Performance Analysis of Distributed Server Systems

by

Roy Gregory Franks, B.A.Sc, M.Eng

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirements for the degree of

Doctor of Philosphy

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

December 20, 1999

The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

"Performance Analysis of Distributed Server Systems"

submitted by Roy Gregory Franks, B.A.Sc., M.Eng.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Department Chair

Thesis Supervisor

External Examiner

Abstract

Client-server systems are becoming increasingly common in the world today as users move to networks of distributed, interacting computers. This form of work demands new performance models as the interactions in client-server systems are more complex than the types supported by classic queueing network solvers such as Mean Value Analysis. A Layered Queueing Network is one of these models; it uses hierarchical decomposition and surrogate delays to solve the model.

This thesis describes a new analytic modeling tool called LQNS (Layered Queueing Network Solver) which extends previous techniques used to model distributed client-server systems. The contributions of the thesis are as follows. First, the model now supports *forwarding*. Forwarding is a technique where a reply to a client is deferred to a lower level server in a multi-level system, improving performance by reducing message traffic. Forwarding can also be used to convert open models to closed models. Second, systems that use *early replies* can be modeled. Early replies are used to reduce the response time by replying to a client before all of its work at a server is completed. Previous techniques have been extended to multiservers and to allow multiple clients. Third, *activities* have been introduced. Activities represent the smallest unit of modeling detail and can have arbitrary precedence relationships. Finally, the solver has been extended to handle models with both homogeneous and heterogeneous threads within a task. Homogeneous threads are used to model multiservers. Heterogeneous threads are used to model fork-join interactions such as asynchronous remote procedure calls and in in RAID storage devices. The solver also incorporates accuracy improvements for models with early replies and for models with multiple layers.

The solver has been used to analyze numerous systems found in existence today including a tele-operator system and a transaction processing system. Finally, a extensive performance model of the Linux 2.0 Network File System (NFS) is presented.

Acknowledgements

I would like to dedicate this thesis to the memory of my father, Roy Wilfred Franks.

I would like to thank my supervisor, Professor Murray Woodside, for his support and friendship throughout the years. His guidance been instrumental in success of this work.

I would like to thank my family and friends for their moral support. In particular, I would like to thank my wife, Sylvie Lépine, for her support and desire to help, my son, Erik, for his smile each and every morning which always boosted my spirits, and my mother, Eleanor, for her constant encouragement.

I would also like to thank all the people at Carleton University I have had the pleasure to meet. Alex Hubbard, Curtis Hrischuk, Dorina Petriu and Shikharesh Majumdar all deserve mention for their involvement with this work. Don Bailey, Darlene Herbert, Naren Mehta, Dave Sword and Danny Lemay also deserve mention, either for keeping me on the straight and narrow, or for keeping everything in the RADS lab running so smoothly.

Financial assistance was provided by Communications and Information Technology of Ontario (CITO), the National Science and Engineering Research Council of Canada (NSERC) and Carleton University.

Contents

List of Figures

List of Tables

Glossary

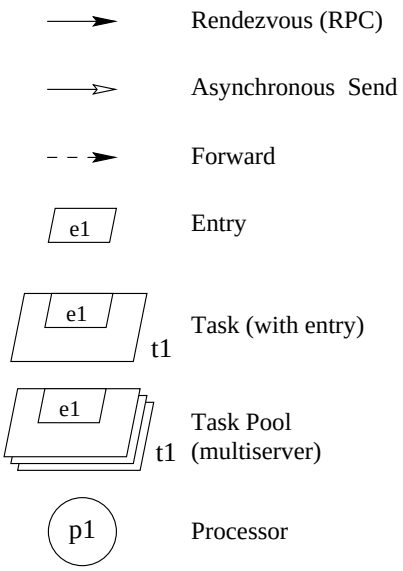


Figure 1: SRVN Nomenclature

m	station index.
M	the number of stations.
k	chain index.
K	the total number of chains in the queueing network model.
$\mathbf{e}_k$	the vector $[0_1, 0_2, ..., 1_k, ...0_K]$.
J_m	the number of servers at station m.
e	entry index.
E_m	the number of entries at station m.
s_{mk}	the mean service demand per visit of a chain-k customer at station m.
s_{mk1}	the mean phase-one service demand per visit of a chain-k customer at station m.
s_{mk2}	the mean phase-two service demand per visit of a chain-k customer at station m.
μ_{mk}	the mean service rate per visit of a chain-k customer at station m. $\mu_{mk} = S_{mk}^{-1}$
$\mathbf{N}$	the population vector of the network, with each element representing a chain.
$\alpha_m(i)$	the service rate multiplier for i customers of all chains at station m.
$\alpha_m(\mathbf{n})$	the service rate multiplier for a population $\mathbf{n}$ at station m.
$P_m(i, \mathbf{N})$	the probability that there are i customers at queue m.
$P_m(\mathbf{n}, \mathbf{N})$	the probability that the vector of the number of customers is $\mathbf{n}$ at queue m.
$PB_m(\mathbf{N})$	the probability that all M_m servers are busy at queue m. $PB_m(\mathbf{N}) = 1 - \sum_{i=1}^{M_m} P_m(i, \mathbf{N})$
$L_{mk}(\mathbf{N})$	the mean number of chain-k customers at queue m including those in service.
$Q_{mk}(\mathbf{N})$	the mean number of chain-k customers at queue m, but not in service. $Q_{mk}(\mathbf{N}) = L_{mk}(\mathbf{N}) - U_{mk}(\mathbf{N})$
$W_{mk}(\mathbf{N})$	the mean waiting time for a single visit of a chain-k customer at queue m.
$U_{mk}(\mathbf{N})$	the mean utilization of chain-k customers at queue m.
$U_{mk1}(\mathbf{N})$	the mean phase-one utilization of chain-k customers at queue m.
$U_{mk2}(\mathbf{N})$	the mean phase-two utilization of chain-k customers at queue m.
$U_{mk}^{(1)}(\mathbf{N})$	the mean utilization of one chain-k customer at queue m.
v_{mk}	the mean number of visits of a chain-k customer at station m.
λ_k	the chain-k throughput.
$\mathcal{C}_l$	The set of clients in submodel l.
$\mathcal{E}_m$	The set of entries for task or processor m.
$\mathcal{L}$	The set of MVA queueing submodels.
$\mathcal{P}$	the set of processors.
$\mathcal{S}_l$	The set of servers in submodel l.
$\mathcal{T}$	the set of tasks.

Table 1: Nomenclature

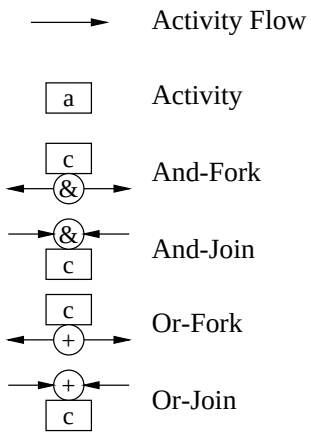


Figure 2: Activity Nomenclature

Chapter 1

Introduction

To gain a competitive edge, suppliers of computer systems either want to maximize performance for a given price-point, or minimize cost for a given level of functionality. Users of computer systems have the same goals. To achieve this end, performance analysis is necessary at all stages in the life cycle of a computer system. During the early design of a system performance analysis is used to aid in the comparison of design alternatives. When a system is being purchased or sold, performance analysis is used to determine its size. Finally, end-users can use performance analysis to determine whether the system is performing properly, and what effect changes to the configuration will have.

Business and industry is moving to the "client-server" computing paradigm to lower both the cost of the hardware and software needed today [130]. With this computing model, clients with varying degrees of sophistication are connected to one or more servers. The servers run applications on behalf of the clients, or store some resource such as data, or perform both functions. Servers may also call other servers, which forms the basis of the so-called *three-tiered* architecture.

The sections that follow describe briefly common techniques used for performance evaluation, the emerging client-server paradigm for distributed computing, and how designers

can improve the performance of client-server systems. This chapter concludes with contributions to the model used to solve performance models of client-server networks followed by an outline of the proposal.

1.1 Performance Evaluation

Three techniques are used typically for performance evaluation: measurement, simulation, and analytic modelling. Measurement involves the construction and test of a live system while simulation and analytic modelling rely on a model of the system being evaluated. The tradeoffs involved between the methods are discussed below.

The measurement technique is only possible if a system close to the desired configuration already exists. Since it requires a real system, it is often the most costly of the techniques described here. However, measurement of a live system can often give the most accurate results, provided that environmental parameters such as workload are representative. This factor alone often makes performance results derived from a measurement study the most believable when compared against results derived using the other two techniques.

Analytic modelling uses relatively simple mathematical expressions to derive the performance results for a system under study. These expressions can usually be solved quite quickly which aids in the exploration of the parameter space of a system. Unfortunately, many simplifying assumptions are often necessary in order to use analytical models. These assumptions may result in a model which may not represent accurately the system being studied, although experience modelling many systems has found that the prediction error for response time typically ranges between 10 and 30%. This error range is acceptable for a great number of applications [75].

A simulation relies on a model of a system being studied; as such, it can take place at any point in the life-cycle of the product. Once the model has been formulated, a program is generated that tracks the evolution of time in discrete steps that model events in the

actual system. The simulation may be generated automatically by a software tool such as RESQ [123], be written in a language for a special purpose simulation system such as GPSS [45], or it may be written in a generic programming language.

One major advantage of simulation over analytic modelling is that it can be used to create very detailed, thus potentially accurate models. Unfortunately, very detailed models are often time consuming and difficult to design, code, debug, parameterize and execute. The accuracy also depends on the run-time of the simulation because of its statistical nature. Finally, as the simulation becomes more costly to execute, the difficulty of evaluation of different system increases.

Deciding which technique to use is often based on the following criteria (ranked from most to least important) [59]:

Stage: point in life cycle when study is to take place.

Time required: when the results are needed.

Tools: analytic modelling tools, simulators, measurement packages.

Accuracy: degree to which results match reality.

Trade-off evaluation: ability to study different system configurations.

Cost: time and money needed to conduct the study.

Saleability: the degree to which others will believe the results.

Table 1.1 compares the techniques based on these criteria.

1.2 The Client-Server Model

The client-server computing paradigm is a system where processing of requests from users is distributed among several tasks. Tasks interact with one and another using the *remote*

Criterion	Analytical Modelling	Simulation	Measurement
1. Stage	Any	Any	Post prototype
2. Time Required	Small	Medium	Varies
3. Tools	Analysts	Computer languages	Instrumentation
4. Accuracy	Low	Moderate	Varies
5. Trade-off evaluation	Easy	Moderate	Difficult
6. Cost	Small	Medium	High
7. Saleability	Low	Medium	High

Table 1.1: Criteria for selecting an evaluation technique (from [59]).

procedure call (RPC) [143, 5]. Applications make requests for services using what appear to be conventional procedure calls. However, rather than branching to another section of the same program, a message is sent to another *task* which may or may not reside on the same computer. When the remote task replies, the remote procedure call returns. The client-server model also applies to a wide variety of other systems. Distributed systems built using the Common Object Request Broker (CORBA) [93], systems built using Ada [2] (the rendezvous), and those that run on the V [14] and Amoeba [88] operating systems, among others, also use the same send-receive-reply paradigm that makes up the remote procedure call.

1.2.1 Performance Analysis of Client-Server Systems

The blocking nature of the remote procedure call causes problems for mean value-based performance analysis. The remote procedure call is a type of simultaneous resource possession: the requesting task and the serving task are both held by the same customer while the remote procedure call is in progress. Furthermore, should the server continue to execute after the remote procedure call replies, a second customer is effectively created.

Fortunately, techniques have been developed which overcome the restrictions imposed by mean value analysis while preserving the solution speed. These approximations generally

rely on the techniques of surrogate delays and hierarchical decomposition to solve a multi-tier client-server model as a set of smaller product-form queuing network models. Two of the more successful approaches are Stochastic Rendezvous Networks [152] and the Method of Layers [110].

1.3 Performance of Client Server Systems

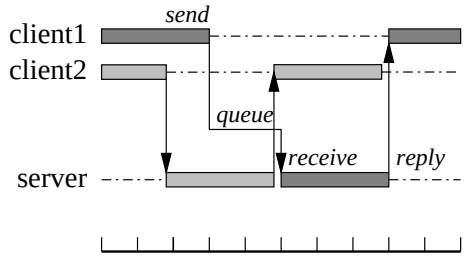
In a multi-tier client-server system, overall system performance may be limited by the performance of intermediate servers. In fact, the system may be saturated even though none of the devices are fully utilized. Performance can be improved by minimizing the blocking time at the client, at the server, or at both, shown in Figure 1.1. Performance can also be improved by using "multi-threaded" clients, shown in Figure 1.2. The various techniques are described below.

1.3.1 Server-Side Performance Improvement Techniques

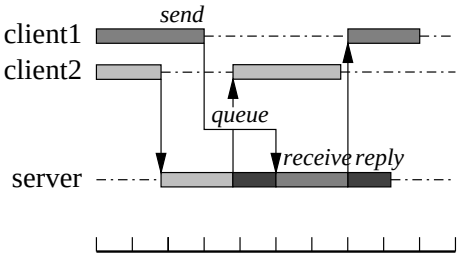
System performance can be improved at the server in two ways. First, the server may reply to the client before the request is actually completed, thus allowing the client to resume operation (Figure 1.1(b)). For example, a file server that replies to a client's write request once the data is received but before the data is written to the disk is one potential application of this technique. Second, the server process may be duplicated or "cloned". Queues that formed at the server will be pushed down to lower level servers (Figure 1.1(e)).

1.3.2 Client-Side Performance Improvement Techniques

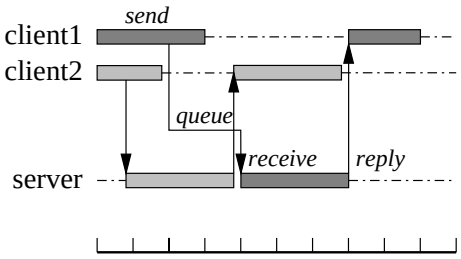
Performance in the system may also be improved by making changes at the client. Figure 1.1(c) shows the effect of an asynchronous remote procedure call. Again using a file server as an example, the client issues a read request well ahead of the point in time where it needs the data. The client continues execution, blocking only when the data is actu-



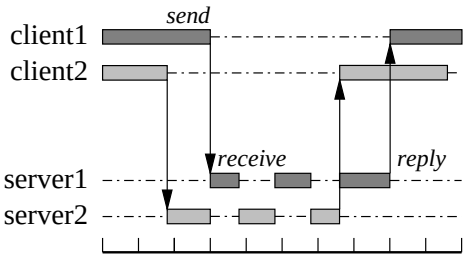
(a) A simple remote procedure call.



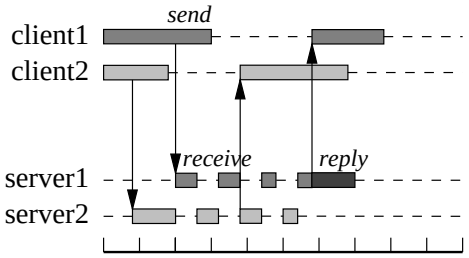
(b) Early Reply: reply to client before completing request.



(c) Asynchronous RPC: send to server before needing reply.



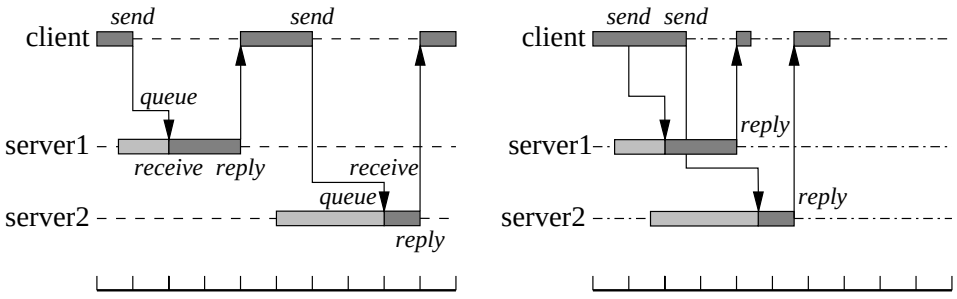
(d) Multiple Servers: Replicate server so clients don't queue.



