PERFORMANCE AND SCALABILITY OF DISTRIBUTED SOFTWARE ARCHITECTURES: AN SPE APPROACH

CONNIE U. SMITH* AND LLOYD G. WILLIAMS†

Abstract. Distributed systems were once the exception, constructed only rarely and with great difficulty by developers who spent significant amounts of time mastering the technology. Now, as modern software technologies have made distributed systems easier to construct, they have become the norm. Unfortunately, many distributed systems fail to meet their performance objectives when they are initially constructed. Others perform adequately with a small number of users but do not scale to support increased usage. These performance failures result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows.

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 phase. This paper discusses assessment of the performance characteristics of distributed software architectures using the Software Performance Engineering (SPE) approach. We describe the information required to perform such assessments, particularly the information about synchronization points and types of synchronization mechanisms, and the modeling approach. The case study demonstrates how to construct performance models for distributed systems and illustrates how simple models of software architectures are sufficient for early identification of performance problems.

Key words. performance, software performance engineering, software architecture, software quality, object-oriented development, performance modeling, performance engineering tools, $SPE{\bullet}ED$ performance modeling tool.

1. Introduction. Distributed systems were once the exception, constructed only rarely and with great difficulty by developers who spent significant amounts of time mastering the technology. Now, as modern software technologies have made distributed systems easier to construct, they have become the norm. The explosion of web applications is just the tip of the distributed systems iceberg. Distributed applications include embedded real-time systems, client/server systems, traditional business applications and integrated legacy systems.

Unfortunately, many distributed systems fail to meet their performance objectives when they are initially constructed. Others perform adequately with a small number of users but do not scale to support increased usage. These performance failures result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, and missed market windows. In the case of web applications, these failures can also be embarrassingly public.

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 phase. Poor performance is more often the result of problems in the architecture or design rather than 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." [8]

This means that performance problems are actually introduced early in the development process. However, most organizations ignore performance until integration testing or later. With pressure to deliver finished software in shorter and shorter times, their attitude is: "Let's get it done. If there is a performance problem, we'll fix it later." Thus, performance problems are not discovered until late in the development process, when they are more difficult (and more expensive) to fix.

The following quote from Auer and Beck is typical of this "fix-it-later" attitude:

Performance myth: "Ignore efficiency through most of the development cycle. Tune performance once the program is running correctly and the design reflects your best understanding of how the code should be structured. The needed changes will be limited in scope or will illuminate opportunities for better design." [2]

"Tuning" code to improve performance is likely to disrupt the original design, negating the benefits obtained from a carefully crafted architecture. It is also unlikely that "tuned" code will

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

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

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.

The "fix-it-later" attitude is rooted in the view that performance is difficult to predict and that the models needed to predict the performance of software (particularly a distributed system) are complex and expensive to construct. Predicting the performance of a distributed system can, in fact, be difficult. The functionality of distributed systems is often decentralized. Performing a given function is likely to require collaboration among components on different nodes in the network. If the interactions use middleware, such as CORBA, the models can become even more complex.

Despite these difficulties, our experience is that it is possible to cost-effectively engineer distributed systems that meet performance goals. By carefully applying the techniques of software performance engineering (SPE) throughout the development process, it is possible to produce distributed systems that have adequate performance and exhibit other desirable qualities, such as reusability, maintainability, and modifiability.

SPE is a method for constructing software systems to meet performance and scalability objectives [27]. Performance refers to the response time or throughput as seen by the users. The SPE process begins early in the software life cycle and uses quantitative methods to identify a satisfactory architecture and to eliminate those that are likely to have unacceptable performance. SPE continues throughout the development process to: predict and manage the performance of the evolving software, monitor actual performance against specifications, and report and manage problems if they are identified. SPE begins with deliberately simple models that are matched to the current level of knowledge about the emerging software. These models become progressively more detailed and sophisticated, as more details about the software are known. SPE methods also cover performance data collection, quantitative analysis techniques, prediction strategies, management of uncertainties, data presentation and tracking, model verification and validation, critical success factors, and performance design principles, patterns, and antipatterns.

This paper illustrates the construction of simple performance models of distributed systems. Model solutions are computed using the tool $SPE{\bullet}ED^{(tm)}$ [1]. $SPE{\bullet}ED$ is a performance modeling tool that supports the SPE process described in [27][31]. $SPE{\bullet}ED$s focus on software processing and automatic model generation make it easy to evaluate distributed architecture and design alternatives. Other features, such as the SPE project database and presentation and reporting features, support aspects of the SPE process in addition to modeling.

We begin with a review of related work. We then present an overview of the SPE models and a description of the $SPE{\bullet}ED$ tool. A case study illustrates how to use the process and models to evaluate distributed system architecture alternatives.

2. Related Work. This work brings together three separate lines of inquiry. The first is the evaluation of the performance of distributed systems. The second is the evaluation of software architectures, and the third is the use of software model notations to create performance models. The synthesis is a focus on the early evaluation of distributed system architectures as a means of identifying and mitigating performance risks.

Work on the performance of distributed systems and web applications has generally focused on the system point of view. Examples include: [9][13][20][33][36]. These approaches focus on the performance of the system as a whole and evaluate the overall performance of the system. Our approach focuses on the software architecture and evaluates the overall effect of alternative software architectures. This means that our models must explicitly represent software processing steps that reflect the interactions among distributed systems. Both Hills et. al. [14] and Szumilas et. al, [32] used a slightly different system approach. They looked at implementation architecture alternatives such as the assignment of processes to processors, and identifying a feasible strategy for achieving performance objectives.

Some earlier work has addressed synchronization from the software point of view: [24][23][27][28]. All these approaches modeled only synchronous communication whereas this work addresses 4 typical types of synchronization. Our approach also differs in its use of a hybrid modeling approach: simple approximation techniques for early analysis combined with simulation solutions for later analysis of

more detailed models.

Scratchley and Woodside have proposed a similar approach to evaluate concurrency options in software specifications. They add performance annotations to use case maps, and generate a virtual implementation model from them [26]. The generated models can be used to study concurrency design alternatives such as multi-processing and multi-threading, and the feasibility of meeting quality of service objectives. This is a powerful technique for generating and evaluating advanced models of concurrency for systems that match their design paradigm, such as concurrent communication applications. They do not provide the simpler, approximate models for early system evaluation.

