Performance Engineering Models of CORBA-based Distributed-Object Systems

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 Engineering Models of CORBA-based Distributed-Object Systems

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

Systems using distributed object technology offer many advantages and their use is becoming widespread. Distributed object systems are typically developed without regard to the locations of objects in the network or the nature of the communication between them. However, this approach often leads to performance problems due to latency in accessing remote objects and excessive overhead for communication. Thus, it is important to provide support for early assessment of the performance characteristics of such systems. This paper presents extensions to the software performance engineering process and its associated models for assessing distributed object systems, and illustrates with a case study.

1. Introduction

Distributed-object technology (DOT) is the result of merging object-oriented techniques with distributed systems technology. This approach makes objects the unit of computation and distribution in a distributed environment by exploiting two key features of objects: their ability to encapsulate both data and operations in a single computation unit, and their ability to separate their interface from their implementation. DOT is enabled and supported by "middleware" such as the OMG Object Management Architecture which provides referential transparency and high-level inter-object communication mechanisms.

Systems based on DOT offer a number of advantages to software development organizations. Developers can design and implement a distributed application without being concerned about where a remote object resides, how it is implemented, or how inter-object communication occurs. Applications can be distributed over an heterogeneous network, making it possible to run each component on the most appropriate platform. In addition, "wrappers" can be used to make commercial off-the-shelf (COTS) products and/or legacy systems appear as objects. This makes it possible to integrate COTS or legacy software into more modern systems.

Distributed object systems are typically developed without regard to the locations of objects in the network or the nature of the communication between them. However, this approach often leads to performance problems due to latency in accessing remote objects and excessive overhead due to inefficient communication mechanisms [WALD94].[1] For example, referential

[1] Waldo, et. al. also note that ignoring the difference between local and remote objects can lead to other problems due to memory access, concurrency and partial failure.

transparency may require that a local object be accessed using remote procedure calls rather than a more efficient mechanism, such as shared memory. The performance of these systems must then be "tuned" by fixing the locations of critical objects (e.g., making remote objects local) and replacing slow communication mechanisms with more efficient ones (e.g., replacing remote procedure calls by shared memory communication for local objects).

This "fix-it-later" approach introduces a number of problems. Systems delivered with poor performance 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 design, negating the benefits obtained from using the object-oriented approach. 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.

Most performance failures are due to a lack of consideration of performance issues early in the development process. Poor performance is more often the result of problems in the design rather than the implementation. As Clements and Northrup note:

> "Performance depends largely upon the volume and complexity of inter-component communication and coordination, especially if the components are physically distributed processes." [CLEM96]

Our experience is that it is possible to *cost-effectively* engineer distributed-object systems that meet performance goals. By carefully applying the systematic techniques of software performance engineering (SPE) throughout the development process, it is possible to produce architectures and designs that have adequate performance *and* exhibit the other qualities, such as reusability, maintainability, and modifiability that have made distributed systems based on object-oriented technology so effective [SMIT93a], [SMIT97].

However, systems based on the OMG Object Management Architecture are relatively new and offer new challenges for performance modeling. Our previous papers extended Software Performance Engineering (SPE) methods to include specific techniques for evaluating the performance of object-oriented systems. They focused on early life-cycle issues and introduced Use Cases as the bridge between object-oriented methods and SPE [SMIT97],[SMIT98]. However, they did not specifically address distributed systems or issues related to the use of CORBA. This paper extends our previous work to include extensions to the SPE methods that address:

- performance engineering methods appropriate for distributed systems,
- extensions to the SPE modeling techniques to evaluate performance issues that arise when using an object request broker (ORB), and
- modeling techniques for CORBA synchronization primitives

In the following sections we: explain the SPE process extensions for distributed systems; cover the performance issues introduced when distributed systems use the OMG Object Management Architecture for inter-object communication and coordination; present the performance models for distributed systems; and illustrate the models with a case study.

2. Related Work

Gokhale and Schmidt describe a measurement-based, principle-driven methodology for improving the performance of an implementation of the Internet Inter-ORB Protocol (IIOP) [GOKH97]. Their paper presents a set of principles (first formulated by Varghese [VARG96]) and illustrates their use in improving the performance of the IIOP. Their work is aimed at improving elements of the CORBA facilities. Ours focuses on the architecture of an application that uses CORBA facilities and the effect of the inter-process communication on its performance.

Meszaros [MESZ96] presents a set of patterns for improving the performance (capacity) of reactive systems. Their work is concerned with identifying a performance problem together with a set of forces that impact possible solutions. The patterns then suggest solutions that balance these forces. Petriu and Somadder [PETR97] extend these patterns to distributed, multi-level client/server systems. Our work focuses on early evaluation of software designs via modeling. Meszaros and Petriu and Sommader propose ways of identifying solutions to performance problems but do not specify whether the problems are identified by measurement or through modeling. Since early modeling could be used to identify performance problems, their work complements ours by providing guidelines for selecting solutions.

Smith and Williams describe performance engineering of an object-oriented design for a real-time system [SMIT93a]. However, that approach applies general SPE techniques and only addresses the specific problems of object-oriented systems in an ad hoc way. It models only one type of synchronization, whereas this paper models three types. Smith and Williams also describe the application of Use Case scenarios as the bridge between design models and performance models in [SMIT97] and [SMIT98]. In contrast, this paper extends the SPE process to evaluate special performance issues in distributed systems using the earlier approach as a starting point.

Smith presents several advanced system execution model approaches for parallel and distributed processing, including remote data access, messages for inter-process communication, and Ada rendezvous [SMIT90a]. Those approaches were very general and thus complex. They focused on the features in the advanced system execution model with only a loose connection to the software execution model. It is viable to evaluate systems with those approaches, but it is better for very early life cycles stages to have a simpler approximation technique based on the software execution models to support architecture and design trade-off studies. This paper presents an approximation technique for software execution models, a simpler approach for representing the synchronization points in the execution graph and an automatic translation to the advanced system execution model.