(e) All Improvements together.

Figure 1.1: Techniques to reduce response times in client-server systems. Two clients running on separate processors make requests to one or more servers running on one processor. The time scale at the bottom of each figure shows how each technique can potentially improve performance.

ally needed. Performance may also be improved by exploiting parallelism in the client. Figure 1.2 shows how the performance at a client that makes requests to separate servers can be improved using parallelism. In Figure 1.2(a), the client waits for each request to complete. Figure 1.2(b) shows the performance improvement by using separate threads of execution for each request.



(a) Serial client (standard RPC)

(b) Parallelism within client (asynchronous RPC)

Figure 1.2: Using client parallelism to improve RPC performance. One client makes a request to two servers. Traffic from other clients competing at the servers is represented by the light shading. The servers run on separate processors.

1.4 Contributions

The contributions of this work are described in the sections below.

1.4.1 Object Oriented Solver

A new object-oriented solver has been designed and implemented that incorporates many of the features of the client-server queueing network solvers that preceded it (see Chapter 3). The solver also incorporates a new submodel construction strategy which generalizes submodel construction.

1.4.2 Accuracy of Solutions

The accuracy of solutions is improved in two ways.

1. The interlocking calculation that compensates for correlated traffic flow from common sources and is now more generalized (see Chapter 4).

2. The overtaking calculation, used for models that incorporate early replies (see Figure 1.1(b)), now takes into account the effects of both multi-phase clients and servers (see Chapter 5).

1.4.3 Modelling Power Enhancements

The modelling power of the solver is broadened in several ways.

1. The model now includes forwarding. Forwarding can be used to model systems that employ asynchronous messaging with synchronous messaging semantics (see §3.3.1).

2. New analytic approximations are incorportated into multiservers to allow for two phases of service (see §6.3).

3. The model now incorporates *activities*. Activities are components that represent the lowest level of detail in the performance model. They extend the performance model because they can be connected together not only sequentially, but with fork and join interactions as well. This change broadens the modelling power substantially because the existing client-server queueing network solvers can only solve models with sequences of activities.

4. The analytic solver can now solve models with intra-task fork-join interactions, permitting the analysis of client-server systems with all of the interaction patterns shown earlier in Figures 1.1 and 1.2.

1.4.4 Case Study

The solver has been used to analyze the performance of the Linux V2.0 Network File System implementation.

1.5 Organization of Thesis

The thesis is organized as follows. Chapter 1 is a brief introduction of the client-server computing paradigm and a broad overview of the subject of performance modelling. Chapter 2 describes performance modelling in more detail, with emphasis on analytic client-server performance models. It also describes previous work in queueing networks with fork-join behaviour. Chapter 3 introduces the Stochastic Rendezvous Network model, the basis for the work in the thesis. Chapter 3 also describes improvements to many of the algorithms in the performance model. Chapter 4 describes interlocking: why it arises in layered queueing models, and how the solver compensates for the effect. Chapter 5 describes the new overtaking approximation. Chapter 6 describes and evaluates the multiserver algorithms used in the solver. It also describes the changes to the various multiserver algorithms needed to incorporate second-phase effects. Chapter 7 describes important extensions to the original input model using *activities*. Activities can be connected together in a more general fashion, broadening the modeling power of the solver. Chapter 8 describes the *fork-join* approximation used by LQNS to solve performance models incorporating heterogeneous threads. In Chapter 9, the solver is then used to evaluate the performance of the Network File System (NFS) implementation in the Linux version 2. kernel. Finally, Chapter 10 contains conclusions.

Chapter 2

Extended Queueing Networks

Analytic performance modelling takes many forms, from simple bounds analysis to the evaluation of complex Markov Chains. Analytical techniques are a popular method for doing performance studies, despite potential accuracy problems, because they can be performed both quickly and cheaply. This chapter supplies a brief overview of analytic modelling with origins to the solution of a Markov chain. In particular, the technique known as "mean value analysis", which is used to solve product-form queueing networks, is described. Next, the limitations of mean value analysis, and ways to overcome them, are discussed with emphasis on client-server queueing systems. Finally, techniques are described for solving queueing networks with fork-join behaviour.

The layered queueing or stochastic rendezvous network model described in this thesis can be viewed as a standardized formulation for extended queueing networks (EQN's), particularly suited for a wide variety of distributed server systems.

2.1 Markov Modelling

Markov modelling involves the construction and solution of a Markov chain representing the system being studied. Tools now exist to automate this process, for example, MARCA [133,

69], GreatSPN [17, 18], SPNP [22] and UltraSAN [122]. However, Markov models have three major problems which limits their use. First, they suffer from state space explosion for all but the smallest of models. Research is continuing to ameliorate this problem, either by exploiting symmetry in the Markov chain, or by decomposing the problem into smaller units and iterating between solutions. Second, chains with huge differences in their transition rates cause numerical instability during solution (this problem is referred to as "stiffness"). Finally, systems which have purely deterministic services times or service time distributions without a rational Laplace transform can be difficult to model accurately. Of the three, the state explosion problem is the one that most often limits the use of this technique.

2.2 Mean Value Analysis

Queueing networks with certain restrictions were found to have a "product form" solution which led to computationally efficient techniques such as Convolution and Mean Value Analysis. Mean Value Analysis has been found to be a particularly popular technique for solving product-form queueing networks because it is reasonably efficient, often robust when assumptions are violated, and intuitive. This section describes briefly product form queueing networks, the mean value analysis technique, the assumptions for its use which can be limiting for certain models, and ways to accommodate these assumption when they are violated. Sections that follow then describe the application of mean value analysis to client-server queueing systems.

2.2.1 Product-Form Queueing Networks

Theory in the solution of networks of queues made rapid progress after the publication of the work by Jackson [57]. He found that solution to the underlying Markov chain for a restricted queueing network had a "product-form" solution. This work was extended by others with Baskett et. al. summarizing the results for a general class of product-form

networks [4].

The service centers in product form networks exhibit a property called "local balance" [91] which permits computationally efficient solution algorithms for moderately sized networks. The first computationally efficient algorithm for solving closed networks was published by J. P. Buzen [10]. However, this method, called convolution [105], often has numerical stability problems.

2.2.2 Exact Mean Value Analysis

Reiser developed a new approach to solving product form queues, based on the arrival theorem, which gives (2.1) and similar results. This approach is called Mean Value Analysis (MVA) [106, 104]. The basis of the algorithm is solving for the waiting time for chain k at each service center m using

$$W_{mk}(\mathbf{N}) = D_{mk}\left[1 + \sum_{j=1}^{K} L_{mj}(\mathbf{N} - \mathbf{e}_k)\right] \tag{2.1}$$

D_{mk} represents the demand at station m for chain k, $\mathbf{N}$ is a vector of customers and $L_{mj}(\mathbf{N} - \mathbf{e}_k)$ is the average queue length at station m with one less customer from chain k present in the network. Once the waiting time is found, the throughputs for all chains can be found, then through Little's law [79], queue lengths. The exact MVA algorithm for multiple routing chains K is given in Figures 3.18 and 3.19 on pages 66 and 67 respectively.

Equation (2.1) requires the queue length at the service center when there is one less customer of chain k present. Exact MVA therefore solves Equation (2.1) for all customer populations, $\mathbf{N}$, starting with zero customers. Because it is necessary to sequence through all customer populations, exact MVA becomes prohibitively expensive for networks with moderate numbers of stations, routing chains and customers. Table 2.1 summarizes the costs in terms of space and time complexity.

Other variations of mean value analysis exist which help in special cases, such as *Mean*

$$\begin{array}{lll} \text{time:} & MK\prod_{k=1}^{K}(N_k+1) & \text{operations} \\ \text{space:} & M\prod_{k=1}^{K}(N_k+1) & \text{storage locations} \end{array}$$

Table 2.1: Storage and operations cost of exact MVA for M service centers, K routing chains and N_k customers in chain k.

Value Analysis by Chain [25], and *Distribution Analysis by Chain* [27]. However, these algorithms also suffer when there are large numbers of stations and customers in the queueing network.

2.2.3 Approximate Mean Value Analysis

Because exact MVA becomes prohibitively expensive for moderate numbers of stations, customers and chains, approximation techniques were devised to estimate the queue lengths at the reduced customer levels, thus eliminating the need to sequence through all customer populations.

Bard [3] and Schweitzer [125] developed the first approximation technique by replacing $L_m(\mathbf{N}-\mathbf{e}_k)$ in Equation (2.1) with

$$L_{mk}(\mathbf{N}-\mathbf{e}_j) = \begin{cases} L_{mk}(\mathbf{N}) & j \neq k \\ \frac{(N_j-1)}{N_j} L_{mj}(\mathbf{N}) & j = k \end{cases} \tag{2.2}$$

and iterating until convergence.

Later, Chandy and Neuse [12] developed the Linearizer algorithm. $L_{mk}(\mathbf{N}-\mathbf{e}_j)$ is estimated using the following:

$$F_{mk}(\mathbf{N}) = L_{mk}(\mathbf{N})/N_k \tag{2.3}$$

$$D_{mkj}(\mathbf{N}) = F_{mk}(\mathbf{N}-\mathbf{e}_j) - F_{mk}(\mathbf{N}) \tag{2.4}$$

$$L_{mk}(\mathbf{N}-\mathbf{e}_j) \quad = \quad (\mathbf{N}-\mathbf{e}_j)_k(F_{mk}(\mathbf{N})+D_{mkj}(\mathbf{N})) \tag{2.5}$$

(Equation (2.5) is equivalent to (2.2) when $D_{mkj}(\mathbf{N}) = 0$.) Estimates for $D_{mkj}(\mathbf{N})$, the difference in queue length when a customer is added, are found by using the Schweitzer approximation with one and two customers removed from the network.

The linearizer algorithm is generally more accurate than Schweitzer's approach because the estimate for $D_{mkj}(\mathbf{N})$ is better than zero. However, linearizer is also more expensive because the Schweitzer approximation must be run at the full customer population and with one customer removed from each routing chain. The cost of the algorithms are summarized in Table 2.2.

	Exact MVA	Schweitzer	Linearizer
Operations	$MK\prod_{k=1}^{K}(N_k+1)$	$\tilde{I}M$	$\tilde{I}M(3+2K)$
Storage	$M\prod_{k=1}^{K}(N_k+1)$	MK	$\frac{MK^2}{2}$

Table 2.2: Storage and Operation cost comparison for the exact and approximate MVA techniques for M service centers, K routing chains and N_k customers in chain k. $\tilde{I}$ represents the number of iterations needed for convergence with the approximate solution techniques.

2.2.4 Assumptions for Product Form Queueing Networks

Product-form queuing networks have been used successfully to solve performance models for a large variety of systems. However, certain assumptions are made about the model in order to use the more computationally efficient solution techniques. Service centers will be assumed to be of specific types:

- Single server, first-come, first-served with exponential service times. The mean service times for all chains must be identical.

- Single server, last-come, first-served, random or processor sharing service. Service time distributions can be general and different routing chains can have different service times.

- Load dependent service, first come, first served only. The service rate is dependent on the number of customers at the station only.

Second, the population of a closed product form network is fixed, although open and mixed networks permit the customer population to vary. Finally, customer routing cannot be dependent on the state of the network. These assumptions may seem overly restrictive for a wide variety of real systems. However, in many models where they are violated the models still produce approximate but sufficiently accurate results for practical use.

Two areas where product-form queueing networks cannot be used directly are systems with simultaneous resource possession, and systems with fork-join behaviour. Systems with simultaneous resource possession cannot be modelled directly because the service rate at a center is dependent on other centers in the network. Systems with fork-join behaviour violate the fixed customer and routing homogeneity assumptions in a closed queueing network because forking generates customers while joining eliminates them. This work uses the the method of surrogates to adapt MVA for systems with these behaviours.

2.2.5 The Method of Surrogates

Simultaneous resource possession arises in a number of places in computer systems, for example:

- limited multiprogramming due to memory capacity or channel contention,

- lock contention in data base systems, and

- remote procedure calls.

(The last example is of particular importance because it is the inter-process communication method typically used in client server systems.) If the effects of simultaneous resource possession are ignored, the throughput estimates from a performance model will be overestimated because the time needed to acquire resources is not accounted for.

To solve this problem in a Mean Value Analysis framework, Jacobson and Lazowska [58] developed the *"method of surrogates"*. It accounts for the time needed to acquire simultaneously held resources by splitting the original model into two submodels and iterating between the two. Each of the submodels includes an explicit representation of one of the simultaneously held resources and a delay center representing other. The waiting time solution for the explicitly modelled resource in the first model is used for the service time for the delay center in the other and vice versa. The iteration continues until the difference in waiting time between solutions becomes negligible.

The method of surrogates was generalized by de Souza e Silva and Muntz [29] to handle resources by customer chain and to handle nesting of resources. Jenq [60] demonstrates the technique for a substantial distributed transaction testbed system with the effects of the concurrency control protocol, the transaction recovery protocol, and the commit protocol all modelled and validated against measurements.

2.3 Client-Server Models

Multi-tier client-server computer systems cannot be modelled directly using mean value analysis because the blocking from nested remote procedure calls is a form of simultaneous resource possession. This problem is overcome in the Stochastic Rendezvous Network Model (SRVN) [147, 149, 152], Stochastic Rendezvous Networks by Task-Directed Aggregation (SRVN-TDA) [100, 97, 101, 98], the Method of Layers (MOL) [112, 113, 110], and by Ramesh and Perros [103] using hierarchical decomposition and the method of surrogates. These approaches break a multi-tier client-server system into a collection of two-layer systems

solve these systems individually, then use the results as the input to the other two-layer systems. While the overall approach between the methods is similar, they differ substantially in their implementation. The sections that follow describe each method in greater detail using the system in Figure 2.1 as an example.

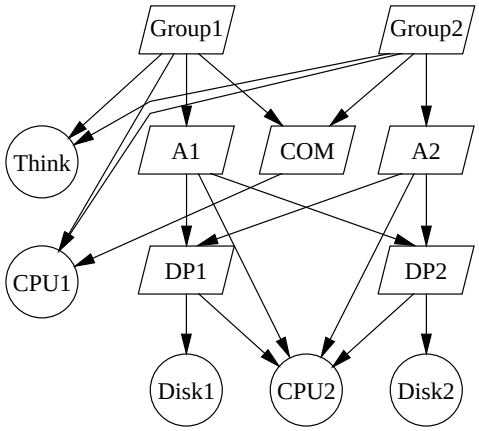


Figure 2.1: Example multi-tier client-server system from [113]. Tasks are represented by parallelograms. Pure servers, such as devices and think times for customers, are represented by circles. The customers themselves are represented by the tasks *Group1* and *Group2*.

Kurasugi and Kino [71] have also developed a technique for solving a two-level client-server queueing model. They too use hierarchical decomposition, but use *flow-equivalent service centers* instead of surrogate delays. This technique is described in greater detail in §2.3.5.

Finally, the problem of blocking at software servers has also been addressed by Fontenot with his *mobile server model* [35]. This technique relies on a new residence time expression for open queueing systems called the *hyperbolic model*. However, the technique appears limited because it can only nest to one level (i.e., it can only model two-tier client-server systems), it appears to be sensitive to the arrival rate distributions from other streams, and it has not been applied to closed queueing networks. Because of these limitations, this technique will not be discussed further.

2.3.1 Stochastic Rendezvous Network Model

The Stochastic Rendezvous Network Model [147, 149, 152] consists of an acyclic graph of clients and servers. Clients and servers are collectively referred to as tasks, which are used to model users, devices, and software processes. Requests from one task to another use the remote procedure call paradigm [143, 5]: i.e., clients are blocked until the server responds.

