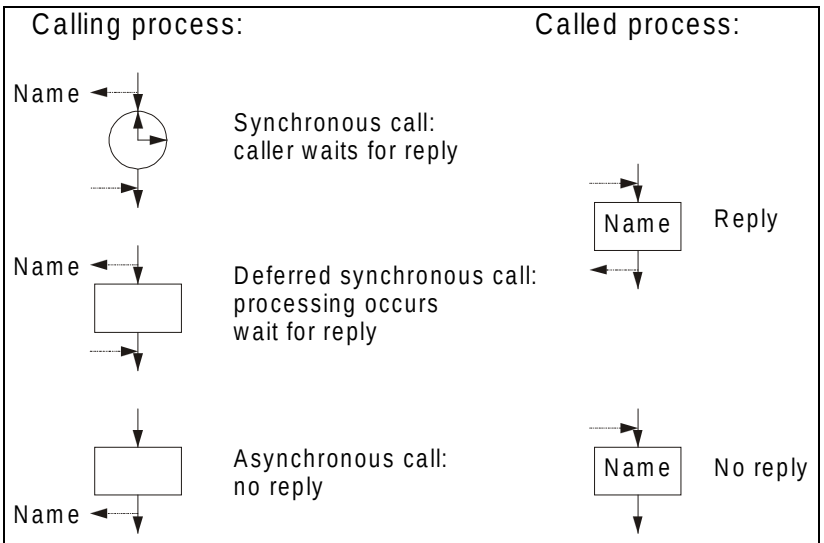

Figure 5. Execution Graph Nodes for Software Synchronization

Figure 5 shows the execution graph nodes that represent the three types of synchronization. The appropriate node from the left column of the figure is inserted in the execution graph for the calling scenario. The called scenario represents the synchronization point with one of the nodes from the right column depending on whether or not it sends a reply. The synchronization occurs in the calling process so the called process need not distinguish between synchronous and deferred synchronous calls. Any of the rectangular nodes may be expanded to show processing steps that occur between the dashed arrows or in connection with asynchronous calls. The expansion may contain other synchronization steps. The called process may execute additional processing steps after the reply is sent.

Next, the analyst specifies resource requirements for the processing nodes, the estimated delay for synchronization nodes, and the number of messages sent via the network. These steps are illustrated with the simple example in the next section.

4. Example

The example is a simple interaction to take and process a NewOrder. Developers often use *scenarios* at the architecture stage in development to describe the interaction between components in a distributed system. As described in [WILL95] scenarios represent a common point of departure between object-oriented requirements or design models and SPE models. Scenarios may be represented in a variety of ways [JACO92], [WILL94]. Here, we use Message Sequence Charts (MSCs) to describe them. The MSC notation is specified in ITU standard Z.120 [ITU96]. Several other notations used to represent scenarios are based on MSCs (examples include: Event Flow Diagrams [RUMB91]; Interaction Diagrams [JACO92]; and UML Sequence Diagrams [RATI97]). However, none of these incorporates all of the features of MSCs needed to establish the correspondence between software scenarios and performance scenarios.

For SPE, we translate the architecture-stage scenario described with an MSC into a *performance scenario* represented by an execution graph. This example uses the *SPE•ED* performance engineering tool to evaluate the resulting model. Other tools are available,

such as [BEIL95], [GOET90], [TURN92], but the model translation would differ for those tools that do not use execution graphs as their modeling paradigm. The modeling approach described in the following sections is partially determined by our tool choice.

The architectural tradeoffs that we consider include:

- the frequency and size of communication among processes
- the assignment of software components to processes
- the assignment of processes to processors
- three types of synchronization between processes
- the number of threads required for server processes

This example demonstrates how to construct a performance model that reflects the performance of a particular combination of choices. It does not consider architectural alternatives or how to correct problems in architectures to resolve performance problems. Other related papers address these issues [SMIT98c],[SMIT98],[WILL98].

4.1 Example Scenario

This example is a hypothetical set of interactions required to take and process a new order. It is based on an actual case study but is simplified to focus on modeling techniques for representing the distributed processing characteristics of the example.

Figure 6 illustrates a high-level MSC for the NewOrder example. Each object, component, or person that participates in the scenario is represented by a vertical line or axis. The axis is labeled with its name (e.g., anOrderTaker). The vertical axis represents relative time which increases from top to bottom; an axis does not include an absolute time scale. Interactions between components are represented by horizontal arrows. The rounded rectangle is an "MSC reference" which refers to another MSC. The use of MSC references allows horizontal expansion of processing steps. For example, the interactions required for enterData are represented in another MSC (not included in this example).