Other authors have presented general approximation techniques for software synchronization (e.g., [THOM85]). They also adapt the system execution model to quantify the effects of passive resource contention. Rolia introduced the method of layers to address systems of cooperating processes in a distributed environment, and a modification to the system execution model solution algorithms to quantify the delays for use of software servers, and contention effects

introduced by them [ROLI95]. Woodside and co-workers propose stochastic rendezvous nets to evaluate the performance of Ada Systems [WOOD95]. All these approaches focus on synchronous communication, however adaptations to the various approximation techniques could be used for an approximate analytical solution to the advanced system execution model described in section 5.2.

3. Distributed System Extensions to the SPE Process

The SPE *process* for evaluating distributed-object systems is similar to that for other systems. However, the models require some extension to evaluate details of concurrency and synchronization. We use the software performance engineering tool, *SPE•ED,* to evaluate the performance models. Other tools are available, such as [BEIL95],[GOET90],[ROLI92], [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.

Software Performance Engineering (SPE) is a systematic, quantitative approach to constructing software systems that meet performance objectives. In early development stages, 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 development proceeds, the models are refined to more closely represent the performance of the emerging software. Because it is difficult to precisely estimate resource requirements early in development, SPE uses adaptive strategies, such as upper- and lower-bounds, and best- and worst-case analysis to manage uncertainty.

Two types of models provide information for design assessment: the *software execution model* and the *system execution model*. The software execution model represents key aspect of the software execution behavior; we use execution graphs to represent performance scenarios. The software execution model solution provides a static analysis of the mean, best-, and worst-case response times for an initial evaluation against performance objectives. The system execution model uses the results of the software execution model solution to study the effect of contention delays for shared computer resources.

SPE for distributed-object systems begins with the same steps used for all object-oriented systems. It begins with the Use Cases identified by developers during the requirements analysis and system design phases of the development cycle. Once the major Use Cases and their scenarios have been identified, those that are important from a performance perspective are selected for performance modeling. These scenarios, represented using Message Sequence Charts (MSCs) [ITU96], are translated to execution graphs which serve as input to *SPE•ED*. This process, originally presented in [SMIT97], is summarized below:

1. *Establish performance objectives* - the quantitative criteria for evaluating the performance characteristics of the system under development.

2. *Identify important Use Cases* - those that are critical to the operation of the system or which are important to responsiveness as seen by the user.

3. *Select key performance scenarios* - those which are executed frequently or those which are critical to the perceived performance of the system.

4. *Translate scenarios to execution graphs* - the MSC Use Case representation is translated to a software execution graph model.

5. *Add resource requirements and processing overhead* - estimates of the amount of processing required for each step in the execution graph, and the amount of service the software resources require from key devices in the hardware configuration.

6. *Solve the models* - solving the execution graph characterizes the resource requirements of the proposed software alone.

If the software model solution indicates problems, analysts consider architecture or design alternatives to address the problems. If not, then analysts proceed to evaluate additional characteristics of distributed systems.

SPE for distributed-object systems adds the following model features:

- Software execution model approximate techniques for estimating the performance effect of distributed objects
- An advanced system execution model to study the effect of contention for shared objects, and other delays for inter-process coordination.

The following section describes distributed object management, then section 5 describes these additional model features.

4. Object Management Performance Issues

Distributed-object technology is enabled by middleware[2] that allows objects in a network to interact without regard to hardware platform, implementation language, or communication protocol. The Object Management Group's (OMG) Object Management Architecture (OMA) is a widely-used specification for a set of middleware standards that allow development of applications in a distributed, heterogeneous environment. The OMA consists of five principal components [VINO97] (Figure 1): object services, common facilities, domain interfaces, application interfaces, and the object request broker.

[2] Middleware provides a layer of services between the application and the underlying platform (operating system and network software) [BERN96]. It provides application-independent services that allow different processes running on one or more platforms to interact.

The core of this architecture is the Object Request Broker (ORB). The ORB is responsible for managing communication between objects without regard to: object location, implementation and state, or inter-object communication mechanisms. The OMA Common Object Request Broker Architecture (CORBA) specifies a standard architecture for ORBs. The primary performance issues that we consider are due to this component. Aspects of the ORB that impact performance can be divided into two categories: those that do not need to be explicitly modeled and those that do.

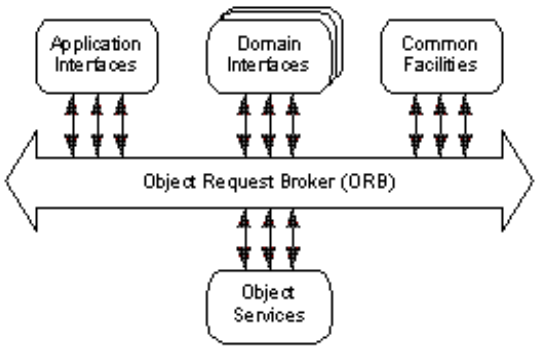


Figure 1. The OMA Reference Architecture

Aspects of the ORB that are vital to object interaction but do not require explicit modeling include:

- *the interface definition language*: declares the interfaces and types of operations and parameters (e.g., float, string, struct, etc.)
- *language mappings*: specifies how the interface definition language features are mapped to various programming languages
- *client stubs and server skeletons*[3]: provide mechanisms for interacting with the ORB to convert (static) request invocations from the programming language into a form for transmission to the server object and to similarly handle the response
- *dynamic invocation and dispatch*: a generic mechanism for dynamic request invocations without compile-time knowledge of object interfaces
- *protocols for inter-ORB communication*: the mechanism for ORB-to-ORB handling of request invocations

While these features affect the overall performance of a distributed system which includes an ORB, they are modeled implicitly by measuring their resource requirements and including it as "processing overhead" for each invocation of a server object.