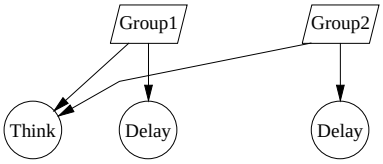
The overall model is solved by first constructing a set of submodels each consisting of only one server and a set of clients and their surrogate delays. For example, Figure 2.2 shows the MVA submodels for each of the servers in Figure 2.1. The clients in each submodel are found by searching for all callers to the particular server and are treated as unique routing chains with populations based on the number of instances of the client task. For single-threaded tasks, the number of instances is one, while for multi-threaded tasks, the number of instances is the maximum number of threads that can be active at one time.

Next, the overall model is solved by applying one-step MVA to each of the submodels. A variation of the Bard-Schweitzer MVA approximation [3, 125] is used where the waiting time expression, (2.1), is modified so that the queue length $L_{mk}(\mathbf{N} - \mathbf{e}_k)$ is found using arrival instant probabilities instead of simply scaling $L_{mk}(\mathbf{N})$ based on a fraction of customers in the system. Throughput results from each submodel are then used to adjust the surrogate delays in all of the other submodels. The solution iterates among all the submodels until convergence criteria are met.

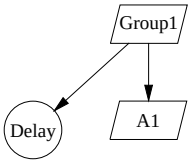
The Stochastic Rendezvous Network Model is more formally discussed in Section 3.1 as it and the Method of Layers are the basis of the modifications being proposed here.

2.3.2 Task Directed Aggregation

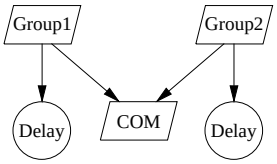
The key to the SRVN approach is in the estimation of the arrival instant probabilities. Petriu improved the accuracy of the estimate through her technique called "task directed aggregation" [100, 97, 101, 98]. This technique uses a Markov submodel to derive the



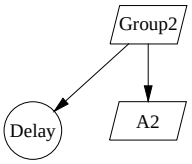
(a) Submodel 1



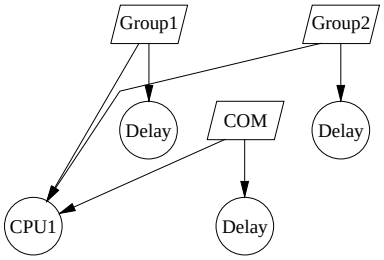
(b) Submodel 2



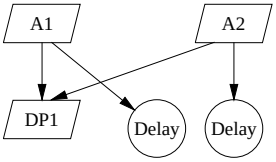
(c) Submodel 3



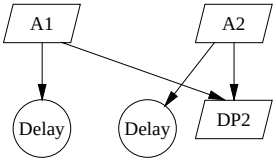
(d) Submodel 4



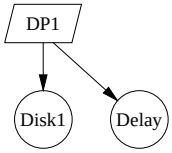
(e) Submodel 5



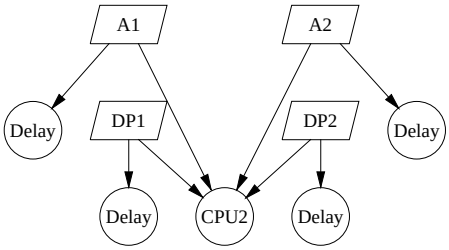
(f) Submodel 6



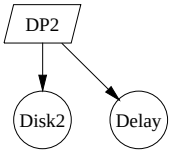
(g) Submodel 7



(h) Submodel 8



(i) Submodel 9



(j) Submodel 10

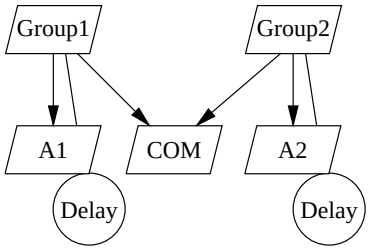
Figure 2.2: Stochastic Rendezvous Network solution partitioning for the example in Figure 2.1. The circles marked *Delay* are surrogate delays introduced during the solution of the model.

in-service and in-queue probabilities rather than using the probabilistic approach used by Woodside et. al. However, the approach is somewhat more computationally expensive.

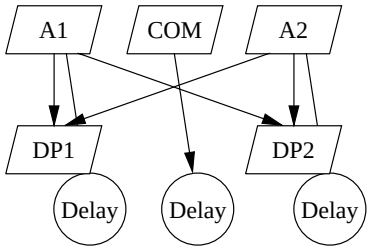
2.3.3 Method of Layers

The Method of Layers (MOL) [112, 113, 110] solves client-server queueing networks by decomposing the network into a set of two level MVA submodels then solving each of these models using the linearizer algorithm [12]. Each submodel forms a conventional product form queueing network where the servers form the stations and the clients form the customers. The MVA submodel are constructed first by splitting the input model into two submodels, one for hardware contention and the other for software contention. The hardware contention submodel consists of all of the tasks and devices from the input model; the software tasks act as clients and the devices act as servers. Next, the software contention submodel, using only the tasks from the input model, is sorted into *layers*. Software submodel n is then constructed by using all of the tasks in level n as clients, and all of the tasks from level $n+1$ as servers. The final set of submodels for the example in Figure 2.1 is shown in Figure 2.3.

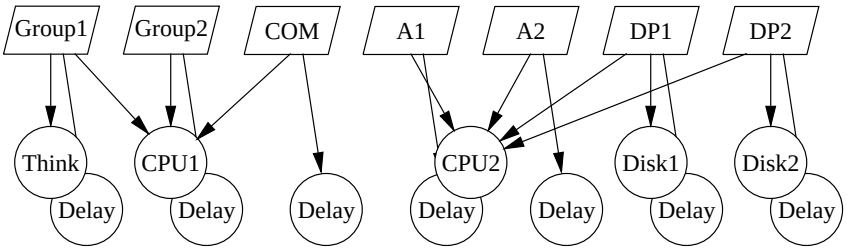
The Method of Layers estimates the performance of the system under study by iterating among the various submodels. It begins by solving the software submodels from submodel 1 to submodel $N-1$. (There is no software submodel N because the pure servers at level N make no requests). Once the software submodels have converged, the performance results are used to set the think and service times for the tasks in the hardware model. The performance estimates from the solution of the hardware model are then used to set the service times for the various software submodels. This sequence continues until the desired convergence criteria are met.



(a) Software Contention Submodel 1



(b) Software Contention Submodel 2



(c) Hardware Contention Submodel

Figure 2.3: Submodels used by the Method of Layers for the example in Figure 2.1. The circles marked *Delay* are surrogate delays introduced during the solution of the model.

2.3.4 Client-Server Queueing Network Model

The Client-Server Queueing Network model of Ramesh and Perros [103] decomposes a strictly-layered model into a set of two-level submodels similar to the approach used by the Method of Layers. The clients and the servers in each submodel may have Coxian distributions, in which case each submodel is further divided up into a set of sub-submodels consisting of exactly one client.

The solution of the network consists of a backward and a forward pass. The backward pass starts from the lowest level server submodel and finds the first three moments of waiting time to its clients. These values are used to set the service time of the servers in the next higher submodel. This pass terminates when the servers at level two are evaluated (level one consists of clients only). Next, the clients at level 1 are solved using a $C_2/C_2/1$ queue. The solution to this queue is used to set the arrival process to the servers in level 2. The servers in level 2 are then re-solved to set the arrival processes for the servers in level 3, and so on. The algorithm terminates when the difference in the arrival process probabilities is sufficiently small. Otherwise, the backward and forward passes are repeated.

2.3.5 Extended Flow-Equivalent Queueing Models

The Extended Flow-Equivalent Queueing Models of Kurasugi and Kino [71] decomposes a three-level, strictly-layered model into two submodels. The models are constructed in a manner similar to the method of layers (i.e., the intermediate-level servers are customers in the lower submodel, and queueing stations in the upper submodel). The lower submodel is a closed queueing network with multiple routing chains. This model is solved for all customer populations once; these solutions are used to either set the transition rates for a Markov chain, or to set the service rates of extended flow-equivalent service centers in the upper model.

The benefit of this approach is that it is non-iterative, unlike all of the preceding tech-

niques. Unfortunately, this advantage will become a hindrance if there are a large number of routing chains with a large number of customers because of the need to calculate solutions to all of the intermediate population values in the lower submodel. The authors only consider three-level models at present. However, it appears the technique can be extended to models with more layers.

2.4 Queueing Networks with Fork-Join Interactions

Forking describes the phenomenon where a single thread of control within a process branches out into two or more independent threads of control that can potentially execute in parallel. *Joining* is the reverse operation: multiple threads of control all wait for each other to complete at which time a single thread of control continues on.

Computer systems with fork-join behaviour cause two problems for mean-value based performance analysis. First, the number of customers (threads of control) varies with time. Second, the join delay must be found. These problems preclude the direct application of exact MVA. However, several authors have solved systems with forks and joins using mean value analysis, either by setting up a Markov Chain to describe the customer states and using MVA for each state in the chain, or by estimating the join delay and using a surrogate delay in an MVA model. The sections that follow briefly outline these approaches in a roughly chronological order.

There are a lot of references on fork-join systems. The following are just a few that are most relevant to the present work as closed models are considered here.

2.4.1 Heidelberger and Trivedi

Heidelberger and Trivedi developed two techniques for systems with fork-join behaviour [51]. The systems under study consisted of a set of homogeneous jobs which forked, then joined.

The first technique used hierarchical decomposition with two models. The upper level

model consisted of a Markov chain where each state represented a set of tasks executing concurrently. The lower level model consisted of a set of closed product form queueing networks, one for each state in the Markov Chain. First, each of the queueing networks was solved. Next, the results from the queueing networks were used to set the transition rates for the Markov Chain. Finally, the Markov Chain was solved giving the overall response time for the system.

The second technique developed by the authors employed the method of surrogate delays. The system was modelled using one product form queueing network with two additional infinite servers. The first infinite server was used to model the serial portion of the overall system (i.e., when only one job was running). The second infinite server was used to model the synchronization delay. The solution to this model was then used to derive new values for the synchronization delay. These steps were repeated until the change in the synchronization time reached some small value.

The value of the synchronization delay is the maximum value of all of the response times, W_i, of the tasks involved in the join. These times were assumed to be exponentially distributed with mean values of $\lambda_i = 1/W_i$. Therefore, the synchronization delay was

$$\begin{aligned} \mathrm{Max}_W &= \sum_{i=1}^{N} \frac{1}{\lambda_i} - \sum_{i<j} \frac{1}{\lambda_i + \lambda_j} + \sum_{i<j<k} \frac{1}{\lambda_i + \lambda_j + \lambda_k} \\ &\quad - \cdots + (-1)^{N-1} \sum_{i_1 < i_2 \ldots < i_N} \frac{1}{\lambda_i + \cdots + \lambda_N} \end{aligned} \tag{2.6}$$

This technique was used to model a task which forked into a number of sub-tasks, then joined. Of the two approaches, the hierarchical decomposition technique was found to be more accurate when compared to simulation (it had a mean relative error of 1% versus 5% for the other method). The solution times of both techniques were found to be comparable.

2.4.2 Thomasian and Bay

Thomasian and Bay [138, 139] use hierarchical decomposition to find the response time of a distributed system with fork-join interactions. Multiple fork-join interactions can take place, unlike the models studied by Heidelberger and Trivedi. All tasks are assumed to join, thus completing a cycle in the overall model.

The performance model consists of two components. The top-level component is a Markov chain which is used to find the overall response time for the system. Each state in the Markov chain corresponds to the number of active tasks in the system. The second component is a set of queuing network models, one for each state in the Markov chain. The solution to each queueing network model is used to find the transition rates for the corresponding state.

This technique appears to be quite accurate. However, the computational complexity can become prohibitive due to the number of queueing network models that may have to be solved. The cost of the technique is reduced somewhat by exploiting state aggregation in the Markov chain where possible, and by further decomposing the model.

2.4.3 Chu, Sit and Leung

The approach taken by Chu et. al. [19, 20, 21] uses two submodels to find response times in a distributed system, one to find contention delays at devices, and one to find the overall response time for requests. The overall system is considered to be open, i.e., once a task finishes a request, it is free to accept a new one. The upper-level model describes the precedence relationships among the tasks. This model is used to aggregate the response times from the lower level model to find the overall response time for requests to the system. The lower-level model consists of an extended queueing network model which is used to find the response time for each task in the system. Tasks in the system are assumed to be independent and execute concurrently where possible. Furthermore, since the overall

system model is open, the population for the lower-level model (i.e. the tasks in the upper level) is fixed, unlike the models by Heidelberger and Trivedi, and Thomasian and Bay. The authors also develop a Markov Chain based method for solving networks with simultaneous resource possession. The solution provides the first two moments of response time, used in the following step.

Synchronization delays, needed by the upper level model, are found by finding the mean and variance for the function $Y = \max(y1, y2, ..., y_n)$ where y_i is the response time for thread i in a fork-join interaction. To find the moments, the fork-join graph is repartitioned into a set of fork-joins each with exactly two threads. The mth moment for each of the new graphs is then found using

$$E[Y^m] = \int_0^\infty y^m F_1(y) f_2(y) dy + \int_0^\infty y^m F_2(y) f_1(y) dy, \quad m = 1, 2, ... \tag{2.7}$$

where $f_i(y)$ is the probability density function and $F_i(y)$ is the probability distribution function for thread i. Threads can have Erlangian, exponential, or hyper-exponential distributions

2.4.4 Nelson, Towsley and Tantawi

Nelson, Towsley and Tantawi model [90] a system where an open stream of jobs arrive at a processor complex, fork, then possibly join. The system is modelled as a bulk arrival $M^X/M/c$ queueing system and solved using a continuous time Markov Chain. The solution for the overall system is complicated by the fact that there is no closed form expression for the service completion time – a set of recurrence expressions must be solved numerically instead.

This solution is applicable to the class of systems where a job forks then joins on a pool of processors. It is not suitable for systems with more complex fork-join patterns, nor for systems with heterogeneous threads.

2.4.5 Mak and Lundstrom

Mak [85] and Mak and Lundstrom [84] solved series-parallel task systems using only one queueing network model by modifying the MVA waiting time expression (2.1) to account for the probability that tasks interact with one and another. The overlap probabilities are estimated by reducing the task graph after the queueing network model is solved. The queueing network solution and graph reduction steps are repeated until the estimate of mean task completion time converges. This technique is much the second approach used by Heidelberger and Trivedi [51] except that extra delay centers are not incorporated explicitly in the model. This method may not be as accurate as solving a Markov chain representing the task precedence model, however, it does not suffer the problem of state space explosion inherent in Markov chain based solutions.

As mentioned earlier, the MVA waiting time expression (2.1) is modified to account for "overlap" in the execution of two or more tasks[1]. Furthermore, (2.1) is simplified by constraining each chain in the queueing network to exactly one customer; the term $\sum_{j=1}^{K} L_{mj}(\mathbf{N} - \mathbf{e}_k)$ is replaced with $\sum_{j=1, j\neq k}^{N} L_{mkj}$. Equations (2.8) through (2.11) show the modified waiting time expressions:

$$W_{mk} = D_{mk}(1 + A_{mk}) \tag{2.8}$$

$$A_{mk} = \sum_{j=1, j\neq k}^{N} L_{mkj} \tag{2.9}$$

$$A_{mk} \approx \hat{A}_{mk} = \sum_{j=1, j\neq k}^{N} \frac{p_{kj} d_{kj}}{W_k} L_{mkj} \tag{2.10}$$

$$L_{mkj} = \frac{W_{mkj}}{\sum_{i=1}^{K} W_{mki}} \tag{2.11}$$

The term p_{kj} is the *overlap probability* and d_{ij} is the *overlap duration*, described below.

[1] This approach is much like that used to remove contention due to "interlocking", as described in Section 4.2.

Equation (2.11), an approximation of the reduced population residence time, also exploits the constraint that there is only one customer in each chain.