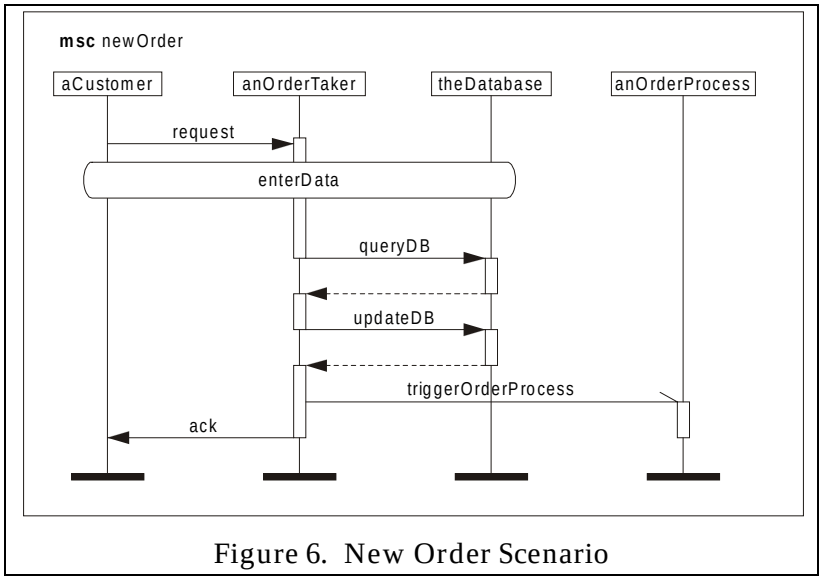

Figure 6. New Order Scenario

Next, the interaction represented in the MSC is translated into one or more performance scenarios modeled with execution graphs. This model will contain three scenarios – we do not create a scenario for aCustomer, but we create a scenario for each of the other three columns. We first examine the scenario that executes on a Client processor. Later sections present the two scenarios that execute on the Server processor.

The first performance scenario we consider is the processing represented in anOrderTaker column. The performance scenario begins with EnterData [3] followed by a call to QueryDatabase on the Server. This is represented in the figure with a synchronous call node. The processing that occurs on the Client after completion of the Query before the Update is represented in the graph with the InitiateOrder node. Next is the synchronous call to UpdateDatabase on the Server. The last processing step is TriggerOrderProcess represented by the asynchronous call node.

The next step is to specify resource requirements for each processing step. The key resources that we examine in this model are the CPU time for the processing, the number of I/Os to the local database (DB), the number of messages sent among processors (Msgs), and the estimated Delay in seconds for the "blips" on other processors until the message-reply is received. These values are also shown in Figure 7.