Work on evaluation of software architectures has focused on early evaluation of software architectures to reveal problems at a point in the software development process where they can be corrected most easily and economically. Notable in this area is the work of Kazman and co-workers [16][17]. Their approach is scenario-based and considers various stakeholders in the system (e.g., users, system administrators, maintainers) and develops usage scenarios from their various points of view. Their scenarios are expressed informally as brief textual descriptions that capture uses of the system that are related to quality attributes, such as ease of modification. The architecture is then evaluated on how much effort is required to satisfy the scenarios.

Our work differs from that of Kazman and co-workers in its focus on performance and its use of more rigorous scenario descriptions. While Kazman, et. al., apply their technique to a variety of quality attributes, including performance, as noted above, they use informal, natural language descriptions of scenarios.

Other, related, work on performance evaluation of software architectures includes that of Balsamo, et. al. [3] and Lung, et. al. [19]. The work of Balsamo, et. al. compares the relative performance of software architectures specified in the Chemical Abstract Machine (CHAM) formalism. Queueing model parameters (the arrival rate distributions and service time distributions) are represented symbolically and the model solutions provide relative performance measures for the architectures they consider. By contrast, our approach provides a framework for estimating the model parameters, so it produces quantitative results rather than symbolic results. Lung, et.al. propose a process similar to that described in [29] that is adapted to their specific architecture notations, and describe results achieved using it.

Work on the creation of performance models from design notations includes [18], [21], [29]. All these approaches require extensions to include the performance specifications. The models produced differ in their ability to predict distributed system performance.

This paper illustrates model solutions using the $SPE{\bullet}ED$(tm) performance engineering tool [1][29][30]. A variety of other performance modeling tools are available, such as [5][4][11][12][22][34]. However, the approach described here will need to be adapted for tools that do not use execution graphs as their modeling paradigm.

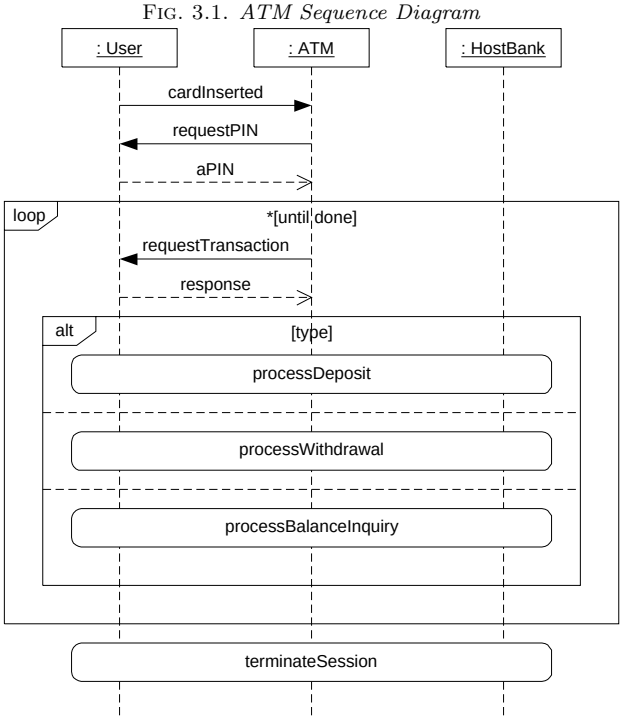
3. SPE Models for Distributed Systems.

3.1. Scenarios. The SPE process begins with the system's use cases [6]. Here we focus on the scenarios that describe the use cases. A scenario is a sequence of actions describing the interactions between the system and its environment (including the user) or between the internal objects involved in a particular execution of the system. The scenario shows the objects that participate and the messages that flow between them. A message may represent either an event or an invocation of one of the object's methods (operations). Performance scenarios are the subset of the use case scenarios that are executed frequently, or those that are critical to the perceived performance of the system. We use Unified Modeling Language (UML) sequence diagrams, augmented with features from the message sequence chart (MSC) standard [15], to represent performance scenarios.

To illustrate the notation, we will use a simple automated teller machine (ATM). Figure 3.1 illustrates a high-level scenario for the ATM. In a UML sequence diagram, each object that participates in the scenario is represented by a vertical line or axis. The axis is labeled with the object name (e.g., anATM). The vertical axis represents relative time that increases from top to bottom (sequence diagrams do not use a representation of absolute time). The horizontal arrows represent interactions between objects (events or operation invocations).

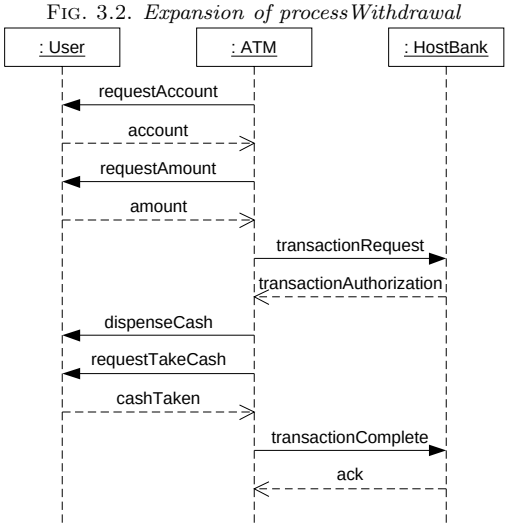
The rectangular areas in Figure 3.1 labeled "loop" and "alt" are from the MSC standard. They

FIG. 3.1. *ATM Sequence Diagram*



denote repetition and alternation. The scenario indicates that the user may repeatedly select a transaction, which may be a deposit, a withdrawal, or a balance inquiry. The rounded rectangles (also from the MSC standard) are "references" which refer to other sequence diagrams. The use of references allows horizontal decomposition of scenarios. The sequence diagram corresponding to processWithdrawal is shown in Figure 3.2.