The *overlap probability*, p_{kj}, for tasks k and j with a start time S and and end time E is estimated using:

$$p_{kj} = 1 - \Pr(E_j < S_k) - \Pr(E_k < S_j) \tag{2.12}$$

Let A and B be two independent nonnegative continuous random variables, and $f_A(x)$ and $F_B(x)$ be their respective probability density and distribution functions, then

$$\begin{aligned} \Pr(A < B) &= \int_0^\infty \Pr(B > x) f_A(x) dx \\ &= \int_0^\infty [1 - F_B(x)] f_A(x) dx \end{aligned} \tag{2.13}$$

If the distributions of A and B are assumed to be of the Erlang type with r stages of $1/\lambda$ duration, then (2.13) simplifies to:

$$\Pr(A < B) = \left(\frac{\lambda_A}{\lambda_A + \lambda_B}\right)^{r_A} \sum_{k=0}^{r_B - 1} \left(\frac{\lambda_A}{\lambda_A + \lambda_B}\right)^k \cdot \frac{(r_A + k - 1)!}{(r_A - 1)! k!} \tag{2.14}$$

The overlap duration is the amount of time two tasks can execute concurrently given they overlap to begin with. The residence times of the tasks are assumed to be exponentially distributed, therefore

$$d_{ij} = \frac{1}{\lambda_i + \lambda_j} \tag{2.15}$$

The prediction method was found to be both accurate and computationally efficient. The time complexity is $O(N^2M + N^3)$ per iteration and the space complexity is $O(N^2M)$ where N is the number of tasks, and M is the number of servers.

2.4.6 Liu and Perros

Liu and Perros [81, 82] use decomposition and aggregation to find the response time of a closed queueing system with fork-join interactions. To solve a system with a K-way fork-join interaction, with $K > 2$, the K-way fork-join is reduced to a 2-way fork-join through aggregation, ignoring the serial part of the queueing network. The aggregated 2-way fork-join network is then combined with the serial portion of the original queueing network to form a new queueing network which is solved numerically.

The technique produces a lower bound for the system's throughput for a heterogeneous fork-join network. For a homogeneous fork-join network, the error introduced by the aggregation is proportional to the number of branches present. The accuracy can be improved by solving the system numerically for $k = 2$ and $K = 3$, and with the approximation technique for $K = 3$ to find a scaling constant that can be applied to a network with an arbitrarily large K.

2.4.7 Rolia's Synchronization Server

The Method of Layers [112, 113, 111] allows two tasks to synchronize on a third special task called a "sync provider". This task is an explicit representation of the surrogate delay for synchronization in the join operation; the delay itself is found using (2.6). Waiting times are assumed to have hyper-exponential distributions. A source with this distribution can be modelled as two sources with exponential distribution which permits higher accuracy in the overall solution while retaining the simplicity and speed of the join-delay calculation.

The synchronization server allows two or more remote procedure calls from disjoint sources to synchronize with each other. Since RPC's block the solution is feasible, unlike the case with two disjoint asynchronous input streams. There is no provision in the model to permit forking.

2.4.8 Jonkers

The Generalised Architecture Modelling with Stochastic techniques methodology, GLAMIS, combines general task graphs with separable closed queueing networks [63, 65, 64, 66]. Task graphs are used to model interactions among tasks. Closed queueing networks are used to model contention among devices. Task service times are assumed to be deterministic, unlike the other methods described here. The task graph is of the series-parallel type.

Two techniques were developed by Jonkers and his colleagues. The first approach used a surrogate delay to represent the synchronization delay in the system [63, 65]. Initially, the closed queueing network model is solved ignoring the effects of synchronization in the task graph. The task graph is then solved to find the synchronization delays along each path from the fork to the join. Since the service times at the tasks in the upper level model are deterministic, the expression used to find the join delay is exceptionally simple. These delays are then incorporated into the queueing network model. These steps are repeated until some convergence criteria are met.

The second approach solves a number of closed queueing network models, each with one less set of customers than the previous [64, 66]. Initially, all tasks the the system are modelled. The path in the task graph with the shortest response time, i.e., when the most contention is present in the system, is then removed. The service times for all remaining tasks are then adjusted to remove the overlap time; the modified queueing network is then solved. This process is repeated until there are no customers left.

Replication is exploited to speed the solution. Furthermore, services centers have deterministic service times. Equation (2.16) shows the expression used to approximate the service rate at a center with m replicated servers, and n customers.

$$\mu_m(n) = \frac{1}{S}\sum_{j=0}^{n-1}\left(\frac{m-1}{m}\right)^j \quad \text{if } m \geq n, \quad \mu_m(n) = \frac{1}{S}\sum_{j=0}^{m-1}\left(\frac{n-1}{n}\right)^j \quad \text{if } m \leq n, \qquad (2.16)$$

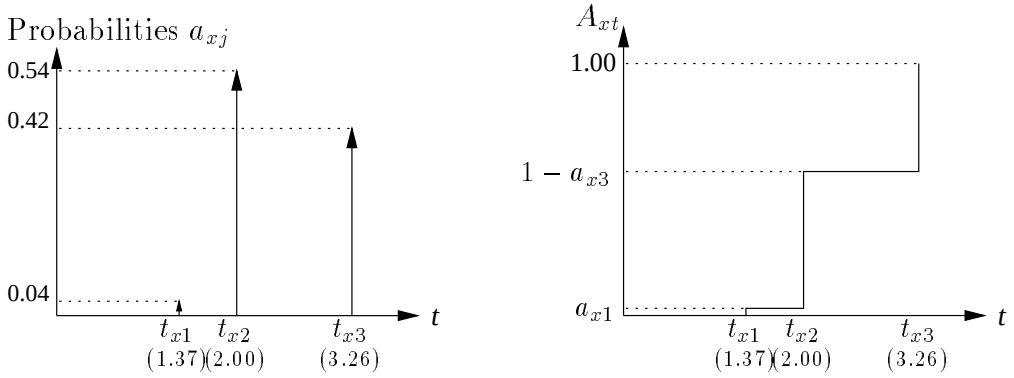
This approximation was found to be superior to the exact MVA expression which assumes

exponential service times.

2.4.9 Woodside, Jiang, Hubbard

In many of the methods described here, it is necessary to find the result of the expression $Y = \max(y_1, y_2, ..., y_n)$ where y_i is a random variable describing the response time for a thread in a fork-join interaction. Two problems arise. First, the distribution of the random variables, y_i, is not known; mean value analysis only supplies means. Second, assuming second moments are available, numerical integration of (2.7) can be costly.

Given that second moments for response times can be estimated (see [21, 152, 113]), Woodside et. al. estimate synchronization delays using a three point approximation [150, 62], shown in Figure 2.4. This technique is much faster than numerically integrating (2.7). However, accuracy suffers for some distributions.



(a) Approximate distribution for a branch x.

(b) Distribution Function

Figure 2.4: Three-point approximation for a distribution with $\overline{t_x} = 2.0$ and $\sigma_x^2 = 0.4$.

2.4.10 Huang et. al.

Huang et. al. constructed a model of a Multicube multiprocessor system and its cache coherence protocol [56] and used mean value analysis to solve it. The primary difficulty that arose in modelling this system was that of forking. The authors modelled the forking behaviour by treating the forked traffic stream as an open class in the MVA model. The arrival rate of the open class was then set by the solution of the closed model for the system. The resulting mixed model was then solved iteratively until convergence[2]

2.4.11 Lee and Katz

Fork-Join interactions are found in Redundant Arrays of Inexpensive Disks (RAID) [13] because a read or write operation requires the completion of reads and writes to multiple disks before the originating request can return.

Lee and Katz [78] derive a performance model for a RAID level 5 disk array with left-symmetric parity placement. The inputs to the model consist of L processes making requests of size n to the disk array. An exact expression for the disk array utilization is found which is then simplified to

$$U \approx \frac{1}{1 + \frac{1}{L}\left(\frac{1}{p} - 1\right)} \tag{2.17}$$

where p is the probability of a request accessing a given disk. Little's result is then used to find throughputs and response times.

This model is quite accurate and insensitive to disk service times. Further work is needed to allow the L customers in the model to have think times instead of constantly issuing requests to the disk array.

[2]This technique is identical to the solution of send-no-reply type requests in Stochastic Rendezvous Networks.

2.4.12 Varki

Varki [140] considers a queueing system consisting of a sequential and parallel subsystems. A parallel subsystem consists of $K > 1$ identical queueing systems. The technique transforms a parallel subsystem into a flow-equivalent service center with state-dependent rates, the state being the number of jobs in service as a result of a fork.

Results in the paper show the technique is very accurate, with errors of less than 5% in most cases. Further work is needed to extend the technique to multi-class networks and to networks with more general parallel subsystems.

Chapter 3

The Layered Queueing Network Solver

The Layered Queueing Network Solver (LQNS), developed in the present research, combines the strengths of SRVN and MOL solvers to broaden the modelling scope and to improve the accuracy of solutions to layered queueing networks. This chapter first describes the semantics of layered qeueing networks, then follows with a description of the Layered Queueing Network Solver.

3.1 The Stochastic Rendezvous Network Model

A Stochastic Rendezvous Network Model consists of the inputs *tasks, entries,* and *phases*, and the output, *throughput.* Tasks represent hardware and software objects which may execute concurrently, entries differentiate service demands at the tasks and phases denote different intervals of service within entries. Requests for service are made from entry to entry through send-receive-reply message interactions. Tasks do not possess internal concurrency, a deficiency which will be addressed in Chapter 7. Table 3.1 shows the icons used in the model.

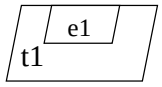
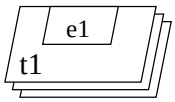
Name	Icon	Description
RPC request		Remote Procedure Call.
Forwarded RPC†		Request which has been forwarded.
Asynchronous Message		Send only – no reply.
Entry		An entry is a subdivision of a task corresponding to a particular service.
Task		A task is an object which has a single thread of control and can initiate or accept service requests.
Task Pool		A set of tasks sharing a common input queue.
Device		A device consumes time (i.e. executes) on behalf of a request from a task.
†Forwarded remote procedure calls were not part of the original model.		

Table 3.1: Layered Queueing Network graph notation.

The underlying assumptions of the Stochastic Rendezvous Network Model are [152]:

- Messages arriving at a task are queued with a first-come, first-served discipline.
- The CPU demand at a task in response to a message is divided up into exponentially distributed slices between requests to lower level servers.
- Calls to lower level services are geometrically distributed with the specified mean, or occur in the (deterministic) number specified.
- The call graph with tasks as nodes and requests as arcs is acyclic (cyclic graphs may deadlock).

3.1.1 Model Components

Tasks in Stochastic Rendezvous Network Models are divided into three groups: pure clients, active servers and pure servers. *Pure clients*, also called *reference tasks*, only send messages.

These tasks cycle continuously, and can be used to model actual users and other input sources. *Pure servers* only receive requests and are analogous to stations in conventional queueing networks. Typically, they model hardware such as processors and disks. *Active servers* accept requests, then go on to make their own. Active servers are typically software processes. In Figure 2.1, *Group1* is a an example of a reference task, *A1* and *DP1* are examples of active servers, and *Disk1* is an example of a pure server.

Tasks communicate between one and another using the send-receive-reply [16] messaging paradigm, shown in Figure 3.1. The calling task, the *client*, is blocked during the interval between the send and the reply (the blocking time is labeled *rendezvous delay* in the figure). This style of inter-process communication is also known as a *rendezvous* [2] and models the remote procedure call [143, 5].

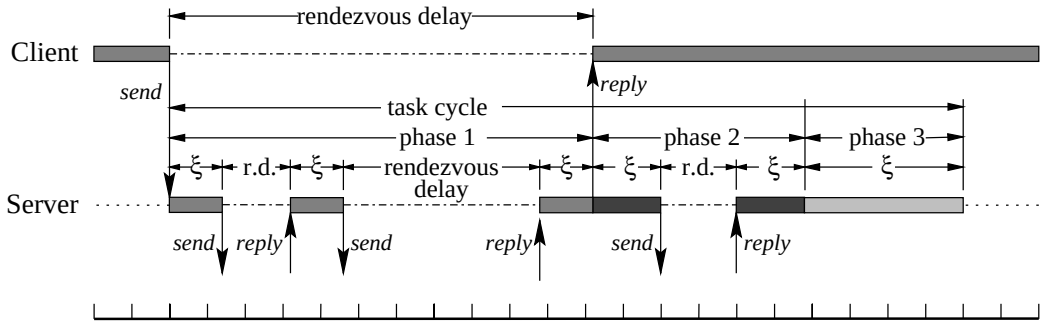


Figure 3.1: Task cycles and phases. Entry execution is divided up into a sequence of phases, which are themselves divided up into a number of slices, ξ.

Messages are serviced using a first-come first-served queueing discipline. The time a task spends processing a message, either awaiting responses from lower level servers, or executing on a processor, is broken up into *phases*. The time spent between the receive and reply operations in a server is a *service phase* and is called *phase one* (this phase is equivalent to the service time at a station in a product form queueing network). Subsequent phases are called *autonomous phases* because the client that initiated their execution is no longer

blocked. Simple remote procedure calls (e.g., Figure 1.1(a)) have no autonomous phases. Synchronous message passing systems which release the caller as soon as the message is received, have no service phase; this is implemented in hardware on Transputers [53]. Send-receive-reply interactions which can have autonomous second phases are supported by the programming language Ada [2] and with the Thoth [16], and Amoeba [88] operating systems (among others), and with remote procedure calls [143, 5].

A server may differentiate its actions based on the requests it receives which are offered through *entries*. Entries may correspond to actual communication ports on tasks, or they may correspond to message types that invoke different actions at a server. They can also be thought of as *classes* in a conventional queueing network.

Figure 3.2 shows a Stochastic Rendezvous Network with two reference tasks, one active server, and two pure servers. The two pure servers each have two entries whereas the other tasks in the model have one. The list of numbers within each of the entries represents the average service time for each phase. Variance may also be specified by phase, but is not shown here.

Arcs on the figure denote requests from one entry to another. The labels on the arcs denote the average number of requests made each time the corresponding phase in the source entry is executed. The number of requests is normally geometrically distributed. However, the model also permits a deterministic number of requests for a phase, which has proven useful when modelling pipelines [146].

3.1.2 Model Inputs and Outputs

Inputs to the model consist of mean service times, variance, phase types and call rates:

s_{ep} = the mean total execution time of entry e phase p.

c^2_{ep} = the squared coefficient of variation for a *slice* of execution time of entry e, phase p.

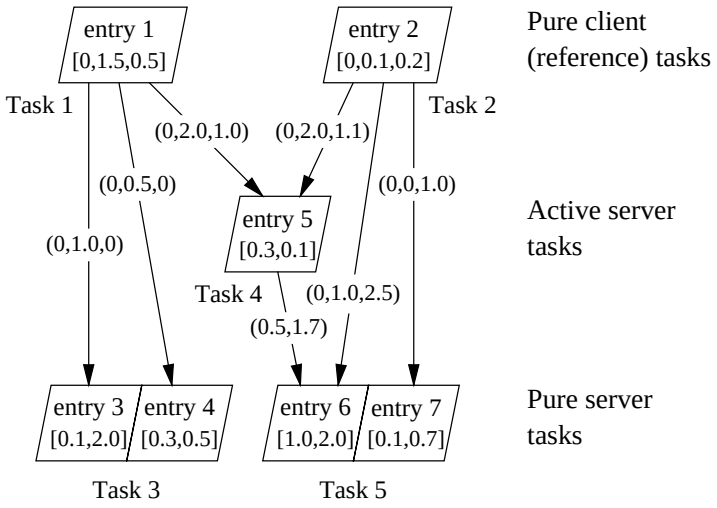


Figure 3.2: A Stochastic Rendezvous Network using the notation of Woodside et. al. [152]. Processors are not shown in this diagram.

pt_{ep} = the phase type (stochastic or deterministic). Stochastic phases emit a random number of requests to lower level servers with a geometric distribution with the request mean. Deterministic phases emit the exact number of (intergral) requests to lower level servers.

λ_{0e} = an *open* or external arrival stream of requests to entry e. Entry e must be part of a task designated as a server.

y_{edp} = the mean number of requests from phase p of entry e to entry d.

The throughput of each of the entries is the main result from the solution of a Stochastic Rendezvous Network:

λ_e = the throughput of entry e in messages per unit time.