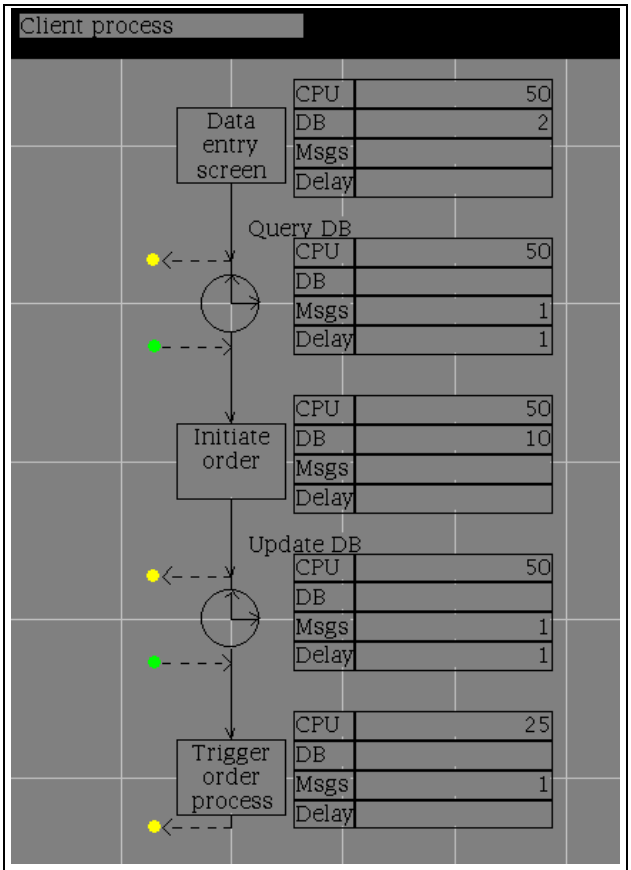

Figure 7. Client process software execution model

4.2 Overhead Matrix

In *SPE•ED*, the computer resource requirements for each software resource request are specified in an overhead matrix stored in the SPE database. Figure 8 shows the overhead matrix for this case study. The software resource names are in the left column of the matrix; the devices in the facility are in the other columns. The device names and quantity are in the top section of the matrix. The CPU, Disk, and Delay devices are unique to this Client facility. The far right column represents the network "device" GlNet. It is shared by all computer processing facilities in the model. The values in the middle part of the matrix specify the amount of

[3] For simplicity, this step is summarized in this model with a single execution graph node. A more realistic model would represent this node with an expanded node and represent its details in a subgraph.

computer device processing required for each software resource request.

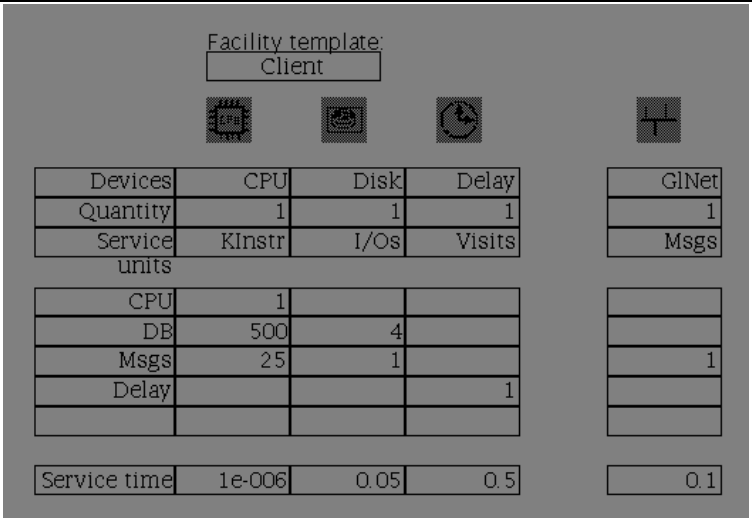

Figure 8. Overhead matrix

For example, the matrix specifies that each Msg in the software model causes one visit to this network device. Its service time per message is 0.1 seconds. Each Msg in the software model also causes 25K instructions to be executed on the CPU and 1 Disk I/O (for logging). The CPU values are in thousands of instructions. Each DB visit specified in the software model causes 500K instructions to be executed on the CPU and 4 I/Os to the Disk device. The delay in the software model is specified in visits; each visit represents a delay of 0.5 seconds. These values are derived from measurement experiments on the target platform, or estimated in a performance walkthrough. They are derived once and reused for all studies of performance scenarios that execute in that environment.

In the client scenario, we estimate a 0.5 second delay for each synchronous call to the Database Server process, and specify that one message is sent via the GlNet device. There is no delay for the asynchronous TriggerOrderProcess, and one message is sent via the GlNet device.