FIG. 3.2. *Expansion of processWithdrawal*



Additional extensions to the sequence diagram notation are in [31]. We conclude this section with some extensions for modeling synchronization in distributed systems. We will focus on four types of communication and synchronization that are typically supported in middleware: * synchronous, * asynchronous, * deferred synchronous, and * asynchronous callback communication.

You can show synchronous and asynchronous messages in the UML using different types of

arrowheads. Figure 3.3 shows a synchronous communication using a filled arrowhead for the message and a return (dashed) arrow for the reply. Figure 3.4 shows an asynchronous communication using a half-stick arrowhead. Both of these examples use standard UML notation.

FIG. 3.3. *Synchronous Communication*

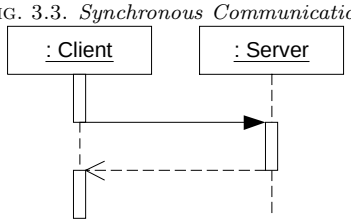
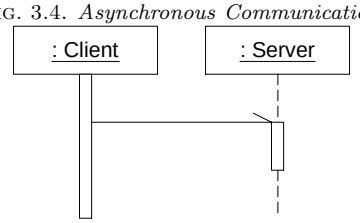
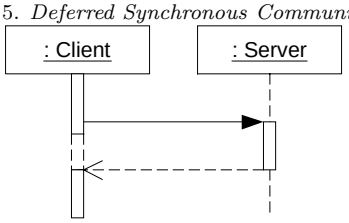


FIG. 3.4. *Asynchronous Communication*



We have also found it useful to model deferred synchronous communication in distributed systems (for example CORBA-based systems). This type of communication is similar to a synchronous interaction in that the client sends a message to the server and expects a reply. In this case, however, the client sends the message and continues processing. Then it requests the result later. This type of interaction is shown in Figure 3.5 using an extension to the sequence diagram notation. The extension is the addition of a dashed section of the activation bar to show that the client has a potential delay while the server finishes responding to the request.

FIG. 3.5. *Deferred Synchronous Communication*

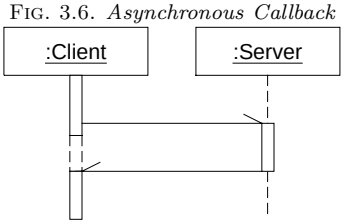


Similar behavior can be achieved in systems that do not support deferred synchronous communication with asynchronous callback as in Figure 3.6. In this case an asynchronous call is sent and the client continues processing. When the server completes the request it sends another asynchronous call to the client.

3.2. SPE Model Overview. Software performance engineering is a quantitative approach to constructing software systems that meet performance objectives. It incorporates models for representing and predicting performance as well as a set of analysis methods, techniques for gathering data, and other steps mentioned earlier. SPE uses deliberately simple models of software processing with the goal of using the simplest possible model that identifies problems with the system architecture, design, or implementation plans. These models are easily constructed and solved to provide feedback on whether the proposed software is likely to meet performance goals. As the software process proceeds, the models are refined to more closely represent the performance of the emerging software.

The precision of the model results depends on the quality of the estimates of resource requirements. Because these are difficult to estimate early in the software process, SPE uses adaptive

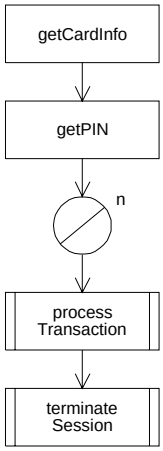
FIG. 3.6. *Asynchronous Callback*



strategies, such as upper- and lower-bounds estimates and best- and worst-case analysis to manage uncertainty. For example, when there is high uncertainty about resource requirements, analysts use estimates of the upper and lower bounds of these quantities. Using these estimates, analysts produce predictions of the best-case and worst-case performance. If the predicted best-case performance is unsatisfactory, they seek feasible alternatives. If the worst-case prediction is satisfactory, they proceed to the next step of the development process. If the results are somewhere in-between, analyses identify critical components whose resource estimates have the greatest effect and focus on obtaining more precise data for them. A variety of techniques can provide more precision, including: further refining the design and constructing more detailed models or constructing performance benchmarks and measuring resource requirements for key components.

Two types of models provide information for design assessment: the software execution model and the system execution model. The software execution model represents key aspects of the software execution behavior. It is constructed using an execution graph [27] to represent each performance scenario. Nodes represent components of the software; arcs represent control flow. The graphs are hierarchical with the lowest level containing complete information on estimated resource requirements. Figure 3.7 shows the execution graph corresponding to the user interaction scenario from Figures 3.1 and 3.2. The graph shows that, following getCardInfo and getPIN, the ATM will repeat the processTransaction node n times. Both processTransaction and terminateSession are expanded nodes; they are expanded in a separate graph. Figure 3.8 shows the expansion of processTransaction.

FIG. 3.7. *ATM Execution Graph*

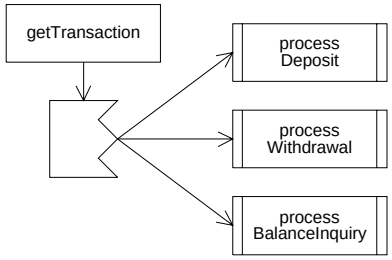


Execution graphs for distributed system models have other types of nodes to represent the synchronization points and the type of synchronization. They are illustrated in the case study.

Solving the software model provides a static analysis of the mean, best- and worst-case response times. It characterizes the resource requirements of the proposed software alone, in the absence of other workloads, multiple users or delays due to contention for resources. If the predicted performance in the absence of these additional performance-determining factors is unsatisfactory, then there is no need in constructing more sophisticated models.

If the software execution model indicates that there are no problems, analysts proceed to construct and solve the system execution model. This model is a dynamic model that characterizes the

FIG. 3.8. *Expansion of processTransaction*



software performance in the presence of factors, such as other workloads or multiple users that could cause contention for resources. The results obtained by solving the software execution model provide input parameters for the system execution model. Solving the system execution model provides the following additional information:

- more precise metrics that account for resource contention
- sensitivity of performance metrics to variations in workload composition
- effect of new software on service level objectives of other systems
- identification of bottleneck resources
- comparative data on options for improving performance via: performance scenario changes, software changes, hardware upgrades, and various combinations of each

The system execution model represents the key computer resources as a network of queues. Queues represent components of the environment that provide some processing service, such as processors or network elements. Environment specifications provide device parameters (such as CPU size and processing speed). Workload parameters and service requests for the proposed software come from the resource requirements computed by solving the software execution model. The results of solving the system execution model identify potential bottleneck devices and correlate system execution model results with software components.