3.2 Solver Capabilities

The first solvers of Stochastic Rendezvous Networks were SRVN [147, 149, 152] and MOL [112, 113, 110]. The solver described here, LQNS, was developed from ideas from both, starting in 1993. MOL has continued to be developed, but when it is referred to here, the version extant in 1993 and described in [113] is indicated.

The existing capabilities of the MOL, SRVN and LQNS solvers are summarized in Table 3.2 below [40]. Starting from the top of the table, the device scheduling parameter refers to the type of scheduling supported by hardware devices; task scheduling refers to the order in which messages are accepted by tasks. Open arrivals are an external Poisson input stream with a given rate to some entry of a non-reference task. The phase type of an entry specifies whether an exact number of requests are made (deterministic phases), or a random number of requests are made (stochastic phases). The squared coefficient of variation, c_v^2, $(\overline{s^2}/\sigma^2)$ sets the variance of the service time of an execution slice. By default, execution slices have exponential distributions (i.e. $c_v^2 = 1$). The fast coupling heuristic[1] applies a correction to servers with both frequent and infrequent arrivals at different entries. Forwarding[2] specifies rendezvous-type messages that are sent to multiple servers before the reply is sent to the original client. A "multiserver" refers to a fixed-size set of identical single-threaded tasks that share a single queue of requesters. For an infinite server there is no upper limit on the number of copies. Finally, the capacity parameter refers to the size of model that can be analyzed. A medium sized model may have upwards of twenty five tasks and processors; there is no fixed limit for large models.

[1] The fast coupling heuristic is not found in [40] – refer to [152].
[2] Forwarding is not found in [40] – refer to 3.3.1.

Parameter	MOL [152]	SRVN [113]	LQNS
Device Scheduling†	FPHS	FPH	FHPS
Task Scheduling†	F	F	FH
Open arrivals	no	yes	yes
Phase type‡	S	SD	SD
Vary c_v^2	yes	yes	yes
Fast Coupling Heuristic	no	yes	yes
Asynchronous sends	no	yes	yes
Forwarding	no	no	yes
Multiservers	yes	no	yes
Infinite-servers	yes	yes	yes
† F: FIFO, P: Preemptive Priority, H: Head-of-Line Priority, R: Random, S: Processor Sharing			
‡ S: Stochastic, D: Deterministic			

Table 3.2: Solver capabilities.

3.2.1 Extensions to the Original Model

The following improvements have been made and incorporated into the Layered Queueing Network solver. Changes denoted by section references '§' are by the author. Changes denoted by references '[]' are by others.

- The layering strategy from the Method of Layers has been modified to unify the processor/process model along the lines of Stochastic Rendezvous Networks (§3.3.2).
- The solver is object-oriented to allow changes to be made easily (§3.4).
- Replication of tasks can be performed to allow rapid solution of large models with varying degrees of symmetry (See [94]).
- Forwarded remote procedure calls are added to the model (§3.3.1).
- Think times by reference task and entry are added to the model (§3.3.1).
- The "interlocking" calculation from the SRVN solver and MOL solvers is generalized (§4).

- The "overtaking" calculation used by the SRVN and MOL solvers is replaced with a new, more accurate algorithm (§5).
- Multiservers can have two phases of service (§5).
- Tasks may have heterogenous threads of control (§7).

3.3 The Layered Queueing Network Solver

The Layered Queueing Network Solver (LQNS) is a new solver combining features from the Method of Layers (MOL) [112, 113, 110] and the Stochastic Rendezvous Network (SRVN) [147, 149, 152] solvers. Layered queueing networks are solved using surrogate delays to solve the simultaneous resource possession problem arising from the nested calling pattern in the system being modelled. This goal is accomplished by partitioning the input layered queueing network model into a set of smaller *"MVA submodels"*, then iterating among these submodels until convergence in waiting times.

The steps to solving a client-server queueing network model using LQNS are as follow:

1. Read the input file and construct an object data base consisting of the tasks, processors and entries and the calls between them.
2. Perform processor, forwarding and think-time transformations.
3. Generate the layer submodels, then construct MVA submodels from the layer submodels.
4. Solve the MVA submodels. The inputs to and the outputs from each submodel are extracted from or saved to the object database. This step is repeated until the waiting time results converge for each layer.
5. Write the results out.

Figure 3.3 shows the steps above algorithmically; the steps are described in greater detail below.

```
1:   Read input and create LQNS model.
2:   Transform model.
3:   Generate layers and submodels.
4a:  Find Type one throughput bounds to give an initial solution.
     repeat
        for l → 1 to submodel L do
4b:        Set MVA parameters for submodel l
4c:        Solve submodel l
4d:        Set waiting times for submodel l and think times for submodel l + 1
        end for
     until convergence
5:   Write results.
```

Figure 3.3: Solution algorithm.

3.3.1 Model Transformation

Processor Transformation

The original input model consists of entries assigned to tasks which are, in turn, assigned to processors. In the input model, service time demands are assigned to phases of entries. However, the execution of a program can only take place on processors. The processor transformation step converts the input model so that service demand is assigned to processors only.

The processor transformation step proceeds by replacing the service time s_{ip} of the original phase p of entry i by a number $(\sum_j y_{ijp} + 1)$ of requests for *slices* of execution at the task's processor where y_{ijp} represents the requests made from entry i to some other

entry j on a lower-level serving task. The service time for the slice on the processor is:

$$\xi_{ip} = \frac{s_{ip}}{\sum_j y_{ijp} + 1} \tag{3.1}$$

When a submodel with a processor is solved, the requests for slices of execution will include processor contention. Figure 3.4 shows the processor transformation applied to Figure 3.2.

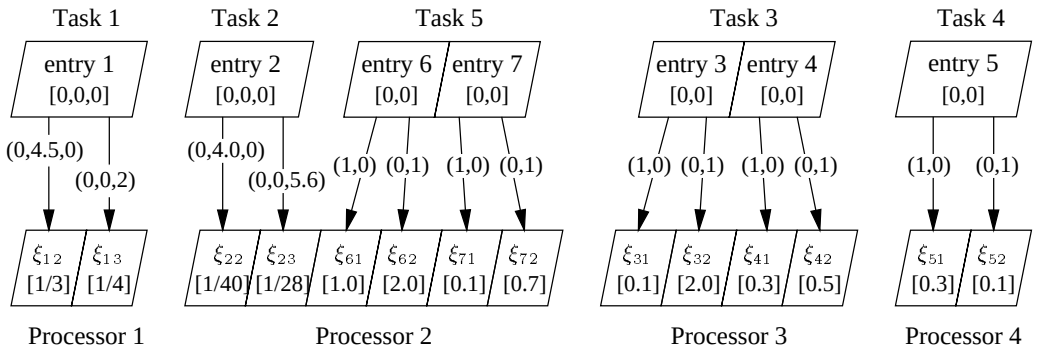


Figure 3.4: Processor Transformation applied to Figure 3.2.

Entry Think Times

Think times can be specified for both a task as a whole, and for individual entries. The think time for a task is handled by the underlying MVA solver as the think time for a chain. Think times for individual entries are handled by first creating a delay center, then making requests to the delay center.

Forwarding

Forwarding occurs when a server passes a message to yet another server rather than replying to the originating client, shown in Figure 3.5, and is a new modelling capability. This interaction can be used to improve performance for small RPC requests by removing the need for a client to send a request to one server, wait for a reply, then send a subsequent

request to a second server. This form of interaction was found in one of the earliest operating systems using the send-receive-reply message system, namely Thoth's `.forward()` function [16]. A more recent example is the `callit()` function in the portmapper [26] in SunOS.

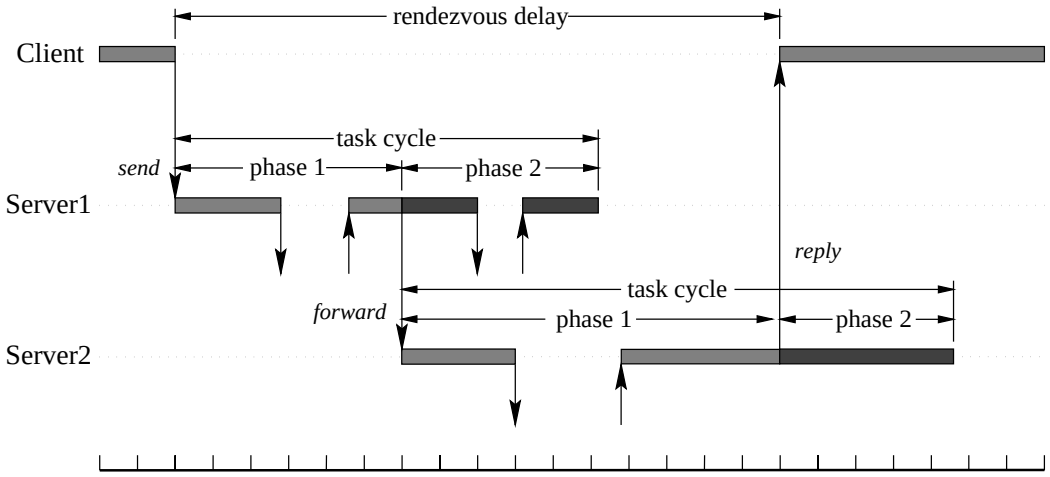
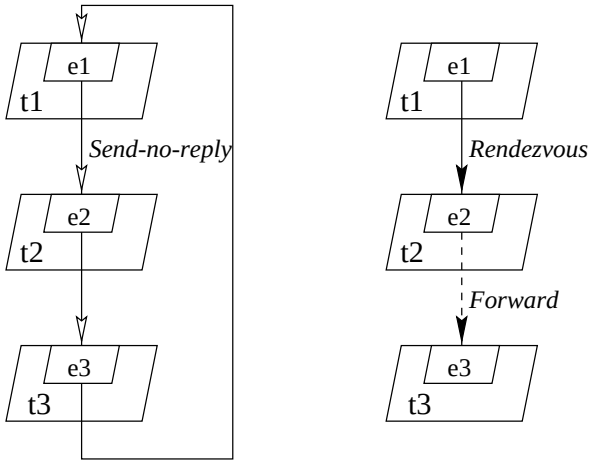


Figure 3.5: Forwarding.

Forwarding can be used to model a system with asynchronous messaging, provided that there is a cycle in the call graph. This interaction is shown in Figure 3.6(a) and occurs when a task sends an asynchronous message then blocks awaiting a response from some server stimulated by the original request. The converted model, shown in Figure 3.6(b) is created by locating a task within the cycle of asynchronous messages that first sends a message, then blocks waiting for a reply. The outgoing request is replaced with a rendezvous. All remaining asynchronous requests in the cycle from the remaining tasks are replaced with forwarding requests.

Forwarding can be used to convert an open model, shown in Figure 3.7(a) to a closed model, shown in Figure 3.7(b). The open arrival source is replaced by a set of client tasks that make rendezvous requests. The remaining send-no-reply requests are then converted



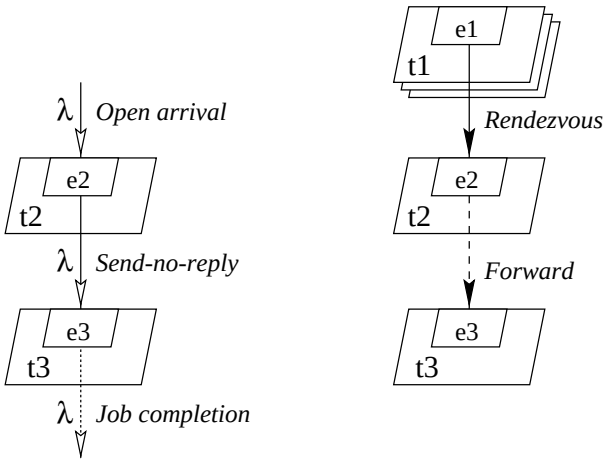
(a) Send-no-reply Loop (b) After Transformation

Figure 3.6: Asynchronous to Synchronous messaging model conversion with forwarding.

to forwarding calls. Shousha et. al. [129] have applied this technique with good results to a telecommunications system.

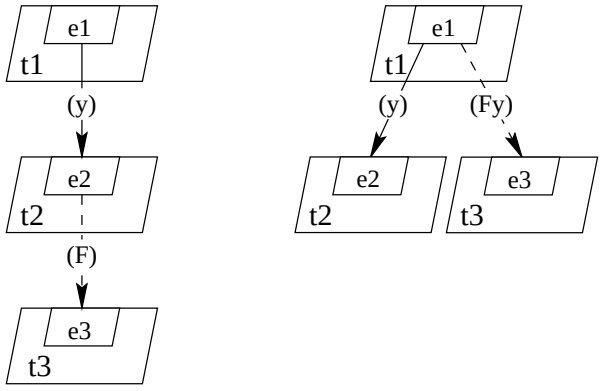
Forwarding in the LQNS solver is handled by transforming the input model. The transformation reconnects the forwarding requests to the client making the original rendezvous request (in essence the client makes all of the requests in the transformed model). Figure 3.8 shows a three-level system with forwarding before and after transformation. Note that one level of servers is removed.

It is important to note that the transformed model is **not** the same as one where the client makes two remote procedure calls directly; the transformed requests are not included when finding slice times, nor are they used to find the overtaking and interlocking probabilities.



(a) Open System (b) After Transformation

Figure 3.7: Open to closed model conversion with forwarding.



(a) Original Specification (b) After Transformation

Figure 3.8: Model transformation for forwarding.

3.3.2 MVA Submodel Generation

Layered queueing networks are solved by hierarchical decomposition and the method of surrogates [58]. The input layered queueing network model is broken down into a set of submodels $\mathcal{L}$ each consisting of a set of clients $\mathcal{C}_l$ and a set of servers $\mathcal{S}_l$. Processors and other pure-servers will only appear as servers in submodels, reference tasks and other pure-clients will only appear as clients, while active servers will apear in some submodels as clients and in other submodels as servers. The submodels are then solved using approximate MVA.

LQNS incorporates four different layering strategies for partioning the input model into submodels. These strategies will be demonstrated using Figure 3.9, a version of Figure 3.2 showing processors but omitting entries. The topological sorter and the layering strategies are described next.

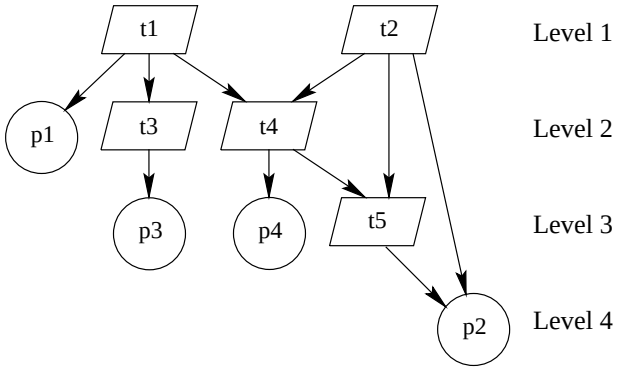


Figure 3.9: The tasks of Figure 3.2 sorted by nesting level. Entries are not shown.

Topological Sort

Before model solution can begin, the tasks and processors in the input model must be sorted to set their respective *nesting levels*. The nesting level for reference tasks is defined as 1. The nesting level for tasks which accept open arrivals is defined as 2 (the open arrival requests implicitly arrive from level 1). The nesting level for any other task or processor in

the input model is defined as the longest call path from the reference task or open arrival source. Figure 3.9 shows the nesting level for the model in Figure 3.2.

Strict Layering

Strict Layering is the primary approach used by the Method of Layers [113]. Tasks are sorted into levels by the topological sorter and form the *software submodel*. Processors are all assigned to their own level called the *hardware submodel*.

In the Method of Layers, tasks from level l make up the clients for submodel l, and tasks from level $l+1$ make up the servers. Difficulties arise from this approach when the call graph is not strictly layered, as is the case with Figure 3.9 (the task *t2* is not an immediate parent of task *t5*). This problem is overcome by adding psuedo-tasks between the requesting and accepting tasks prior to layer construction.