The *Object Adapter* is the component of the ORB that actually connects objects to other objects. We also model several aspects of the object adapter by measuring their effect and including it in the processing overhead. These aspects are:

- objects and interfaces registration
- object reference generation
- server process activation
- object activation
- static and dynamic invocations
- communication overhead for transmitting request invocations
- processing overhead for converting requests across languages, operating systems, and hardware

[3] In the OMA, communication is peer-to-peer, however, the terms *client* and *server* describe the roles in a particular communication when one object requests services from another.

We measure the overhead of each of these aspects of the Object Adapter. Performance scenarios derived from Use Cases then provide the processing details that quantify the number of times each is used. For example, a Use Case may specify typical operational conditions that excludes system initialization (thus has no object registration, activation, etc.), and has a particular static object invocation in a loop that is executed 250 times. The corresponding performance scenario would have 0 registrations, 0 reference generations, 0 activations, and 250 static invocations (with processing overhead for invocation transmission and conversions derived from measurements). The software model solution quantifies the total processing requirement due to object adapter overhead.

Five aspects of the ORB that must be explicitly modeled are:

1. object location
2. process composition
3. request scheduling
4. request dispatching
5. coordination mechanisms

Object location and process composition determine the processing steps assigned to each performance scenario [WILL98]. Request scheduling and request dispatching are partially determined by contention for called processes which is determined by the coordination mechanism and other processing requirements of performance scenarios. The models in section 5 quantify these aspects of the ORB. Coordination mechanisms require extensions to the Use Case and performance modeling formalisms. These are described in the following section.

5. Distributed System Models

Early in development, 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 as in steps 1 through 6 in section 3 without explicitly representing synchronization -- it is represented only as a delay to receive results from a server process. A companion paper describes these approximate models and the delay-estimation approach [SMIT98c]. Later in the development process, more realistic models, described in section 5.2, translate the software execution models into a system execution model. They are solved with advanced system execution model techniques.[4]

In the following discussion we focus on synchronization and communication in distributed-object systems that use a CORBA compliant ORB. 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. The case study in Section 6

[4] These discussions assume some familiarity with the modeling notations used here, including execution graphs and Message Sequence Charts (MSCs). More information about these notations may be found in [SMIT90a] and [SMIT97].

illustrates more complex synchronization patterns in a distributed system; each individual communication is one of the three in the following section.

For this paper, we consider the three types of coordination mechanisms between objects currently supported by the CORBA architecture [VINO97].

- *Synchronous invocation*: The sender invokes the request and blocks waiting for the response (an invoke call).
- *Deferred synchronous invocation*: The sender invokes the request and continues processing. Responses are retrieved when the sender is ready (a send call, followed by a get_response call to obtain the result).
- *Asynchronous invocation*: The sender invokes the request and continues processing; there is no response a (send call to a one_way operation or with INV_NO_RESPONSE specified).

Synchronous or asynchronous invocations may be either static or dynamic; currently, deferred synchronous invocations may only be dynamic. SPE Model extensions to handle these performance issues are discussed in the next section.

5.1 Approximate Models of Software Synchronization

Our companion paper describes techniques for approximating the performance of the three types of synchronization [SMIT98c]. The process is summarized here, details may be found in the cited paper. The approximate models are illustrated in the case study in section 6.

Analysts first create a separate performance scenario for each process in the Use Case, specify resource requirements for processing steps and estimate the delay for communication and synchronization with other processes. The models may be iterative - the solution of each independent performance scenario quantifies its processing time. The processing time for the called processes can be used to refine the estimate of the delay for communication and synchronization.

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.

5.2 Detailed Models of Synchronization

Detailed models of synchronization will connect client requests across processing nodes more realistically reflecting the complex processing behavior. We start with the separate performance scenarios created in the approximate analysis step. We insert special notations into the MSCs at points when CORBA-based coordination is required. These lead to the insertion of special nodes into the execution graphs to represent synchronization steps. An advanced system execution model is automatically created from the execution graphs and solved to quantify contention effects and delays. These steps are described in the following sections.

5.2.1 MSC Extensions to represent CORBA Coordination

As noted in Section 3, the SPE process for distributed-object systems begins with a set of Use Case scenarios expressed in the Message Sequence Chart (MSC) notation. In order to construct

performance models of these scenarios, it is necessary to specify the synchronization points and type of synchronization in these scenarios. Since the MSC notation [ITU96] does not provide graphical syntax for representing the CORBA synchronization primitives, we use elements of the UML notation [RATI97] for synchronous and asynchronous calls together with an extension of our own to show the deferred synchronous invocation. These are shown in Figure 2.

The notation for synchronous invocation shows a break in the processing when the caller waits for the response. The deferred synchronous notation represents continued processing with a potential delay before receiving the reply. The arrow representing the reply will be shown at the point when it occurs, processing may continue in the called process. The notation for asynchronous invocation has a partial arrow for the call and no return arrow.