If the model results indicate that the performance is likely to be satisfactory, developers proceed. If not, the model results provide a quantitative basis for reviewing the proposed design and evaluating alternatives. Feasible alternatives can be evaluated based on their cost-effectiveness. If no feasible, cost-effective alternative exists, performance goals may need to be revised to reflect this reality.

This discussion has outlined the SPE process for one early design-evaluation cycle. These steps repeat throughout the development process. At each phase, the models are refined based on the more detailed design and analysis objectives are revised to reflect the concerns that exist for that phase [27]. Early models approximate delays for synchronization among distributed processes. Later, advanced system execution models quantify the synchronization delays and provide additional data for evaluating the distributed systems.

3.3. Deriving Execution Graphs from Sequence Diagrams. The performance analysis techniques, as well as the *SPE•ED* tool (Section 4), are based on execution graphs. Thus, a key step in the SPE process is the derivation of execution graphs from sequence diagrams. Currently, this is a manual process. The close correspondence between sequence diagrams and execution graphs suggests that an automated translation might be possible, however.

For single-threaded scenarios or scenarios with sequential flow of control, going from a sequence diagram to an execution graph is straightforward. For scenarios that involve multiple threads of control or distributed objects, a little more effort is needed to identify operations that serialize and account for communication and synchronization delays. In either case, the process of translating a sequence diagram to an execution diagram is similar.

Each message received by an object triggers an action - either an operation or a state machine transition. The simplest way to construct an execution graph from a sequence diagram is to follow the message arrows through the performance scenario and make each action a basic node in the execution graph. However, in many cases, individual actions are not interesting from a performance perspective and several of them may be combined into a single basic node. Alternatively, you can

use an expanded node to summarize a series of actions and provide details of the sequence of actions in its subgraph.

If you use the MSC extensions discussed in Section 3.1, repetition and case nodes are easy to identify. If not, you will need to walk through the scenario to identify repetitions. To find alternative processing steps, you will probably need to look at different scenarios from the same use case that represent alternative uses of the system.

A reference is most easily rendered as an expanded node with the execution graph corresponding to the sequence diagram that it points to in the subgraph.

4. $SPE{\bullet}ED$ Overview. This section provides a brief overview of the features of the SPE tool, $SPE{\bullet}ED$, that make it appropriate for distributed (and other) system evaluations throughout their development cycle.

4.1. Focus. $SPE{\bullet}ED$'s focus is the software performance model. Users create graphical models of envisioned software processing and provide performance specifications by creating and specifying execution graphs. Queueing network models are automatically generated from the software model specifications. A combination of analytic and simulation model solutions identify potential performance problems and software processing steps that may cause the problems. $SPE{\bullet}ED$ facilitates the creation of (deliberately) simple models of software processing with the goal of using the simplest possible model that identifies problems with the software architecture, design, or implementation plans. Simple models are desired because in the early life cycle phase in which they are created: * developers seldom have exact data that justifies a more sophisticated model, * they need quick feedback to influence development decisions, * they need to comprehend the model results, especially the correlation of the software decisions to the computer resource impacts.

4.2. Model description. Users create the model with a graphical user interface streamlined to quickly define the software processing steps. The user's view of the model is a scenario, an execution graph of the software processing steps [27]. Software scenarios are assigned to the facilities that execute the processing steps. Models of distributed processing systems may have many scenarios and many facilities.

Users specify software resource requirements for each processing step. Software resources may be the number of messages transmitted, the number of SQL queries, the number of SQL updates, etc. depending on the type of system to be studied and the key performance drivers for that system. A performance specialist provides overhead specifications that specify an estimate of the computer resource requirements for each software resource request. These are specified once and re-used for all software analysis that executes in that environment.

4.3. Model solution. $SPE{\bullet}ED$ produces analytic results for the software models, and an approximate, analytic mean-value-analysis solution of the generated queueing network model. A simulation solution is used for generated queueing network models with multiple software scenarios executing on one or more computer system facilities. $SPE{\bullet}ED$ uses an embedded version of Mesquite Software's CSIM modeling tool to solve the models [25]. Thus $SPE{\bullet}ED$ supports hybrid solutions - the user selects the type of solution appropriate for the development life cycle stage and thus the precision of the data that feeds the model. There is no need for a detailed, lengthy simulation when only rough guesses of resource requirements are specified.

4.4. Model results. The results reported by $SPE{\bullet}ED$ are the end-to-end response time, the elapsed time for each processing step, the device utilization, and the amount of time spent at each computer device for each processing step. This identifies both the potential computer device bottlenecks, and the portions of the device usage by processing step (thus the potential software processing bottlenecks).