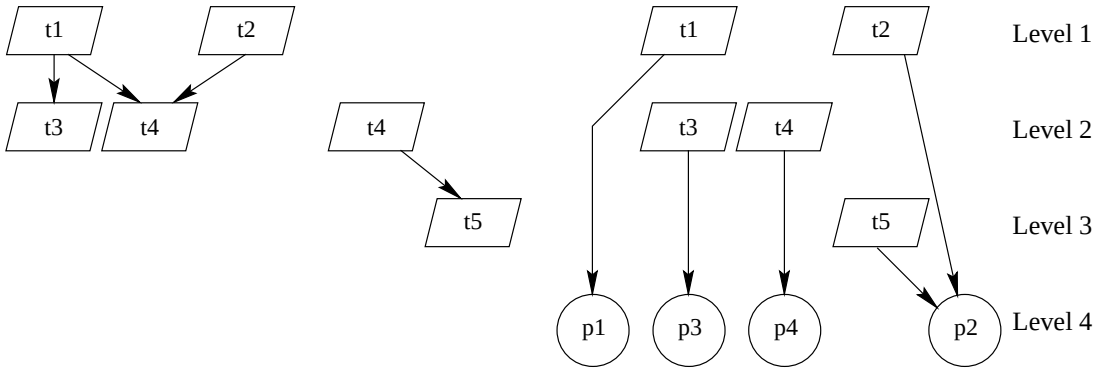
The Layered Queueing Network Solver takes a different approach: tasks from level $l+1$ still make up the servers for submodel l, but clients can come from any other level. This approach has the advantage of not adding psuedo-tasks to fill the intervening layers, but at the cost of extra storage space to save waiting times for each layer.

The hardware submodel consists of all the devices in the input model acting as servers and all of the tasks acting as clients.

Figure 3.10 shows the submodels resulting from this technique.

Loose Layering

Loose Layering is the approach used by Stochastic Rendezvous Networks and also implemented in the Method of Layers. Layers consist of exactly one serving task or processor acting as a server plus all of the tasks that call the task or processor acting as clients. Figure 3.11 shows the submodels that arise from Figure 3.9 using this technique.



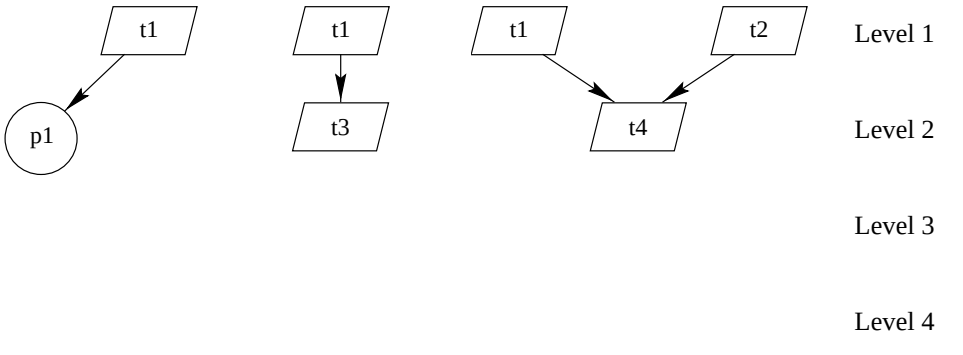
(a) Submodel 1 (b) Submodel 2 (c) Submodel 3

Figure 3.10: Submodels for the model shown in Figure 3.9 using strict layer partitioning.

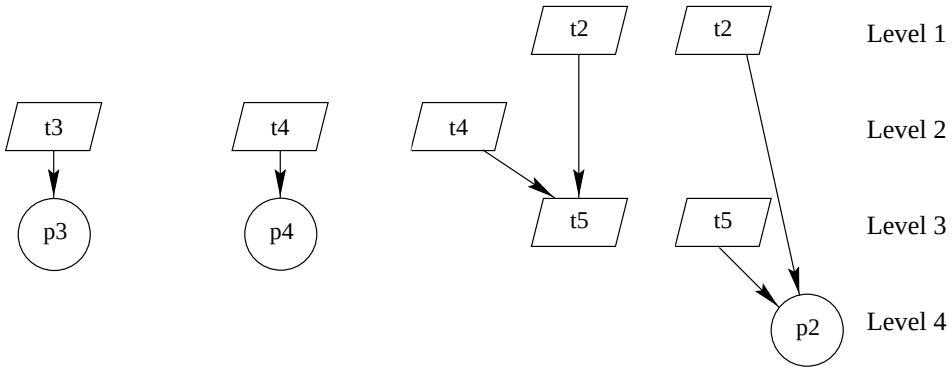
Batched Layering

Batched Layering is the primary layering strategy used by the new Layered Queueing Network Solver. As in loose layering, processors and tasks are treated uniformly; i.e., processors are included as servers at the appropriate level along with software tasks. Submodel l is constructed by finding all clients, regardless of level, that make calls to servers in nesting level $l + 1$.

Figure 3.12 shows the submodels for the system in Figure 3.9. Reference tasks in the original input model (for example, *t2*) never appear as servers. Similarly, pure servers (for example, the processor *p2*) never appear as clients. Active servers appear as both clients and servers, but in different submodels (for example, task *t5*). Note that task *t2* appears in all three sub-models.

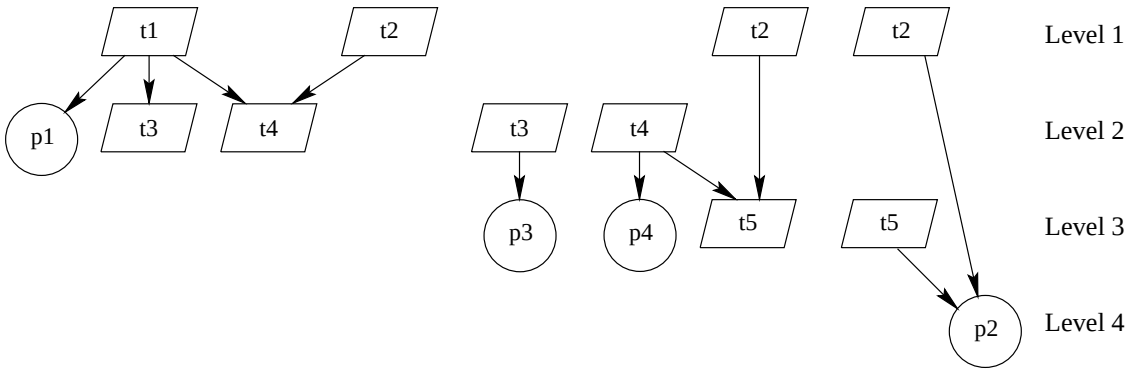


(a) Submodel 1 (b) Submodel 2 (c) Submodel 3



(d) Submodel 4 (e) Submodel 5 (f) Submodel 6 (g) Submodel 7

Figure 3.11: Submodels for the model shown in Figure 3.9 using loose layer partitioning.



(a) Submodel 1 (b) Submodel 2 (c) Submodel 3

Figure 3.12: Submodels for the model shown in Figure 3.9 using batch partitioning.

Squashed Layering

With *Squashed Layering*, only one submodel is created. Reference tasks appear as clients, pure servers (processors) appear as servers and active servers appear as both clients and servers. Figure 3.13 shows the submodel generated from Figure 3.9. Note that *active servers* such as *t3*, *t4* and *t5* appear twice, once as a client and once as a server in the submodel.

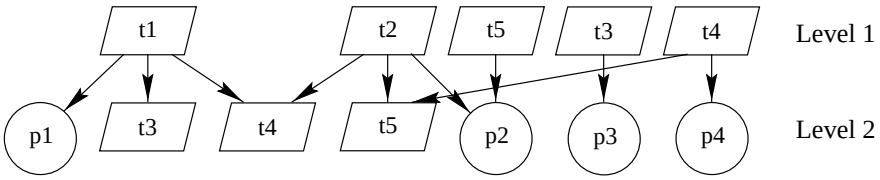


Figure 3.13: Submodels for the model shown in Figure 3.9 using squashed partitioning.

Comparison of Layering Strategies

The overall solution performance of the LQNS solver depends on four factors:

1. the number of submodels, N,
2. the number of clients, C_n, in submodel n,
3. the number of servers, S_n, in submodel n,
4. the rate of convergence, $\tilde{I}_n$ for submodel n, and
5. the rate of convergence, $\tilde{\Delta}$ for the overall model.

The solution cost is:

$$\text{Cost} = O(\tilde{\Delta}(N-1)\sum_{n=1}^{N} S_n C_n^3 \tilde{I}_n). \tag{3.2}$$

The terms of the summation are from Linearizer [12]. $\tilde{\Delta}$ is a random quantity reflecting the rate of convergence of the outer iteration of Figure 3.3. The N submodels must be solved at least $N-1$ times because queueing delays from the lowest level submodel only propagate upwards one layer each time the outer loop executes.

Typically, the solution time is dominated by the layer with the largest number of clients because of the C^3 term in (3.2). Solution time can be improved if the Schweitzer approximate MVA [125] is used instead, but with a loss of accuracy.

The best layering strategy minimizes (3.2). In all cases, the *loose* layering strategy produces the most submodels and the least number of servers $S = 1$ in each submodel. Conversely, the *squashed* layering strategy produces the least number of submodels, $N = 1$, but has the most clients and servers in the one submodel. The *strict* and *batched* layering strategies will always produce the same number of submodels equal to the maximum call depth. However, the terms S, C, $\tilde{I}$ and $\tilde{\Delta}$ will differ. Finally, since $\tilde{I}$ and $\tilde{\Delta}$ depend in a complex way on the submodels, and impossible to know *a-priori*, optimal strategy selection is difficult.

Table 3.3 compares the four strategies from the 18 test cases from [149] and the 21 test cases from [152][3]. The column labeled N is the number of submodels. The next three columns show the average number of times key functions are executed over all of the test cases. The column `solve` is the mean number of times a submodel is solved using Linearizer, the column `step` is the average number of times the core MVA step function is executed, and the column `wait` is the average number of times any waiting time is calculated[4]. The number of time the `wait` function is called determines the overall run time. The final column, labeled $\epsilon(\lambda)$, shows the average relative error in throughput (in percent) when the results are compared to exact values computed using GreatSPN [17, 18].

strategy	N	`solve`	`step`	`wait`	$\epsilon(\lambda)\%$
strict	3	16.9	188.889	56365.2	3.07
loose	6	63	611.556	222211	3.07
batched	3	20.8	297.167	168058	3.06
squashed	1	6.6	201.167	455684	3.07

(a) Test cases from [149]. Convergence was set at 10^{-5}.

strategy	N	`solve`	`step`	`wait`	$\epsilon(\lambda)\%$
strict	3	83.4	989.286	364402	2.72
loose	7	74.2	762.15	92293.5	2.72
batched	3	37.4	579.952	254355	2.72
squashed	1	15.3	621.667	1.42858×10^{6}	2.86

(b) Test cases from [152]. Convergence was set at 10^{-6}.

Table 3.3: Run-time cost comparison of layering strategies. Each of the test cases produced results comparable to the others in accuracy.

From Table 3.3, there is no simple conclusion about the best approach. Strict layering works best for the 18 tests cases from [149], while loose layering works best for the 21 test

[3]The example used here and shown in Figure 3.2 is the first test case from [152].
[4]These functions are described in Section 3.4.3.

cases from [152]. Squashed layering is clearly inferior due to the C^3 term in (3.2), though this would change if the simpler Schweitzer approximate MVA were used instead. Batched layering appears to be a reasonable compromise.

Convergence

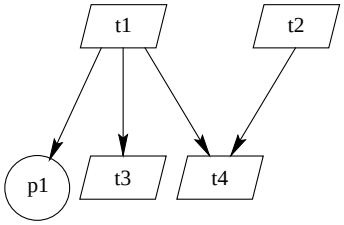
The $\tilde{\Delta}$ term in (3.2) expresses the inflation factor in the run-time cost due from the rate of convergence in the outer iteration of the main algorithm. Experience has shown that convergence problems arise when two or more servers in a particular submodel are saturated. In some instances, traffic is shifted from one server to the other during execution, causing oscillations. Under-relaxing the waiting time result from each submodel usually corrects the problem, but at a cost of more iterations of the outer loop.

3.3.3 MVA Submodel Construction

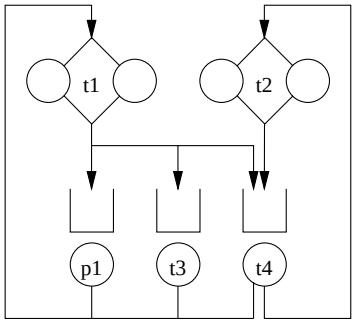
Once layer submodels are created from the original layered queueing network using one of the four strategies described in the preceding section, MVA submodels are created to find throughputs and response times. Clients in a given submodel are modelled as delay stations and also form the routing chains in the corresponding MVA model (the populations of the routing chains are described in the next section). The servers are modelled as queueing stations where the queueing discipline is derived from the properties of the associated tasks. Server types are described in Section 3.4.6 below. Figure 3.14 shows the MVA submodels that arise from the example system using batched layering.

Initial Service Times

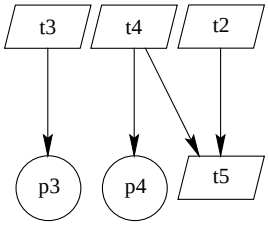
Initial service times for all of the MVA submodels are set by finding the minimum service time for each entry. The minimum service time value for an entry is found by adding the entry's service time demand to the service time of all entries that it calls. Since entries



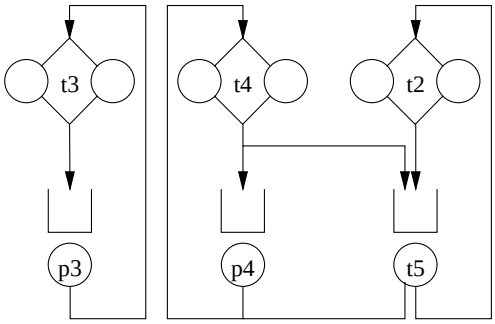
(a) Submodel 1



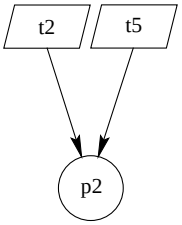
(b) MVA Submodel 1



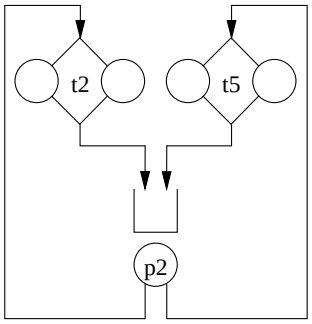
(c) Submodel 2



(d) MVA Submodel 2



(e) Submodel 3



(f) MVA Submodel 3

Figure 3.14: MVA submodels arising from the submodels shown in Figure 3.12.

at higher levels depend on the service times of entries at lower levels, the bounds must be found starting from the lowest level in the model.

Type one throughput bounds are defined as the throughputs of the entries when there are no contention delays; i.e., they define the "guaranteed not to exceed" values of throughput. These bounds are simply the reciprocal of the minimum service times and are used to set the initial request rates for send-no-reply requests.

Customer Derivation

Customers for routing chains are based on the number of potentially active instances of a task when it appears as a client in a submodel. For simple single threaded first-come first-served fixed-rate (FCFS-FR) tasks, the number of active instances is one. Multi- and Infinite-servers pose another problem. Representing these tasks as routing chains with only one customer would not produce the right customer mix in the queueing network and would consequently produce low estimates for contention delays. Representing infinite servers with routing chains with an infinite number of customers would produce an infeasible queueing network. Similarly, a multiserver may not be be driven to the point where all its instances are active at once; too many customers in the routing chain will overestimate the amount of contention.

The customer derivation algorithm estimates the number of customers in each routing chain based on the client's corresponding task type. For FCFS-FR clients, the number of customers in the corresponding routing chain is set to one. For multiserver clients, the number of customers is set to the minimum of either the number of servers at the multiserver task, or to the number of clients that can call the multiserver. For infinite servers, the number of customers is the number of clients that call the task[5].

[5]Reference tasks must always have a fixed number of instances. To model an unlimited source of requests, use "open arrivals" instead.