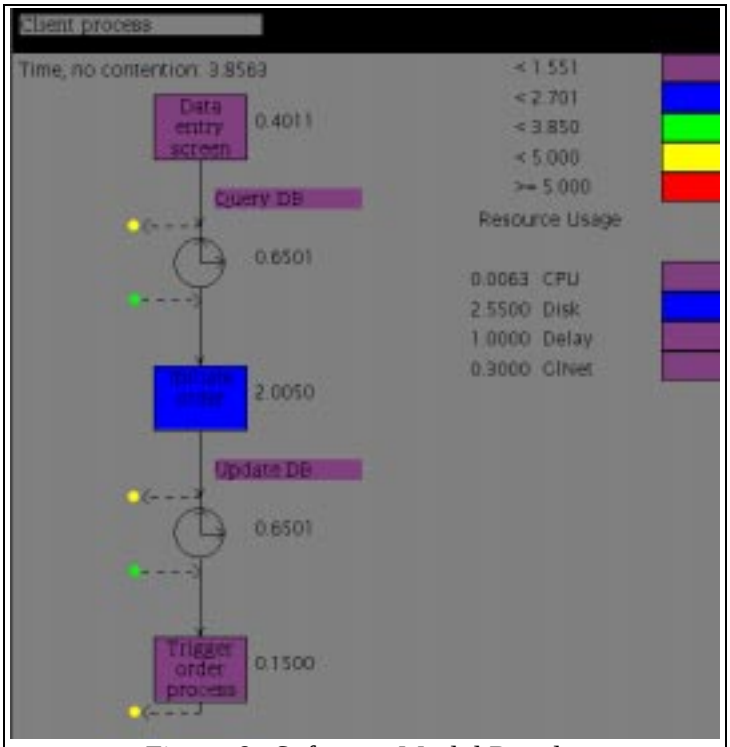

Figure 9. Software Model Results

The model solution in Figure 9 shows that the elapsed time for the Client process with no contention is 3.85 seconds, most of which is for local database processing. The synchronization with other processes is 0.3 seconds for sending messages via the network and an estimated one second delay for the two synchronous calls. There are no particular performance problems.

4.3 Database Server Scenario

Figure 10 shows the processing steps for the second performance scenario derived from theDatabase column. The top-level model has a

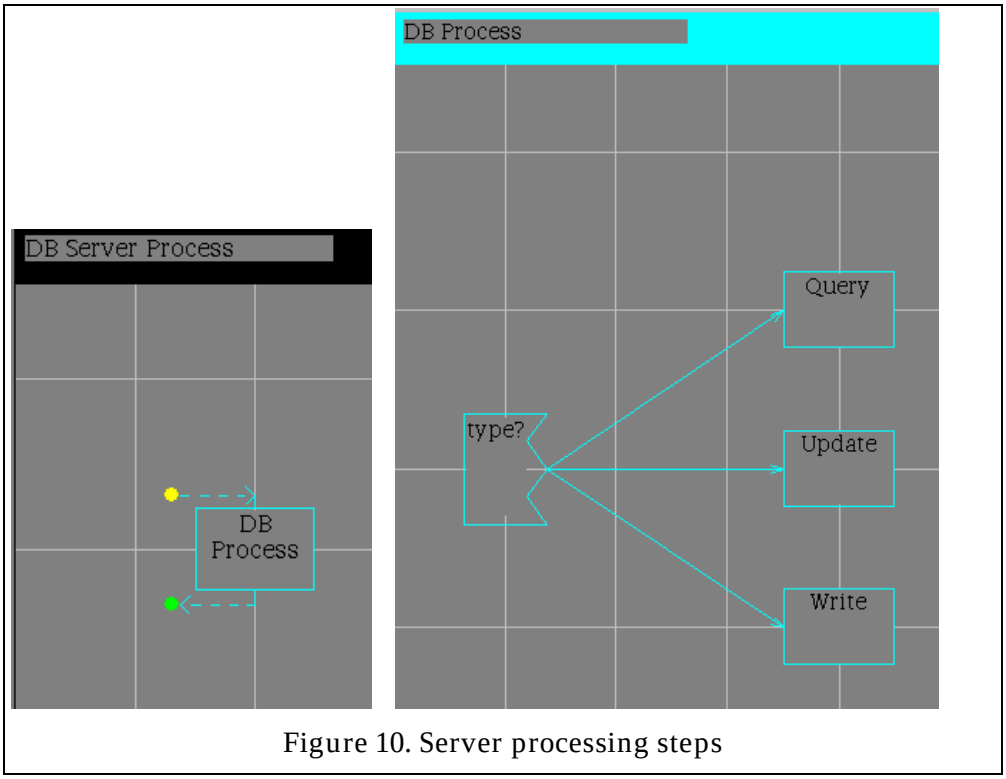

Figure 10. Server processing steps

single node for a called process with a reply. Its basic node is expanded to represent 3 types of calls shown in the subgraph: Query, Update, and Write. In this scenario the probability of a query is 0.5, an update is 0.5 and a write is 0.

The software execution model solution (not shown) results in an elapsed time of 0.24 seconds. The results indicate that this example has no particular performance problem.