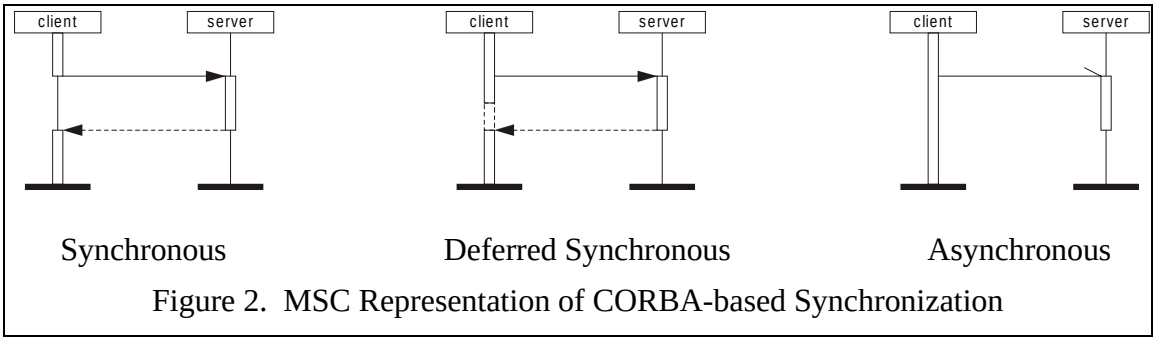


Figure 2. MSC Representation of CORBA-based Synchronization

5.2.2 Software Execution Model Extensions

The execution graph notation also requires extension to represent the CORBA coordination mechanisms. Figure 3 shows the new 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

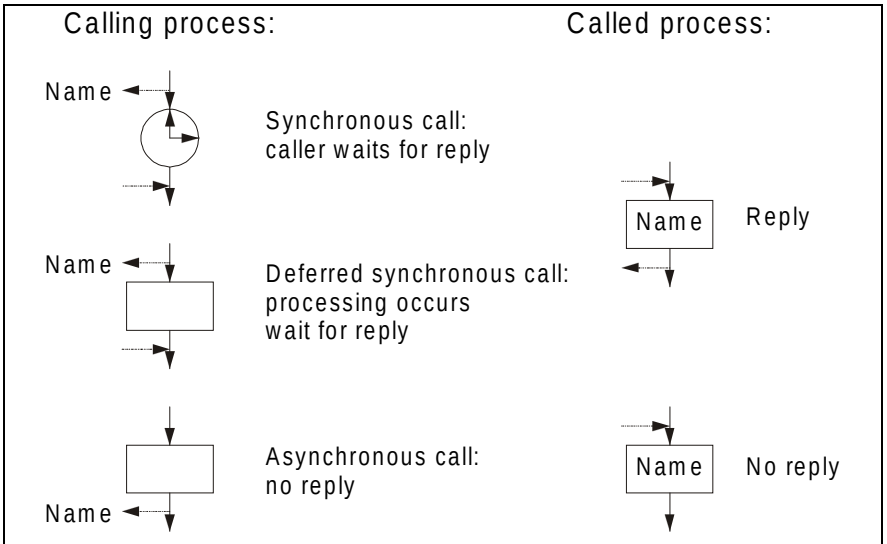


Figure 3. Execution Graph Nodes for Software Synchronization

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 steps and the number of threads for called processes. Each scenario is assigned to a processing node. The software execution model solution provides the model parameters for the advanced system execution model.

SPE•ED uses the CSIM simulation tool to solve the advanced system model. CSIM is a simulation product that is widely used to evaluate distributed and parallel processing systems [SCHW94]. The SPE tool solves the software execution model to derive the computer device processing requirements, then creates the advanced system execution model and solves it with a table-driven CSIM simulation model [5]. The solution is actually a hybrid solution. The software execution model solution provides summary data for processing phases [SMIT90a]. This provides an efficient simulation solution at a process-phase granularity rather that a detailed process simulation.

Two CSIM synchronization mechanisms provide the process coordination required for the software synchronization: *events* and *mailboxes*. An event consists of a state variable and two queues (one for *waiting* processes and the other for *queued* processes). When the event "happens" one of the queued processes can proceed. (The waiting queue is not used in this synchronization model). An event happens when a process *sets* the event to the occurred state. A mailbox is a container for holding CSIM messages. A process can *send* a message to a mailbox or *receive* a message from a mailbox. If a process does a receive on an empty mailbox, it automatically waits until a message is sent to that mailbox. When a message arrives, the first waiting process can proceed.

A synchronous or deferred synchronous call is implemented in *SPE•EDs* advanced system execution model with a *send* to the *name* mailbox of the called process (the *name* appears in the processing node of the called process). The message is an *event* that is to be set for the "reply." An asynchronous call is also implemented with a mailbox *send*; the message is ignored because there is no reply. The calling process executes a *queue* statement for the *event*; synchronous invocations place with the queue statement immediately after the send, deferred synchronous invocations place it at the designated point later in the processing. The called process issues a *receive* from its mailbox. At the reply's designated point in the processing, the called process *sets* the event it received. If there are multiple threads of the called process, the next in the receive queue processes the request. The event must be unique for the correct process to resume after the *set* statement executes.

In addition to the standard results reported by *SPE•ED*, the following are reported for the synchronization steps:

[5] An approximate solution to the advanced system model is possible, but not currently in *SPE•EDs* repertoire. Several of the approximations mentioned in the related work section might be applicable if they could be automated and not require modeler's intervention.

- mean, minimum, maximum, and variance response time for called processes
- mean and maximum number of requests and the mean time in queue (mailbox) 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. It shows the proportion of the computer resource requirements used by each scenario, and the overall device utilization.

Next, the case study illustrates both the approximate models and the advanced system execution model features.

6. 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.[6] 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.