3.3.4 MVA Submodel Solution

The input parameters of visits to stations and customer populations of chains to each of the MVA submodels are set by the overall input model and do not vary as it is solved. Think times for pure clients (reference tasks) and service times for pure servers (generally processors) are also set by the overall input model. The remaining parameters, think times for chains in all but submodel 1 and the service times for all pure clients and active servers, are based on previous solutions to the overall model, or from the type one throughput bounds for the first iteration.

Active servers such as *t3* in Figure 3.9 take on roles as both clients and servers in MVA submodels. When a task m is acting as a client in an MVA submodel l, it is always modelled as infinite server. Its service time demand is found by taking the sum of the waiting times w_{mj} to all of its servers in all other submodels except for those in the current submodel l:

$$s_{m_l} = \sum_{i \in \mathcal{L}, i \neq l} \sum_{j \in \mathcal{S}_i} w_{mj} \tag{3.3}$$

For example, Figure 3.15 shows task *t3* acting as a client to processor *p3* in MVA Submodel 2. Since *t3* does not call any servers outside of this MVA submodel, its service time demand is set to zero.

Clients in an MVA submodel also correspond directly to the chains in the submodel. The think time, $Z_{i,l}$, for chain i in an MVA submodel l is found from the utilization, $\rho_{i,l-1}$, of task i, when it is acting as a server in submodel $l-1$:

$$Z_{i,l} = N_{i,l} \frac{1 - \rho_{i,l-1}}{\lambda_{i,l-1}} \tag{3.4}$$

From Figure 3.15, the think time for the chain corresponding to task *t3* in MVA Submodel 2 is found from the utilization of the server *t3* in MVA Submodel 1[6]. As reference tasks

[6]Since clients are infinite servers, the think time could be represented as service time demand at the client

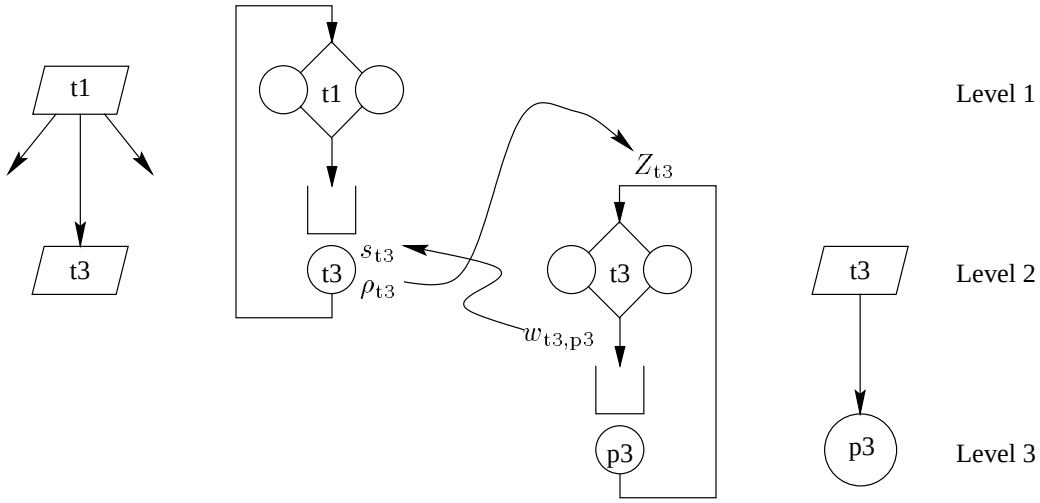


Figure 3.15: Parameter flow from one MVA submodel to another. Think times for submodel $l+1$ are based on the utilizations found from solving submodel l. Service times for servers in submodel l are based on the waiting times found in all submodels.

never appear as servers, the think time for the corresponding chain is always set to the value defined initially by the user in the model input file (by default, zero).

When a task m is acting as a server in an MVA submodel l, its service time is found by taking the sum of the waiting times to all of its servers regardless of submodel:

$$s_{m_l} = \sum_{i \in \mathcal{L}} \sum_{j \in \mathcal{S}_i} w_{mj} \tag{3.5}$$

Returning to Figure 3.15; in MVA Submodel 1, task *t3* is acting as one of the servers to task *t1*. The service time demand for *t3* in this MVA submodel is the waiting time found in in MVA submodel 2 when *t3* calls *p3*.

station instead of a think time for a chain.

3.3.5 Overall Solution

The submodels are solved starting from submodel 1 and finishing with submodel L, except when strict layering is used (this algorithm was shown earlier in Figure 3.3). For strict layering, submodels 1 through $L-1$ (the software submodel) are solved using this approach first. Next, submodel L (the hardware submodel) is solved and the process repeats.

The outer iteration of the solution algorithm stops when the service time difference for every submodel in the solution is less than the convergence value specified by the user. The service time difference for a submodel is the root mean square difference of the service time of all of the phases in the submodel from one iteration to the next.

Back Propagation Improvement

Solution of layered models starts typically at the top most layer and works down. However, as shown in Figure 3.15, submodel l is dependent on the results from submodel $l+1$ and vice versa. Consequently, contention delays are only propagated up one level at a time for each iteration of the outer loop shown in Figure 3.3. Reversing the order of solution will not improve matters. The top-down strategy propagates "think times" down during each iteration of the inner loop, but at the cost of propagating waiting times up one level per iteration of the outer loop. Reversing the order would propagate waiting times up more quickly, but then think times would propagate down slowly.

There two ways to ameliorate this situation. The first is to simply re-run the type-one throughput bounds calculation to re-estimate the waiting times, thus propagating the contention delay from the lowest level upwards. However, care must be exercised so that waiting times at intermediate levels do not oscillate from being set using two different techniques. The second way (the one used by the LQNS solver) is to cycle from the top to the bottom, then back up again, much like an elevator in a building. This technique has the advantage that the waiting times for level $l-1$ will be based on actual contention at

level l, rather than simply being estimated.

3.4 Solver Design

The LQNS solver is written in the object oriented language C++ [134] to speed development, increase the quality of code, to reduce maintenance costs and to allow changes to be made easily [48, 6]. Furthermore, since classes can be substituted easily, algorithm comparisons can be performed without having to retain multiple differing versions of the software. The trade off for all this flexibility is speed – object oriented programs can take more time to do things due to dynamic lookup and dynamic memory management. This section gives a brief overview of the software architecture of LQNS. The software itself is described in [38].

3.4.1 Principles of Design

The solver consists of two principle parts: an object representation of the input model and an MVA solver to solve the submodels that arise from the input model, shown in Figure 3.16. The rational for the classes for these two parts are described next. The implementation of the classes are described in sections that follow.

Input Model Objects

The input model (a graph) consists of nodes and arcs. The nodes in the model are the processors, tasks, entries and phases; these are all represented by their own classes. The arcs in the model are the calls made from phases to entries.

Task and processor objects share many properties from the standpoint of the solver. The shared components are defined by the class *entity*. Entities contain objects of class *Entry* and Entries contain objects of class *Phase*.

Entity objects are used to define the *stations* that appear in the MVA submodels. Entries represent *class changes* for customer chains in the corresponding MVA submodels.

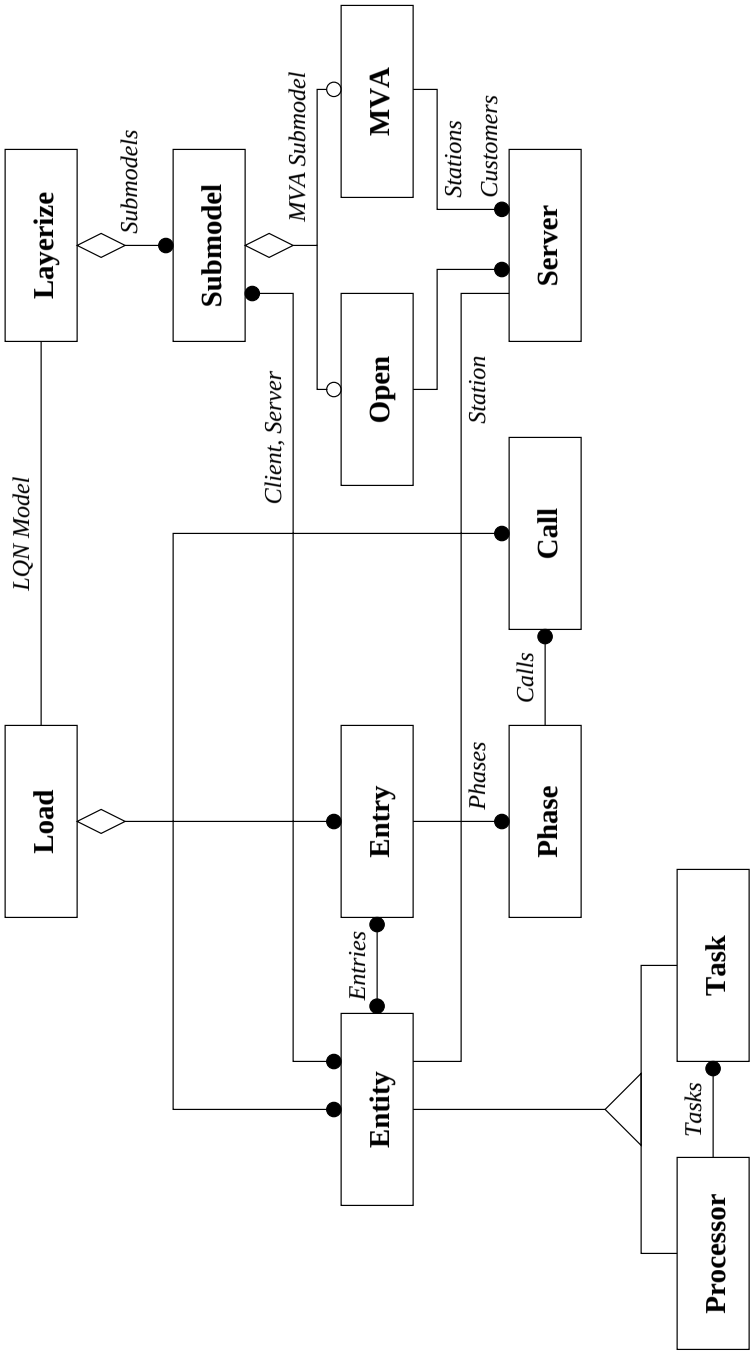


Figure 3.16: Overall class hierarchy of Layered Queueing Network solver. The notation is from Rumbaugh [118].

MVA Solver Objects

Exact and approximate MVA both use the same core algorithms to find the waiting times at stations. The main difference between the two is that exact MVA recursively solves the queueing network equations for all populations, whereas approximate MVA solves the equations at one population level. Linearizer and Schweitzer approximate MVA are also very similar (Linearizer adds a term to (2.2) to better approximate the change to the queue caused by a new customer). These observation leads to the class hierarchy shown in Figure 3.17.

In conventional programming languages, switch statements are used to choose among a collection of different alternatives. In the context of this solver, the alternatives for the switch statement are the different ways to compute the waiting time at a station. This observation leads to the class hierarchy shown in Figure 3.23, where each class implements its own waiting time expression.

3.4.2 Solver Class Organization

Figure 3.16 shows the overall class hierarchy for the LQNS solver. Execution begins at the class *Load*. This class calls the input file parser, which in turn creates instances of the classes *Call*, *Entry* and *Entity*. Call objects represent arcs in the input model and are used to connect Entry objects together. They store the visit counts to entries and the per-call waiting time results. Entry objects contain *Phase* objects, which in turn store input parameters such as service demand, and output parameters such as the elapsed time per invocation of a phase. The *Entity* class is subclassed into *Tasks* and *Processors* which are, in turn, subclassed into classes such as *ReferenceTask* and *MultiProcessor*. Collections of entries, representing the entry lists of tasks and processors are stored in Entity objects.

The class *Layerize* is the heart of the solver. It constructs the layer submodels as instances of class *Submodel*, then iterates among the submodels until the differences in the

service times for entries between iterations approaches a small value.

Class *Submodel* creates the relationships between the entities such as tasks and servers in the MVA submodels. An entity may appear in two or more submodels, depending on what it calls. For example, in Figure 3.14, task *t2* appears as a delay station in all three MVA submodels; task *t4* appears as a delay station in MVA submodel 2, and as a queueing station in MVA submodel 1.

The sections that follow describe the software design of the MVA solver and its stations in greater detail.

3.4.3 Closed Model Mean Value Analysis

The Layered Queueing Network Solver implements five different versions of Mean Value Analysis, shown in the Class Hierarchy in Figure 3.17.

The principle results returned by the MVA class are:

1. a vector of throughputs λ, dimensioned by the number of chains K,
2. An array of per-call waiting times, $\mathbf{W}$ dimensioned by the number of stations M and the number of chains K.

Inputs to an MVA object consists of:

1. a vector of customers $\mathbf{N}$, dimensioned by the number of chains K.
2. a vector of think times $\mathbf{Z}$, dimensioned by the number of chains K.
3. a vector of stations $\mathbf{Q}$, dimensioned by the number of stations M. A station has mean service times, s, visits, v, and a waiting time calculation.

The MVA solver also takes as optional input values:

1. *Overlap probabilities* dimensioned by the number of chains $K \times K$ (see Chapter 8). If fork-joins are not present in the model, the overlap probabilities are all set to one.

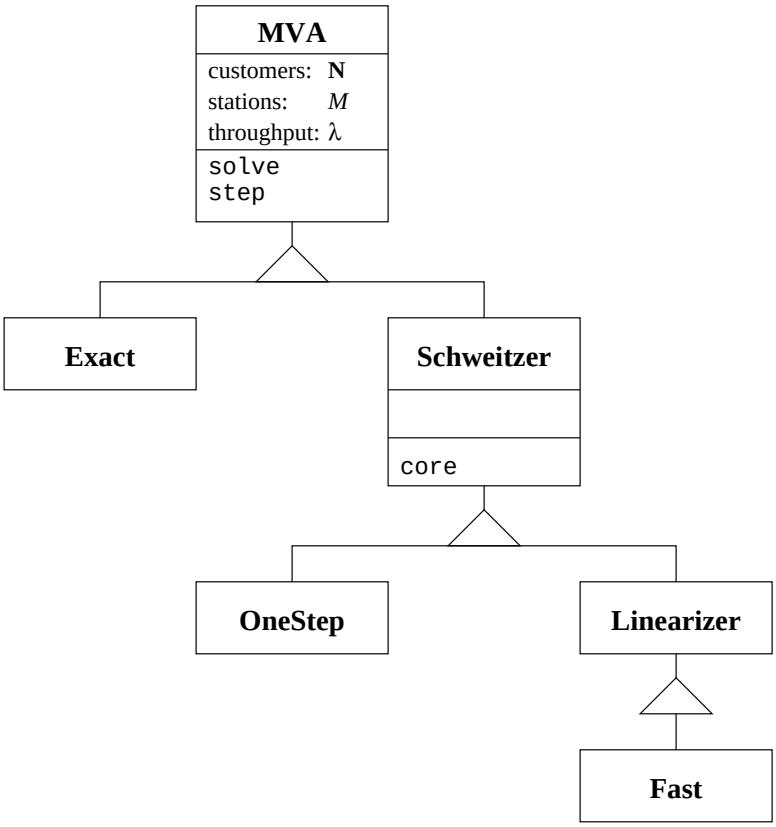


Figure 3.17: Class hierarchy for MVA objects.

2. *Priorties* dimensioned by the number of chains K (see §3.4.5).

To represent the class changes, the input variables s and v (for service time and visits respectively) for a station m are dimensioned both by the number of chains K in the submodel, and by the number of entries for the task or processor, E_m. The outputs of the MVA calculation, waiting time W, queue length L, and utilization U are also dimensioned by both K and E_m.

> Note: Refer to the glossary on Page xxi for definitions of the variables used by the MVA solver.

Class MVA

Class MVA is the abstract superclass and implements *one-step MVA* through the function `step()` shown in Figure 3.18. One-step MVA finds the waiting times and throughputs for $\mathbf{n}$ customers given the solutions at $(\mathbf{n}-\mathbf{e}_k)\forall k \in K$. Waiting times, W, at queueing stations $Q(m)$ are found by calling the function `wait()` [7]. Once the waiting times are found, the queue lengths, L, throughputs, λ, and utilizations U are computed.