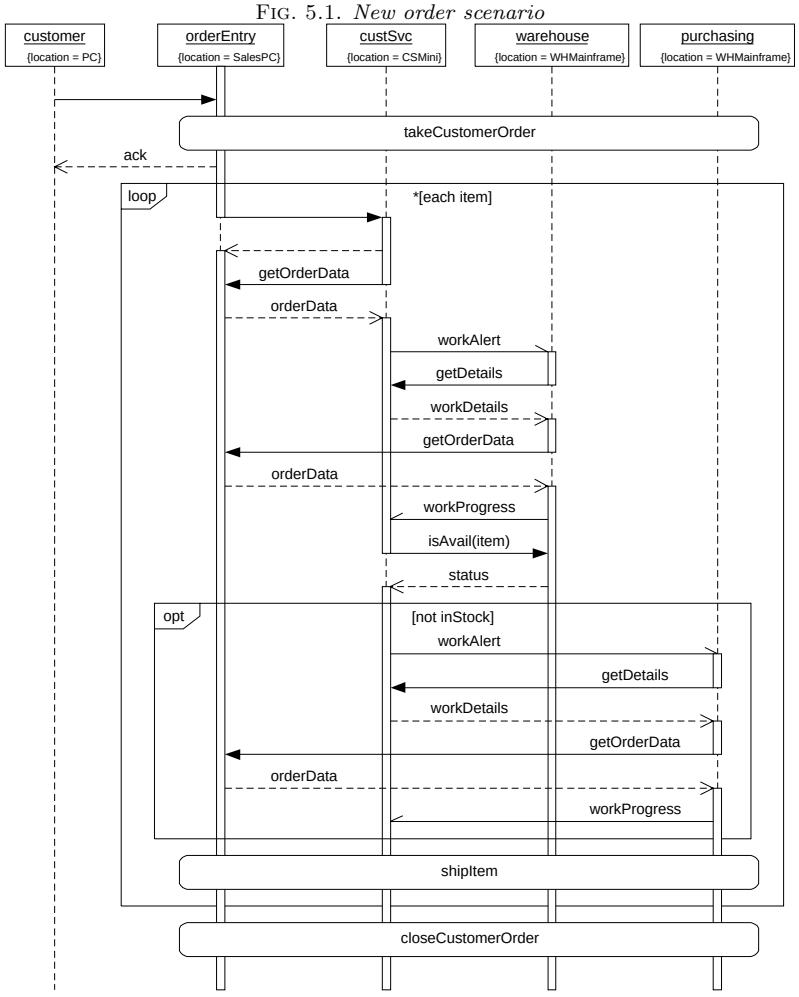
Model results are presented both with numeric values and color coding that uses cool colors to represent relatively low values and hot colors (yellow and red) calling attention to relatively high values. Up to four sets of results may be viewed together on a screen. This lets users view any combination of performance metrics for chosen levels in the software model hierarchy, and even compare performance metrics for design or implementation choices. An export feature lets users

copy model results and paste them into word processing documents and presentation packages, or write out results for charting packages to create custom charts for reports.

4.5. Application areas. *SPE•ED* is intended to model software systems under development, although it may also be used for existing software systems. It may be any type of software: all types of software applications including web applications, operating systems, or database management systems. The software may execute on any hardware/software platform combination. The software may execute on a uniprocessor or in a distributed or client/server environment.

5. Case Study. This case study is from an actual study, however, application details have been changed to preserve anonymity. The software supports an electronic virtual storefront, eStuff.1 Software supporting eStuff has components to take customer orders, fulfill orders and ship them from the warehouse, and, for just-in-time shipments, interface with suppliers to obtain items to ship. The heart of the system is the Customer Service component that collects completed orders, initiates tasks in the other components, and tracks the status of orders in progress.

FIG. 5.1. *New order scenario*



5.1. Approximate Model. The use case we consider is processing a new order (Figure 5.1). The performance objective for the scenario is to complete the order processing in 30 seconds and to support an arrival rate of 1 order every 10 seconds.

It begins with TakeCustOrder, a reference to another, more detailed sequence diagram. An ACK is sent to the customer, and the order processing begins. In this scenario we assume that a customer order consists of 50 individual items. The unit of work for the TakeCustOrder and

CloseCustOrder components is the entire order; the other order-processing components handle each item in the order separately; the sequence diagram shows this repetition with a loop symbol. The similar symbol labeled "opt" represents an optional step that may occur when eStuff must order the item from a supplier.

Each column in Figure 5.1 represents an independent process, so the columns are modeled separately. The first three columns each execute on their own facility; the warehouse and purchasing processes share a facility.

FIG. 5.2. *Execution Graph:* **ProcessItemOrder Expansion**

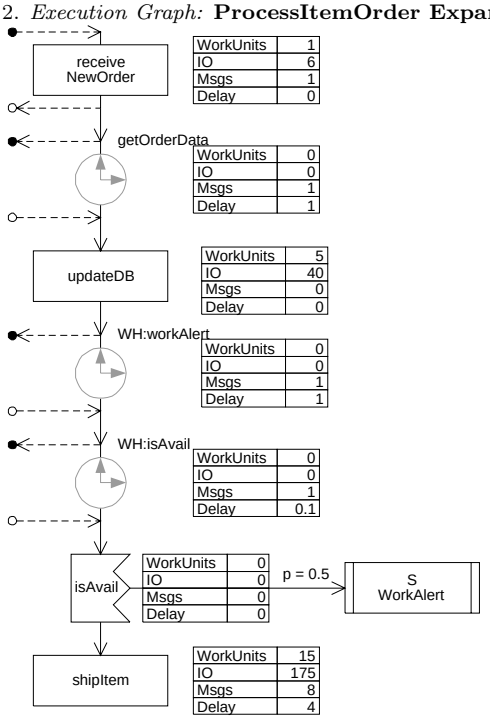


Figure 5.3 shows the execution graph for the CustomerService column in the scenario in Figure 5.1. Everything inside the loop is in the expanded node, ProcessItemOrder. Its details are in Figure 5.2. Figure 5.2 shows the synchronization nodes in the ProcessItemOrder step. It first receives the NewOrder message and immediately replies with the acknowledgement message thus freeing the calling process. Next it makes a synchronous call to GetOrderData and waits for the reply. The availability check is modeled with a synchronous call to WH:IsAvail. The two WorkAlert processing steps are also expanded; their details are not shown here. This software model depicts only the CustomerService processing node; we approximate the delay time to communicate with the other processing nodes.

The next step is to specify resource requirements for each processing step. The key resources in this model are the CPU time for database and other processing (specified in WorkUnits), the number of I/Os for database and logging activities, the number of messages sent among processors (and the associated overhead for the middleware), and the estimated delay in seconds until the message-reply is received. These values are shown in Figures 5.3 and 5.2. They are best-case estimates derived from performance measurement experiments. They may also be estimated in a performance walkthrough [27].

Analysts specify values for the software resource requirements for processing steps. The computer resource requirements for each software resource request are specified in an overhead matrix stored in the SPE database. This matrix represents each of the hardware devices in each of the distributed processors, connects the software model resource requests to hardware device requirements, and incorporates any processing requirements due to operating system or network overhead (see [29] for a detailed description of the overhead matrix).