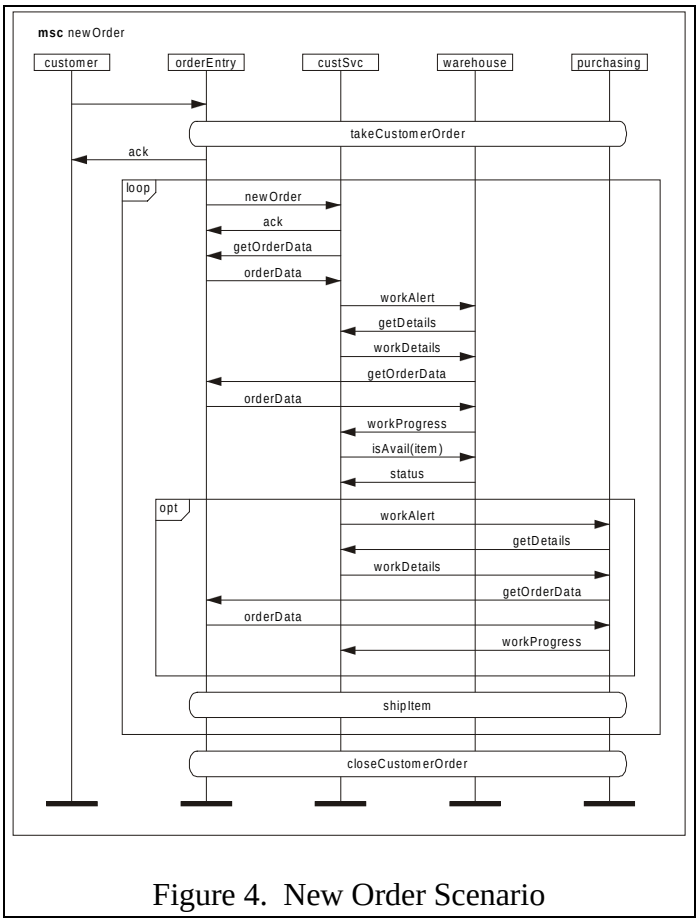


Figure 4. New Order Scenario

6.1 Approximate Model

The Use Case we consider is processing a new order (Figure 4). It begins with TakeCustOrder, an MSC reference to another, more detailed MSC. 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 MSC shows this repetition with a loop

[6] eStuff is a fictional web site. At the time this paper was written no such site existed.

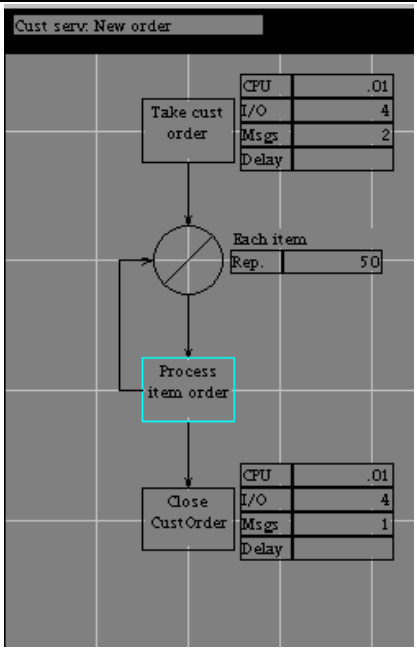


Figure 5. Execution Graph
CustomerService: New Order

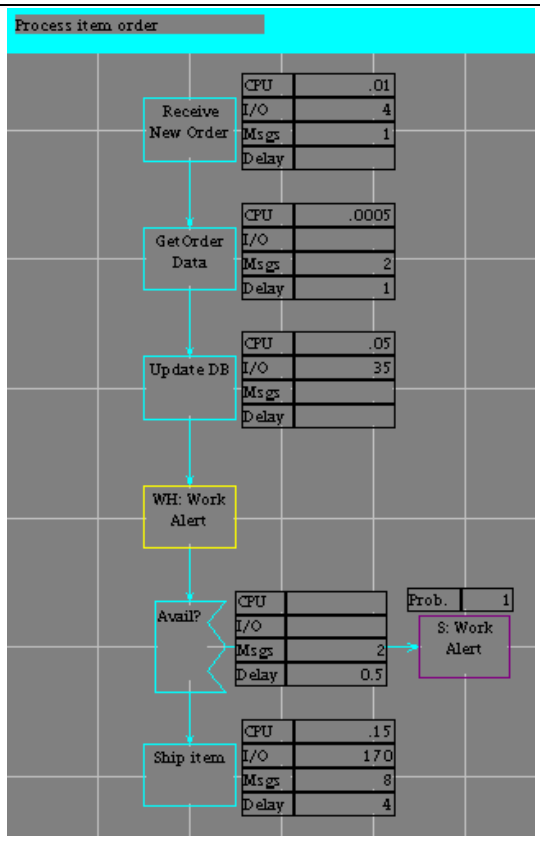


Figure 6. Execution Graph ProcessItemOrder

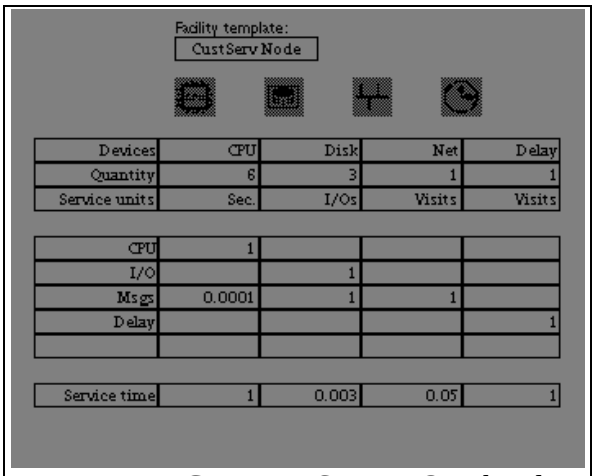
symbol. The similar symbol labeled "opt" represents an optional step that may occur when eStuff must order the item from a supplier.

Figure 5 shows the execution graph for the CustomerService column in the Use Case in Figure 4. Everything inside the loop is in the expanded node, ProcessItemOrder. Its details are in Figure 6. The two WorkAlert processing steps are also expanded; their details are not shown here. At this stage we assume that each of the first three columns in Figure 4 executes on its own facility, warehouse and purchasing share a processor. This software model depicts only the CustomerService processing node; we approximate the delay time to communicate with the other processing nodes.