To evaluate the effect of contention due to DB Server requests from multiple clients, we will evaluate the system execution model. We will start with one thread for the server process, and increase the number if threads if clients have an excessive wait for the server process to complete previous requests.[4] This will increase device contention because the concurrent processes compete for facility devices. The think time is the estimated time in seconds between requests from clients. It is obtained by estimating the time between the "blips" and dividing that number by the number of clients making requests. In some cases the think time may be calculated from a throughput goal: (1/Tput) is the time between requests. The results (not shown) indicate that with one thread the residence time per request is 0.24 seconds (.1 secs. for network messages, .14 secs for Disk I/O, and a neglible amount of CPU time).

[4] A thread is a copy of the process that executes essentially independently of the other copies. It is not totally independent because calls to the process must be synchronized, they may compete for system resources, and they must lock resources to ensure that concurrent updates do not occur.

4.4 Order Process Scenario

The scenario in Figure 11 represents the column for anOrderProces. It executes as a result of the TriggerOrderProcess asynchronous call in the Client process. This example summarizes the scenario with a single Called process node that sends no results. The scenario also executes on the Server facility. This scenario has one thread with an estimated time between arrivals of 5 sec. The contention solution results in an elapsed time of 1.6 seconds. Again there are no particular performance problems.

Figure 11. ProcessOrder Scenario

4.5 System Execution Model

The system execution model in Figure 12 shows that the Client process executes on the Client facility, and the DB Server Process and Order Process both execute on the Server facility. The system model global specifications to the right of each scenario show the service level specifications for each scenario: the priority and arrival rate or number of threads (users) and the time between arrivals (think).

The solution to the system model shown in Figure 13 shows the response time for each scenario, and the elapsed time at each facility device in the template below the scenario symbol. Note the GlNet device utilization is 4% (shown at the bottom of the facility picture next to the network symbol). This is the combined use of this device by all scenarios. The elapsed time for network communication, including the time in the

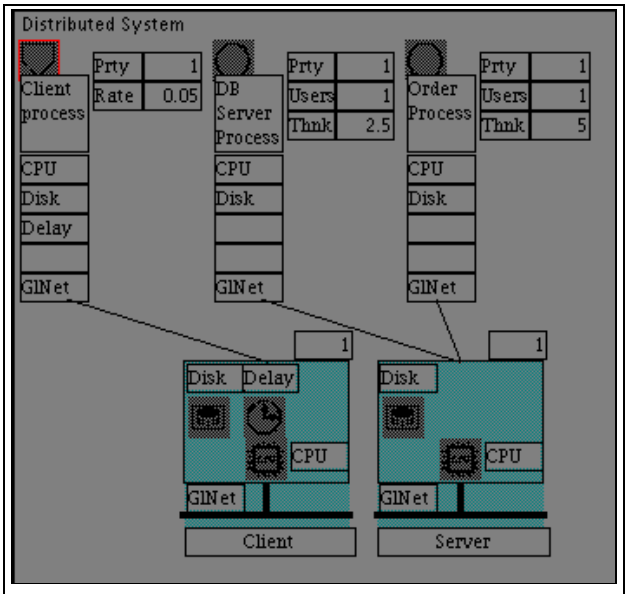

Figure 12. System Execution Model

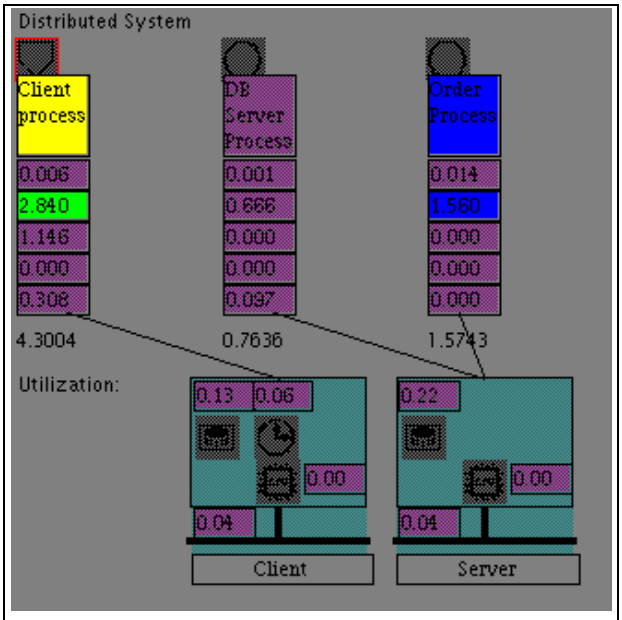

Figure 13. System Execution Model Results

queue, is shown in the bottom row of the facility device template below each scenario: 0.308 sec. for the Client process, 0.097 for the DB Server Process, and 0 for Order Process (there is no reply).