FIG. 5.3. *Execution Graph:* **CustomerService: New Order**

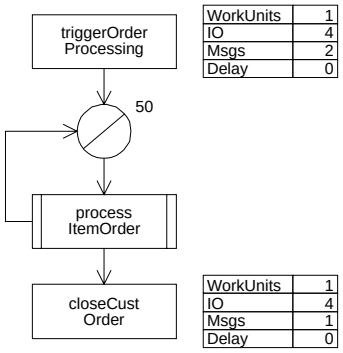
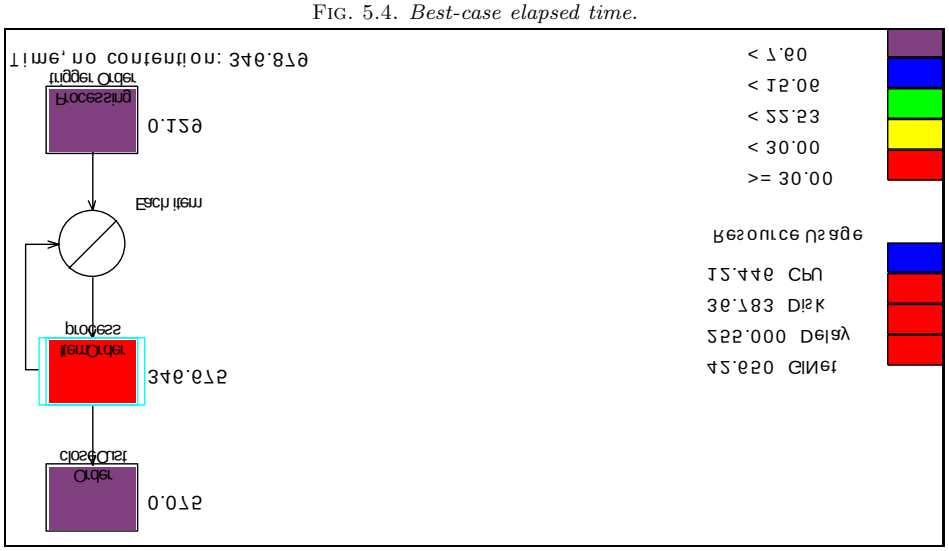


Figure 5.4 shows the best-case solution with one user and thus no contention for computer devices. The overall time is at the top, and the time for each processing step is next to the step. The color bar legend in the upper right corner shows the values associated with each color; defining an overall performance objective sets the upper bound. Values higher than the performance objective will be red, lower values are respectively cooler colors. The 'Resource usage' values below the color bar legend show the time spent at each computer device. Of the 347 total seconds for the end-to-end scenario, 255 seconds is due to the delay for customer interactions and delay for distributed processes.

FIG. 5.4. *Best-case elapsed time.*



The end-to-end time is 347 seconds, most of it is in ProcessItemOrder (the value shown is for all 50 items). Results for the ProcessItemOrder subgraph (not shown) indicate that processing for each item takes approximately 6.9 seconds, most of that is in the ShipItem processing step. Other results (not shown) indicate that 5.1 seconds of the 6.9 seconds is due to estimated delay for processing on the other facilities. Investigation also shows that network congestion prevents the system from supporting the desired throughput. These results reflect the elapsed time without contention (i.e. with only one user).

From these results we calculate the limits on scalability by calculating the maximum arrival rate that the system can support:

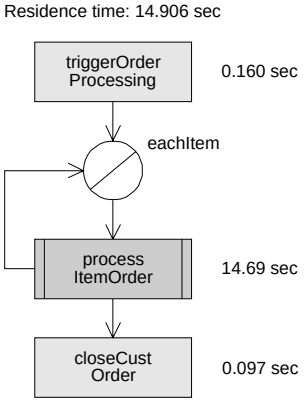
1. Find the device with the maximum resource usage (excluding delay devices) - In this case it is the network, GlNet with 42.65 seconds.

2. The maximum arrival rate is the reciprocal of its demand - 1/42.65 seconds or 0.023 orders per second.

Clearly the system will not scale to support the desired arrival rate 0.1 orders per second.

Analysts can evaluate potential solutions by modifying the software execution model to reflect architecture alternatives. The architecture of this system is largely determined by the constraint that the order entry, warehouse, and purchasing components are existing legacy applications. Thus the architecture is essentially a Stove Pipe [7]. The communication follows the Observer pattern [10]. It uses a lightweight communication: a workAlert message is sent notifying the receiver that there is work to be done, but without specifics of the task. The receiver must request information from customer service about the details of the work (getDetails), then request information from order entry on the specifics of the order to be processed (getOrderData). One alternative is to send the details of the work and the order along with the workAlert. This would eliminate many of the requests, but the messages would be larger. The change to the software model is relatively simple: we delete the synchronous calls for getOrderData and getDetails, and increase the time to transmit the messages. This alternative reduces the elapsed time for one user to 331 seconds, which is still not acceptable. It improves scalability because the network demand is reduced to 30.5 seconds, but not enough to meet the performance objective.

FIG. 5.5. *Revised System at 0.1 jobs per second*



We select another architecture alternative that processes work orders as a group rather than individual items in an order. The changes to the software execution model for this alternative are relatively minor - the number of loop repetitions is reduced to 2 (one for orders ready to ship, the other for orders requiring back-ordered items), and the resource requirements for steps in the loop change slightly to reflect requirements to process a group of items. This alternative yields a response time of 15 seconds with the desired throughput of 0.1 jobs per second as shown in Figure 5.5.

Thus, the overhead and delays due to process coordination were a significant portion of the total end-to-end time to process a new order. Improvements resulted from processing batches of items rather than individual items. These simple models provide sufficient information to identify problems in the architecture before proceeding to the advanced system execution model. It is easy and important to resolve key performance problems with simple models before proceeding to the advanced models described next.