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

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 database and other processing, the number of I/Os for database and logging activities, the number of messages sent among processors (and the associated overhead for the ORB), and the estimated delay in seconds for the network communication and execution time of called processes until the message-reply is received. These values are shown in Figures 5 and 6. They are best-case estimates derived from performance measurement experiments.

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. Figure 7 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 values in the middle part of the matrix specify the amount of computer device processing required for each software resource request. For example, the values for the Msgs row specifies that each message specified in the software model results in 0.1 ms. CPU time for ORB processing[7], 1 Disk I/O, and 1 message via the Net. The values in the last row specify the service time in seconds for each device in the facility. When the device quantity is greater than one, *SPE•ED* uses the optimistic assumption that the requests to devices are equally spread.



Devices	CPU	Disk	Net	Delay
Quantity	6	3	1	1
Service units	Sec.	I/Os	Visits	Visits
CPU	1			
I/O		1		
Msgs	0.0001	1	1	
Delay				1
Service time	1	0.003	0.05	1

Figure 7. Customer Service Overhead Matrix

Figure 8 shows the best-case solution with one user and thus no contention for computer devices. The end-to-end time is 480 seconds, most of that is in the ProcessItemOrder step (the value shown is for all 50 items). Results for the ProcessItemOrder subgraph (not shown) indicate that each item requires approximately 9.8 seconds, most of that is in the ShipItem processing step. Other results (not shown) indicate that 7.5 seconds of the 9.8 seconds is due to estimated delay for processing on the other facilities. Investigation also shows that the system cannot support the desired throughput primarily due to network congestion. Analysts can evaluate potential solutions to both the excessive delay for remote processing and the network congestion by modifying the software execution model to reflect architecture alternatives. An alternative is selected that processes work orders as a group rather than individual items in an order. The changes to the software execution model for

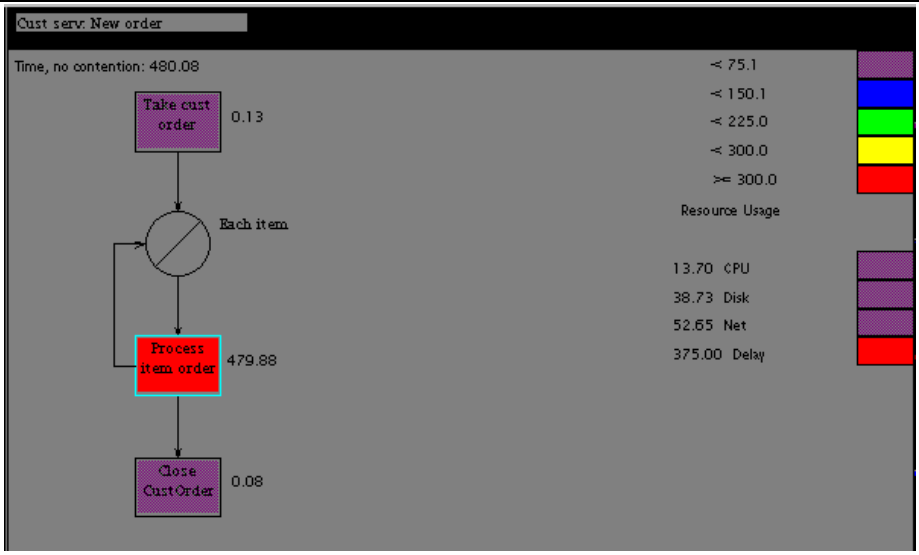


Figure 8. Best-case elapsed time.

[7] The CPU time for ORB processing is from performance benchmarks on a particular platform, and is unlikely to generalize to other environments. See [GOKH97] and [ORFA97] for other measurement values.

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 16.5 seconds with the desired throughput of 0.1 jobs per second.

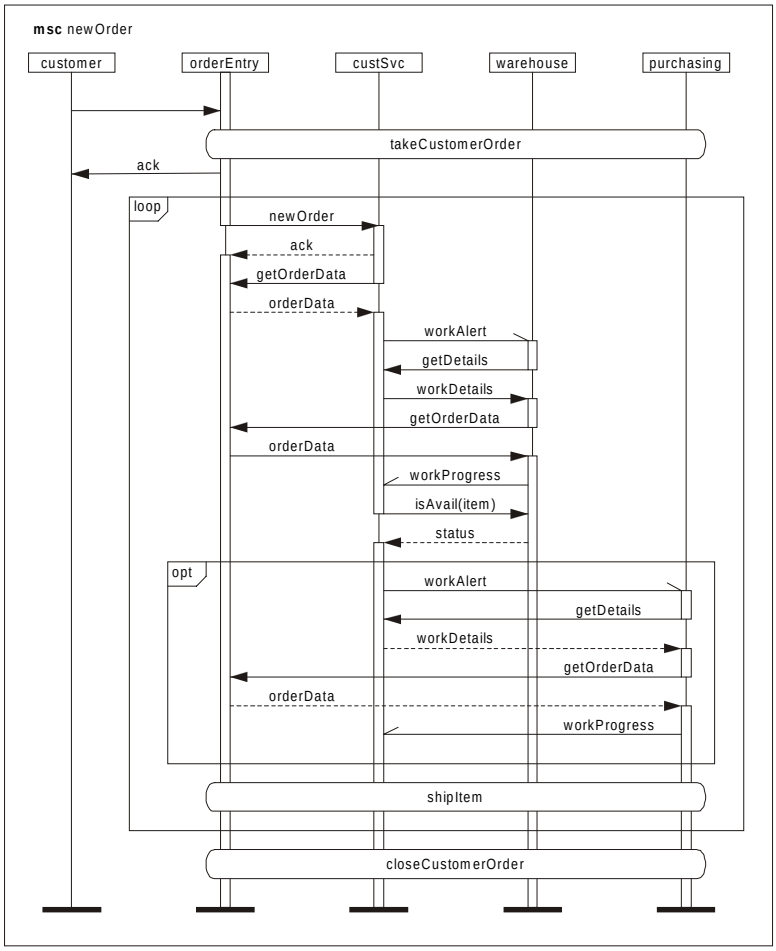


Figure 9. New order scenario with synchronization

Thus, the overhead and delays due to CORBA-based 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 to examine alternatives with simple models. It is also important to resolve key performance problems before proceeding to the more advanced models described in the next section.