Note that this solution uses the delay estimates in the software model for the time for inter-process synchronization, the solution may be iterative. For example, we estimated the time for the DB server process (excluding network time) to be 0.5 seconds. This solution shows an average time of 0.76 seconds. The advanced system model solution will connect the processes and compute the delay time. These approximate results are generally sufficient to study most software architecture and design alternatives such as the extent of inter-process communication, the assignment of scenarios to facilities, the number of threads required, etc.

4.6 Advanced System Model

The advanced system model solution executes the system model simulation and actually makes calls to other processes at the point in the execution where the special synchronization nodes are placed. If the called process is busy, the calling process waits in a queue. In addition to the standard results reported in the system model solution, the following are reported for the synchronization steps:

- mean, minimum, and maximum, and variance response time for called processes
- mean and maximum number of requests and the mean time in the queue for called processes
- throughput of called processes.

These results indicate when processing requirements should be reduced or the number of threads increased to alleviate performance problems due to synchronization. They also indicate what proportion of the total elapsed time depends on other processes. The system model solution will suffice for most of these analyses early in development. The advanced system model solution gives more insight into situations when mean values may be fine, but queue lengths may build in some circumstances and lead to unacceptable performance. These models are described in detail in the companion paper [SMIT98c].

5. Summary

This paper discussed the strategy for creating simple initial models of distributed system processes. Approximation techniques estimate the synchronization and communication delays. Later in development advanced system execution models provide more realistic performance data for intricate processing details.

The execution graph nodes that represent the calling and called processing steps were explained. Then a simple example demonstrated the process of creating, specifying values for, and evaluating the performance of scenarios that contain these nodes. The solutions demonstrate the distributed system performance data of interest.

A comprehensive case study that includes these approximate models and an advanced system execution model is in the companion paper [SMIT98c]. It is based on an actual case study and substantiates the importance of applying these SPE techniques in the development life cycle when the architecture is formulated.

Another recent paper specifies the information required to perform architecture assessments and how it can be extracted from architectural descriptions [WILL98]. The combination of this work leads us to the conclusion that much of the needed information is not contained in current architectural documentation and must be gathered in a performance walkthrough or using other supplemental techniques. We are also convinced that for distributed systems much of this information, particularly the assignment of processing steps to processes and of processes to computer facilities, and whether or not multiple threads are necessary, should be determined by performance assessments and the documentation should be produced afterwards. Thus SPE should play a key role in the development of distributed systems from the early planning stages and throughout development.

References

[BEIL95] H. Beilner, J. Mäter, and C. Wysocki, *The Hierarchical Evaluation Tool HIT*, 581/1995, Universität Dortmund, Fachbereich Informatik, D-44221 Dortmund, Germany, 1995, 6-9.

[CLEM96] P.C. Clements and L.M. Northrup, Software Architecture: An Executive Overview, 1996.

[GOET90] Robert T. Goettge, "An Expert System for Performance Engineering of Time-Critical Software," *Proceedings Computer Measurement Group Conference*, Orlando FL, 1990,313-320.

[ITU96] ITU, Criteria for the Use and Applicability of Formal Description Techniques, Message Sequence Chart (MSC), 1996.

[JACO92] I. Jacobsen, et al., *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA, 1992.

[RATI97] Rational Software Corporation, Unified Modeling Language: Notation Guide, Version 1.1, 1997.

[RUMB91] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

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

[SMIT98] C.U. Smith and L.G. Williams, "Software Performance Engineering for Object-Oriented Systems: A Use Case Approach," *submitted for publication*, , 1998.

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

[TURN92] Michael Turner, Douglas Neuse, and Richard Goldgar, "Simulating Optimizes Move to Client/Server Applications," *Proceedings Computer Measurement Group Conference*, Reno, NV, 1992,805-814.

[WILL94] Lloyd G. Williams, Definition of the Information Requirements for Software Performance Engineering, 1994.

[WILL95] Lloyd G. Williams and Connie U. Smith, "Information Requirements for Software Performance Engineering," *Proceedings 1995 International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Heidelberg, Germany, 1995,.

[WILL98] Lloyd G. Williams and Connie U. Smith, "Performance Evaluation of Software Architectures," *Proc. First International Workshop on Software and Performance*, Santa Fe, NM, October 1998,.