5.2. Advanced System Execution Model. This section illustrates the creation and evaluation of the detailed models of synchronization. Figure 5.1 shows that all the synchronization steps are either asynchronous or synchronous. Deferred synchronous calls and asynchronous callback are only useful when the results are not needed for the next processing steps. Deferred synchronous calls and asynchronous callback may also be more complex to implement. Thus, it is sensible to plan synchronous calls unless the models indicate that deferred synchronous calls result in significant improvements.

Figure 5.6 shows the processing that occurs on the Warehouse facility. It receives the asynchronous request from the CS facility, makes a synchronous call to CS:getDetails, makes a synchronous

FIG. 5.6. *WH:WorkAlert*

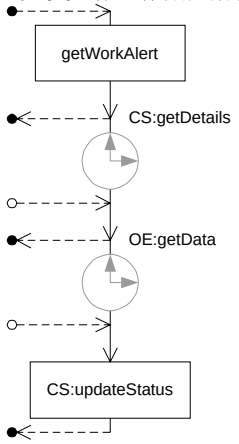


TABLE 5.1
Advanced System Model Results

Scenario	Response Time (Sec.)				TPut	Queue		
	Mean	Min	Max	Variance		Mean	Max	Time
CS:NewOrder	8.2	0.1	64.5	50.2	0.2			
CS:NewOrder(NA)	8.6	0.1	72.8	51.4	0.2			
OE:OrderData	.2	0	4.0	0.1	1.0	0.304	8	0.31
CS:WorkDetails	.2	0	4.3	0.1	0.6	0.160	2	0.27
CS:UpdStatus	.2	0	4.7	0.1	0.6	0.014	4	0.02
WH:WorkAlert	1.8	0.1	11.6	1.6	0.4	1.741	28	4.40
P:WorkAlert	2.0	0.1	13.1	1.8	0.2	0.217	9	1.10

call to OE:getData, then (after the order is shipped) makes an asynchronous call to CS StatusUpdate.

Table 5.2 shows the advanced system model results. The maximum queue length and the queue time for WH:WorkAlert suggests that more concurrent threads might be desirable for scalability. The simulation results can also reflect problems due to "lock-step" execution of concurrent processes. For example, note that the mean response time for P:WorkAlert is slightly higher than for WH:WorkAlert even though they execute the same processing steps and P:WorkAlert executes less frequently (see throughput values). This is because the asynchronous calls to WH:WorkAlert and to P:WorkAlert occur very close to the same time and cause both processes to execute concurrently on the same facility. This introduces slight contention delays for the process that arrives second (P:WorkAlert). In this case study the performance effect is not serious, but it illustrates the types of performance analysis important for this life cycle stage and the models that permit the analysis.

It is difficult to validate models of systems that are still under development. Many changes occur before the software executes and may be measured, and the early life cycle models are best-case models that omit many processing complexities that occur in the ultimate implementation. Nevertheless, the results are sufficiently accurate to identify problems in the software plans and quantify the relative benefit of improvements. In this study, the models successfully predicted potential problems in the original architecture due to network activity.

Note that most of the useful results in early life cycle stages come from the approximate software model. For example, the amount of communication and the synchronization points in the architecture and design, the assignment of methods to processes, assignment of processes to processors, an approximate number of threads per process, etc., can all be evaluated with the simpler models. Likewise, it is easier to evaluate the simpler, visual results than a complex table like Table 5.2 to identify problems and potential solutions. These are the reasons that the SPE approach advocates the use of simple models early in development.

6. Summary and Conclusions. This paper has described a systematic approach to the performance engineering of distributed software systems. A systematic approach to performance is critical to preventing performance failures. Performance failures may result in damaged customer relations, lost productivity for users, lost revenue, cost overruns due to tuning or redesign, missed market windows. In the worst case, it may be necessary to completely re-design the product or even cancel the project.

Distributed systems offer unique challenges for performance engineering due to the complexity of interactions between components and the use of middleware. However, our experience has shown that it is possible to cost-effectively engineer distributed systems that meet performance goals. This paper has described the process of software performance engineering for distributed systems and illustrated the process with a simple case study.

The case study demonstrated how to construct performance models for distributed systems and illustrated that simple models of software architectures are sufficient for early identification of performance problems. It demonstrated that it is relatively quick and easy to construct the software performance models with an SPE tool such as $SPE{\bullet}ED$. It also showed that it is easy to evaluate architecture and design tradeoffs, and to evaluate system scalability before the software construction begins.

This work is part of a larger project to make it easier for developers to perform initial performance assessments. One of the principal barriers to the widespread acceptance of SPE is the gap between the software developers who need performance assessments and the performance specialists who have the skill to conduct comprehensive performance engineering studies with today's modeling tools. Thus, extra time and effort are required to coordinate the design formulation and the design analysis. This limits the ability of developers to explore design alternatives. The matching of Use Case scenarios and performance scenarios, together with the use of a tool, such as $SPE{\bullet}ED$, that automates key aspects of the SPE process represent a significant step toward achieving this goal.

As noted in Section 3, the translation of scenarios to execution graphs is currently a manual process. However, the close correspondence between scenarios as expressed in sequence diagrams (with MSC extensions) and execution graphs suggests that an automated translation may be possible. A future project will explore this possibility. A previous project developed an SPE meta-model that defines the information requirements for SPE [35]. The SPE meta-model can be used by CASE tool vendors to add the capability to collect performance data as part of the design information. By collecting performance data within the design tool and automatically translating the design models to execution graphs, it will be possible to export data from the CASE tool to any SPE tool that supports the meta-model. Thus CASE tools need not replicate the performance expertise already available. This offers a more cost-effective approach to supporting SPE. Our other future work will address performance patterns and antipatterns, the specification of performance requirements, and additional tool features to automate SPE evaluations.

REFERENCES

[1] *L&S Computer Technology Inc.*, Performance Engineering Services Division, Austin, TX 78766, (505) 988-3811, www.perfeng.com

[2] K. Auer and K. Beck, *Lazy Optimization: Patterns for Efficient Smalltalk Programming* in Pattern Languages of Program Design, vol. 2, J. Vlissides, J. Coplien and N. Kerth, ed., Reading, MA, Addison-Wesley, 1996.

[3] S. Balsamo, P. Inverardi, and C. Mangano, *An Approach to Performance Evaluation of Software Architectures* Workshop on Software and Performance, Santa Fe, NM, ACM, 1998, pp. 178-190.