6.2 Advanced System Execution Model

This section illustrates the creation and evaluation of the detailed models of synchronization. Figure 9 shows the scenario diagram modified to show the synchronization. Note that all the synchronization steps are either asynchronous or synchronous. This situation is probably typical in initial designs. Deferred synchronous calls are only useful when the results of the call are not needed for the next processing steps, and deferred synchronous communication may be more complex to implement. Thus, it is sensible to select synchronous calls at this stage. The models can be used to determine whether significant improvements are possible with deferred synchronous calls.

Figure 10 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 OE:GetOrderData and waits for the reply.

The availability check is now explicitly modeled with a synchronous call to WH:Avail? The processing steps here are similar to those in the approximate model.

Next we examine the approximate model for the WH:WorkAlert expanded node in Figure 11. It first invokes the WorkAlert on the Warehouse (WH) facility. The WH facility invokes the GetDetails process and the CS server sends the reply. Sometime later the WH facility makes an asynchronous call to report WorkProgress. The delay would correspond to the time required to ship the item - we do not include this delay in the model, but we do represent the resource requirements it will ultimately require from the facility. For convenience the approximate software model represents the total processing required for steps, but at that stage does not represent that the three steps actually occur in three separate processes. At the architecture stage in development, these details have not yet been determined.

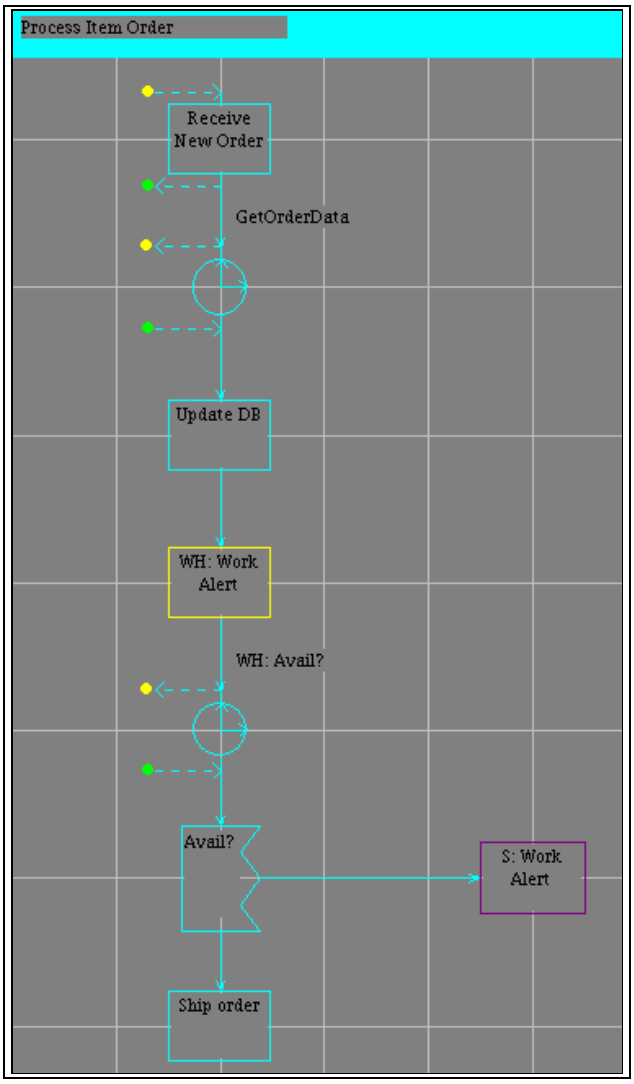


Figure 10. Synchronization in ProcessItemOrder.

In the advanced model, the WH:WorkAlert and S:WorkAlert steps each contain a single asynchronous call. The other two steps occur in separate processes. Figures 12 shows the processing for the status update step (called UpdateDB:Progress in the approximate software model).

Figure 13 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 call to OE:GetOData, then (after the order is shipped) makes an asynchronous call to CS StatusUpdate.

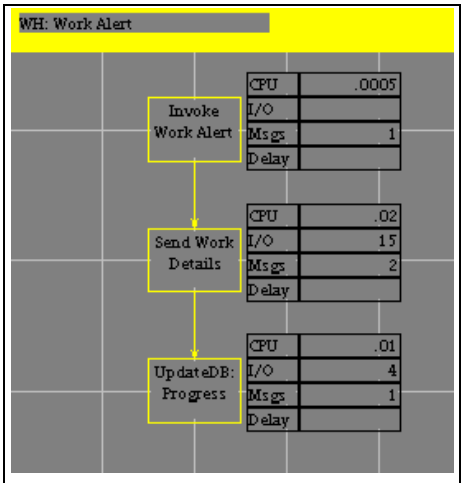


Figure 11. Approximate model for WH:WorkAlert.