[4] H. Beilner, J. Mter, and C. Wysocki, *The Hierarchical Evaluation Tool HIT* in Performance Tools & Model Interchange Formats, vol. 581/1995, Falko Bause and Heinz Beilner, ed., D-44221 Dortmund, Germany, Universitt Dortmund, Fachbereich Informatik, 1995, pp. 6-9.

[5] Heinz Beilner, J. Mter, and N. Weissenburg, *Towards a Performance Modeling Environment: News on HIT* Proceedings 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Plenum Publishing, 1988.

[6] G. Booch, J. Rumbaugh, and I. Jacobson *The Unified Modeling Language User Guide*, Reading, MA, Addison-Wesley, 1999.

[7] W.J. Brown, et al. *AntiPatterns: Refactoring Software Architectures, and Projects in Crisis*, New York, John Wiley and Sons, Inc., 1998.

[8] P.C. Clements *Coming Attractions in Software Architecture*, No.CMU/SEI-96-TR-003, Software Engineering Institute, Carnegie Mellon University,February 1996.
[9] R.G. Franks, et al. *A Toolset for Performance Engineering and Software Design of Client-Server Systems*, Performance Evaluation, vol. 24, no. 1-2, 1995, pp. 117-135.
[10] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA, Addison-Wesley, 1995.
[11] Robert T. Goettge *An Expert System for Performance Engineering of Time-Critical Software* Proceedings Computer Measurement Group Conference, Orlando FL, 1990, pp. 313-320.
[12] Adam Grummitt *A Performance Engineer's View of Systems Development and Trials*, Proceedings Computer Measurement Group Conference, Nashville, TN, 1991, pp. 455-463.
[13] Neil Gunther *E-Ticket Capacity Planning: Riding the E-Commerce Growth Curve*, Proc. Computer Measurement Group, Orlando, 2000.
[14] Greg Hills, Jerome A. Rolia, and Giuseppe Serazzi *Performance Engineering of Distributed Software Process Architectures*, Modelling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, vol. 977, Springer, 1995, pp. 357-371.
[15] ITU *Criteria for the Use and Applicability of Formal Description Techniques* Message Sequence Chart (MSC),", International Telecommnication Union 1996.
[16] R. Kazman, et al. *Scenario-Based Analysis of Software Architecture*, IEEE Software, vol. 13, no. 6, 1996, pp. 47-55.
[17] R. Kazman, et al. *The Architecture Tradeoff Analysis Method*, Software Engineering Institute, Carnegie Mellon University 1997.
[18] Peter King and Rob Pooley *Derivation of Petri Net Performance Models from UML Specifications of Communications Software*, Modelling Tools and Techniques, B. Haverkort, H. Bohnenkamp and C.Smith, ed., Schaumburg, IL, vol. 1786, Springer, 2000.
[19] Chung-Horng Lung, Anant Jalnapurkar, and Aham El-Rayess *Performance-Oriented Software Architecture Analysis: an Experience Report*, Workshop on Software and Performance, Santa Fe, NM, ACM, 1998, pp. 191-196.
[20] Daniel A. Menasc and Virgilio A.F. Almeida *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*, Prentice Hall, 2000.
[21] Daniel A. Menasc and Hassan Gomaa *On a Language Based Method for Software Performance Engineering of Client/Server Systems*, Workshop on Software and Performance, Santa Fe, NM, ACM, 1998, pp. 63-69.
[22] J.A. Rolia *Predicting the Performance of Software Systems*, University of Toronto, 1992.
[23] J.A. Rolia and K.C. Sevcik *The Method of Layers*, IEEE Trans. on Software Engineering, vol. 21, no. 8, 1995, pp. 689-700.
[24] Jerome A. Rolia *Performance Estimate for Systems with Software Servers: The Lazy Boss Method*, Proceedings VIII SCCC International Conference on Computer Science, Santiago, Chile, 1988.
[25] H. Schwetman *CSIM17: A Simulation Model-Building Toolkit*, Proceedings Winter Simulation Conference, Orlando, 1994.
[26] C. Scratchley and C.Murray Woodside *Evaluating Concurrency Options in Software Specifications*, Proc. 7th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, MD, 1999, pp. 330-338.
[27] Connie U. Smith *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.
[28] Connie U. Smith and Lloyd G. Williams *Software Performance Engineering: A Case Study with Design Comparisons*, IEEE Transactions on Software Engineering, vol. 19, no. 7, 1993, pp. 720-741.
[29] Connie U. Smith and Lloyd G. Williams *Performance Engineering of Object-Oriented Systems with SPEED*, in Lecture Notes in Computer Science 1245: Computer Performance Evaluation, Marie R. et. al., ed., Berlin, Germany, Springer, 1997, pp. 135-154.
[30] Connie U. Smith and Lloyd G. Williams *Performance Engineering Evaluation of CORBA-based Distributed Systems with SPEED*, in Lecture Notes in Computer Science, R. Puigjaner, ed., Berlin, Germany, Springer, 1998.
[31] Connie U. Smith and Lloyd G. Williams *Responsive, Scalable Systems: Practical Performance Engineering for Object-Oriented Software*, 2001.
[32] Diana Szumilas, Sandra McCrady, and Ron Leighton *Performance Engineering a Distributed Architecture*, Proc. Computer Measurement Group, San Diego, CA, 1996, pp. 703-712.
[33] A. Thomasian and P. Bay *Performance Analysis of Task Systems Using a Queueing Network Model*, Proceedings International Conference Timed Petri Nets, Torino, Italy, 1985, pp. 234-242.
[34] Michael Turner, Douglas Neuse, and Richard Goldgar *Simulating Optimizes Move to Client/Server Applications*, Proceedings Computer Measurement Group Conference, Reno, NV, 1992, pp. 805-814.
[35] 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, Springer, 1995.
[36] C.M. Woodside, et al. *The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software*, IEEE Trans. Computers, vol. 44, no. 1, 1995, pp. 20-34.

Edited by: Janusz S. Kowalik
Accepted: September 15th, 2000