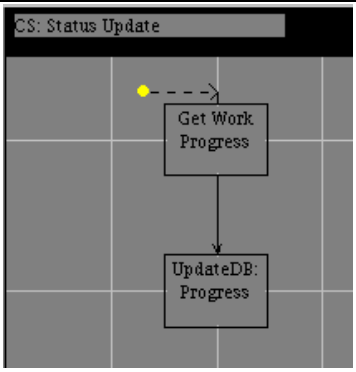


Figure 12. CS:StatusUpdate Process

Table 1 summarizes the results of the advanced system execution model. They reflect results for 10 hours simulated time with a total of 3565 CS:NewOrders during that interval. The confidence level for these results was computed using the batch mean method; the result is 0.172 for the CS:NewOrder scenario. 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.

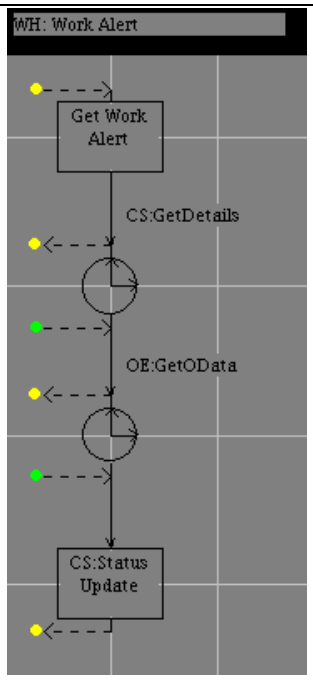


Figure 13. WH:WorkAlert Process

The maximum queue length and the queue time for WH:WorkAlert suggests that more concurrent threads might be desirable for scalability. The simulation results would 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.

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

Table 1. Advanced System Model Results

	Response Time (secs.)				TPut	Queue		
	Mean	Min	Max	Variance		Mean	Max	Time
CS:NewOrder	14.4	0.8	72.7	79.51	.1			
OE:OrderData	0.16	0	2.6	0.05	.5	0.092	5	0.19
CS:WorkDetails	0.2	0	3.7	0.05	.3	0.057	2	0.19
CS:UpdStatus	0.1	0	4.4	0.04	.3	0.004	3	0.01
WH:WorkAlert	1.3	0	9.1	1.14	.2	0.122	9	0.62
P:WorkAlert	1.4	0	9.3	1.16	.1	0.019	3	0.193

processors, an approximate number of threads per process, etc., can all be evaluated with the simpler models.

7. Summary and Conclusions

Studying the performance of software architectures for distributed-object systems, particularly those that have extensive communication and synchronization, is of growing importance. This paper presents and illustrates the SPE process and model extensions for distributed-object systems that use the OMG Object Management Architecture for inter-object communication and coordination. Two types of models are used: approximate software models for quick and easy performance assessments in early life cycle stages, and advanced system execution models for more realistic predictions and analysis of details of interconnection performance.

Approximate software models are constructed from Use Case scenarios identified during the analysis and design phases of the development process. once these models have been constructed, the advanced system execution models are created and solved automatically by the *SPE•ED* performance engineering tool. This simplifies the modelers task because it eliminates the need for debugging complex simulation models of synchronization. It also permits an efficient hybrid solution that summarizes processing requirements by phases instead of detailed process oriented simulations.

The five key aspects of the CORBA Object Adapter that determine the performance of systems and are explicitly modeled were listed in Section 4. Three of them -- process composition, process location, and synchronization type, are actually architecture and design decisions. Thus, it makes sense to execute quantitative evaluations with these performance modeling techniques to determine the appropriate choice.

Future work may explore the feasibility and usefulness of using approximate solution techniques for the advanced system execution model. This could be accomplished by implementing additional solution techniques, or by using the software meta-model to exchange model information with another tool for solution and evaluation [WILL95].

8. 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.

[BERN96] P.A. Bernstein, "Middleware: A Model for Distributed System Services," *Communications of the ACM*, 39(2), 86-98, 1996.

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

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

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

[GOKH97] A. Gokhale and D.C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance," *Proceedings of the Thirty-first Hawaii International Conference on System Sciences (HICSS)*, Kohala Coast, HI, January 1997,.

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

[MESZ96] G. Meszaros, *A Pattern Language for Improving the Capacity of Reactive Systems*, , Addison-Wesley, Reading, MA, 1996, 575-591.

[ORFA97] R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*, John Wiley and Sons, New York, NY, 1997.

[PETR97] D. Petriu and G. Somadder, "A Pattern Language for Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers," *Proceedings of EuroPLoP'97*, Kloster Irsee, Germany, July 1997,.

[ROLI92] J.A. Rolia, Predicting the Performance of Software Systems, Thesis, University of Toronto, , , 1992.

[ROLI95] J.A. Rolia and K.C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, 21(8), 689-700, 1995.

[SCHW94] H. Schwetman, "CSIM17: A Simulation Model-Building Toolkit," *Proceedings Winter Simulation Conference*, Orlando, 1994,.

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

[SMIT93a] Connie U. Smith and Lloyd G. Williams, "Software Performance Engineering: A Case Study with Design Comparisons," *IEEE Transactions on Software Engineering*, 19(7), 720-741, 1993.

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

[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 Models of Distributed System Architectures," *Proc. Computer Measurement Group*, Anaheim, submitted for publication.,.

[THOM85] A. Thomasian and P. Bay, "Performance Analysis of Task Systems Using a Queueing Network Model," *Proceedings International Conference Timed Petri Nets*, Torino, Italy, 1985,234-242.

[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.

[VARG96] G. Varghese, *Algorithmic Techniques for Efficient Protocol Implementation*, , Stanford, CA, 1996,

[VINO97] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications*, 35(2), 46-55, 1997.

[WALD94] J. Waldo, et al., A Note on Distributed Computing, 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] L.G. Williams and C.U. Smith, "Performance Evaluation of Software Architectures," *Proc. First International Workshop on Software and Performance*, Santa Fe, NM, October 1998,.

[WOOD95] C.M. Woodside, et al., "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software," *IEEE Trans. Computers*, 44(1), 20-34, 1995.