Class Exact MVA

Exact MVA [106, 104, 107] solves recursively the queueing network for all customer populations $0 < \mathbf{n} \leq \mathbf{N}$. Provided that the MVA Submodel meets the constraints of a product form queueing network [57, 4], the solution is exact.

Since the solution of the overall queueing network depends on the solutions of the network at reduced population levels, the storage and computation time costs are

$$\$ = O(M \prod_{k=1}^{K} N_k)$$

[7] `wait()` is implemented in class *Server*, described in Section 3.4.6.

{ One-step MVA }

proc *step*(**n**)
  for $m \leftarrow 1$ **to** M **do**
    for $k \leftarrow 1$ **to** K **do**
      for $e \leftarrow 1$ **to** E_m **do**
        $R_{mek} \leftarrow v_{mek} \times wait(m, e, k, \mathbf{n})$
  for $k \leftarrow 1$ **to** K **do**
    begin
    $$\lambda_k \leftarrow \frac{n_k}{Z_k + \sum_{m=1}^{M} \sum_{e=1}^{E_m} R_{mek}}$$
    for $m \leftarrow 1$ **to** M **do**
      $L_{mek}(\mathbf{n}) \leftarrow \lambda_k R_{mek}$
    end

  { Compute marginal probabilities for load-dependent servers only }

  marginalProbabilities(m)
end *step*

Figure 3.18: One-Step Mean Value Analysis for multiple routing chains with class changes. The function *wait()* is implemented in class *Server* (See §3.4.6).

For large networks with multiple customers in each routing chain k, Exact MVA is prohibitively expensive.

```
proc solve( N )
begin
   for m ← 1 to M do
      L_m(0) = 0
   for n ← 1 to N do
      step(n)
end solve
```

Figure 3.19: Exact Mean Value Analysis for multiple routing chains.

Class Schweitzer

For large queueing networks, Exact MVA is prohibitively expensive because of the recursive solution. Schweitzer Approximate MVA [125] breaks the recursion by solving the network at $\mathbf{N}$ and estimating the queue lengths at with one customer removed from each chain j by assuming that $L_{mk}(\mathbf{N} - e_j)$, is proportional to $L_{mk}(\mathbf{N})$, i.e:

$$L_{mk}(\mathbf{N} - e_j) = \begin{cases} L_{mk}(\mathbf{N}) & \text{for } k \neq j \\ \frac{N_j - 1}{N_j} L_{mk}(\mathbf{N}) & \text{for } k = j \end{cases}$$

Figure 3.20 shows the Schweitzer approximate MVA algorithm as incorporated by Linearizer. For Schweitzer approximate MVA, $D_{mkj}(\mathbf{N})) = 0$.

Class Linearizer

Schweitzer approximate MVA often has large errors because the queue length estimate $L_{mk}(\mathbf{N} - e_j)$ is not a simple ratio of the customers in chain j. Linearizer [12] improves the accuracy of the Schweitzer approximation by calculating a scaling factor D_{mkj} to be used to find the populations of the queues with one customer removed. Linearizer finds the scaling

proc *core*(**N**)
begin
  repeat

    { Estimate L_{mk} }

    for $m \leftarrow 1$ **to** M **do**
      for $e \leftarrow 1$ **to** E_m **do**
        for $k \leftarrow 1$ **to** K **do**
          begin
          $F_{mek}(\mathbf{N}) = L_{mek}(\mathbf{N})/N_k$
          for $j \leftarrow 1$ **to** K **do**
            $L_{mek}(\mathbf{N} - e_j) \leftarrow (\mathbf{N} - \mathbf{e}_j)_k \cdot (F_{mek}(\mathbf{N}) + D_{mekj}(\mathbf{N}))$
          end

    { One-step MVA }

    step(**N**)

    { Termination test }

  while $\max_{m \in M, k \in K} \dfrac{|L^I_{mk}(\mathbf{N}) - L^{I-1}_{mk}(\mathbf{N})|}{N_k} > \dfrac{1}{4000 + 16|\mathbf{N}|}$
end *core*

Figure 3.20: Schweitzer MVA approximation for multiple routing chains.

factors by solving the queueing network using the Schweitzer approximation at both the full customer population $\mathbf{N}$, and $\mathbf{N} - e_j$ for all j. Figure 3.21 shows the linearizer algorthim.

Class FastLinearizer

Class *FastLinearizer* implements the improvements to Linearizer described in [28]. This change reduces the computational complexity of Linearizer from $O(MK^3)$ to $O(MK^2)$. The Aggregate Queue Length algorithm [155], which is based on Linearizer but with a lower computational cost still, is not implemented.

Class OneStepMVA

Class *OneStepMVA* exports the `step()` function directly, so that the outer iteration of the main algorithm shown in Figure 3.3 can perform the convergence test directly. Running the Layered Queueing Network Solver in this fashion most closely approximates the solution technique of the Stochastic Rendezvous Network solver.

3.4.4 Open Model Mean Value Analysis

Open models are those which have an infinite stream of arriving customers. These customers traverse the queueing network then depart. Open models arise in layered queueing networks for either open arrival sources (see Figure 3.7(a) on page 46), or from asynchronous send-no-reply interactions (see Figure 3.6(a) on page 45).

The Layered Queueing Network Solver implements mixed-model MVA [4, 7, 75] because layer submodels may have both open and closed components. Chain 0 is used to represent the open component in a given submodel (chains $1..K$ represent the closed chains). λ_0 is the arrival rate of requests, either specified as an open arrival stream in the input file, or found during the solution of a submodel and used as a send-no-reply request.

Figure 3.22 shows the mixed-model MVA algorithm. Servers with no closed chains are solved using the function *openWait*(m) in step 3. The $\alpha_m(\mathbf{N})$, *mixedWait*$(m, \mathbf{N})$,

```
proc solve( N )
begin
   for m ← 1 to M do                                   { Initialization }
      for e ← 1 to E_m do
         for k ← 1 to K do
            begin
            L_mek(N) ← N_k/M E_m
            for j ← 1 to K do
               begin
               L_mek(N − e_j) ← (N − e_j)_k/M
               D_mekj(N) ← 0
               end
            end

   for I ← 1 to 2 do
      begin
      core(N)                                          { Step 1 }
      for c ← 1 to K do                                { Step 2 }
         core(N − e_c)

      for m ← 1 to M do                                { Step 3 }
         for e ← 1 to E_m do
            for k ← 1 to K do
               begin
               F_mek(N) ← L_mk(N)/N_k
               for j ← 1 to K do
                  begin
                  F_mek(N − e_j) ← L_mek(N − e_j)/N_k
                  D_mekj(N) ← F_mek(N − e_j) − F_mek(N)
                  end
               end
      end

   core(N)                                             { Step 1 }

end linearizer
```

Figure 3.21: Linearizer for multiple routing chains.

and $openWait(m)$ functions are implemented as `alpha()`, `mixedWait()` and `openWait()` respecitively in class *Server* and its subclasses described below in §3.4.6.

1: { Scale service rates based on open traffic }
$\mu'_m(\mathbf{N}) = \mu_m(\mathbf{N})\frac{\alpha_m(|\mathbf{N}|-1)}{\alpha_m(|\mathbf{N}|)}$
2: { Solve Closed Model with scaled service rates $\mu'_m(\mathbf{N})$.}
3: { Solve Open Model, accounting for closed model traffic.}
$W_{m0} \leftarrow mixedWait(m, \mathbf{N})$

Figure 3.22: Mixed Model MVA

3.4.5 Priorities

Head-of-Line (HOL) and Preemptive-Resume (PR) priority scheduling are implemented for Exact and Approximate MVA using the techniques of Bryant et. al. [8] and Eager et. al. [33] respectively. These techniques solve the equations in Figure 3.18 starting with the chain with the highest priority and finishing with the chain the lowest priority. The waiting times for chains with priorities less than the highest priority are inflated after solution based on the utilization of the higher priority chains.

For a Preemptive-Resume server, any customer with a higher priority than the customer currently in service will preempt the lower priority customer. The change to waiting time expression for this case is simple: the residence time for lower priority customers is inflated by a factor U', shown in (3.6) for the waiting time expression implemented in class *FCFSServer* on page 75.

$$W_{mek}(\mathbf{N}) = \frac{s_{mek} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij}L_{mij}(\mathbf{N}-\mathbf{e}_k)}{1 - U'_{mk}(\mathbf{N})} \tag{3.6}$$

For a Head-of-Line server, the customer is service is allowed to complete regardless of the priority of any arriving customers. Equation (3.6) is further modified to account for the

delay caused by the customer in service.

$$W_{mek}(\mathbf{N}) = \frac{s_{mek} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{N}-\mathbf{e}_k) + \sum_{j=k+1}^{K}\sum_{i=1}^{E_m} s_{mij} U_{mij}(\mathbf{N}-\mathbf{e}_k)}{1 - U'_{mk}(\mathbf{N})} \tag{3.7}$$

The term U'_{mk} in (3.6) and (3.7) is the inflation factor. For exact MVA, U'_{mk} is found using

$$U'_{mk}(\mathbf{N}) = \sum_{j=1}^{k-1} U_{mj}(\mathbf{N} - L_{mj}(\mathbf{N})\mathbf{e}_j) \tag{3.8}$$

The summation is the utilization of all of the higher priority chains with $\mathbf{N} - L_{mj}(\mathbf{N})\mathbf{e}_j$ representing a closed chain population with L_{mj} fewer chain j customers.

For approximate MVA, the solutions at all of the intermediate population vectors are not available. Equation (3.8) is replaced with

$$U'_{mk}(\mathbf{N}) = \sum_{j=1}^{k-1} U_{mj}(\mathbf{N}) - L_{mj}(\mathbf{N})\left[U_{mj}(\mathbf{N}) - U_{mj}(\mathbf{N}-\mathbf{e}_j)\right] \tag{3.9}$$

Priorities have only been implemented for fixed-rate (simple) single and multi-phase servers. Since priorities violate the product form assumptions of routing homogeneity, there are cases where the solution accuracy is poor.

3.4.6 Server Objects

MVA submodels consist of *Server* objects which represent stations in the queueing network. Server objects are created by Entity objects during MVA submodel construction. Subclasses of Server are based on properties such as second phase, highly variable service time, and station multiplicity defined by the corresponding Entity object. Figure 3.23 shows the high-level class hierarchy for server objects. This Section describes all of the server types except for *Reiser-Multi* and *Phased*. Simple Phased-Servers are described in Chapter 5.

Multiservers, both simple and multi-phase, are described in Chapter 6.

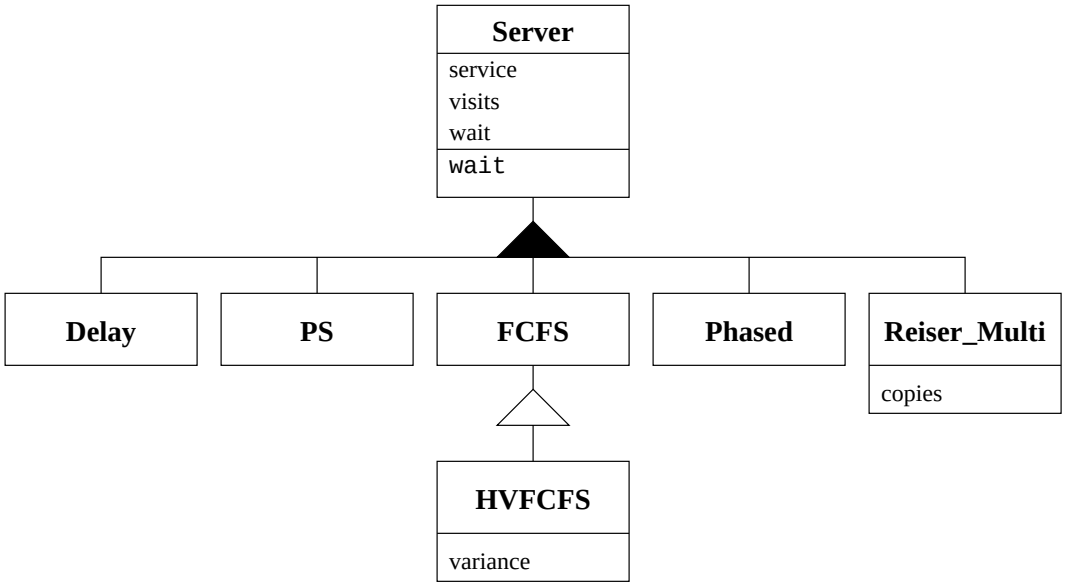


Figure 3.23: Class hierarchy for Server objects.

Class Server

Class *Server* is an abstract super class used to define the interface to `wait()` and `openWait()`. This class is also used to implement `alpha()` and `mixedWait`, needed for open and mixed models, using (3.10) and (3.11) respectively.

$$\alpha(n)_{m0} = 1/\rho_{m0}^{n+1} \tag{3.10}$$

$$W_{me0}(\mathbf{N}) = W_{me0}\left[1 + L_m(\mathbf{N})\right] \tag{3.11}$$

The term W_{me0} in (3.11) is the open chain's waiting time not considering traffic from the closed chains. This term is computed using the `openWait()` function defined in subclasses of class *Server*. Both `alpha()` and `mixedWait()` are overridden in subclasses that have

state-dependent service times (i.e. delay and multiservers).

Class Delay_Server

Customers at a delay server each receive service from their own server, so there is no queueing component to the waiting time; the waiting time is simply the customer's service demand [4].

$$W_{mek}(\mathbf{n}) = s_{mek} \tag{3.12}$$

Equation (3.14) is used to find the waiting time for an open model. This class also overrides the implemenation of `alpha()`, and `mixedWait()` using (3.13), (3.14) and (3.15) respectively, because there is no queueing at a delay server,

$$\alpha(n)_{m0} = \rho^n_{m0} \tag{3.13}$$

$$W_{me0} = s_{me0} \tag{3.14}$$

$$W_{me0}(\mathbf{N}) = s_{me0} \tag{3.15}$$

Class PS_Server

Processor Sharing (PS) Servers are used for stations using an idealized form of round-robin scheduling where the time slice that each task gets is effectively zero. In effect, all tasks are serviced simultaneously, but at a rate inversely proportional to the number of customers at a PS Server in a closed queueing model is shown in (3.16) [107]. Customers at processor sharing servers can have chain-dependent service times.

$$W_{mek}(\mathbf{n}) = s_{mek}\left[1 + \sum_{j=1}^{K}\sum_{i=1}^{E_m} L_{mij}(\mathbf{n} - \mathbf{e}_k)\right] \tag{3.16}$$

This equation is also the product-form expression for First-Come, First Served and Last-Come, First-Served Premptive Resume fixed-rate servers.

Equation (3.17) [73, (3.55)] is the waiting time expression for an open model.

$$W_{me0} = \frac{s_{me0}}{1 - \rho_m} \tag{3.17}$$

Class FCFS_Server

In a product-form queueing network, the service time for customers at a multi-class first-come, first-served fixed rate server must all be identical. When the average service times differ, a useful approximation simply multiplies the service time term in (3.16) through the terms inside the brackets [75], i.e.:

$$W_{mek}(\mathbf{n}) = s_{mek} + \sum_{j=1}^{K} \sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n} - \mathbf{e}_k) \tag{3.18}$$

This equation does not take into account the variance in service time; in cases where the per-class service time is substantially different, large errors will result.

Equation 3.19 [73, (3.48)] is the waiting time expression in an open model.

$$\begin{aligned} W_{me0} &= \frac{\overline{S}_{m0}}{1 - \rho_m} \\ \overline{S}_{m0} &= \frac{\sum_{e=1}^{E_m} \lambda_{me} s_{me0}}{\sum_{e=1}^{E_m} \lambda_{me}} \end{aligned} \tag{3.19}$$

Class HVFCFS_Server

Equation 3.18 tends to be inaccurate when there is a large amount of variability in the service times. A more accurate approximation, by Reiser [108], breaks the time a customer spends waiting at a station, $L_{mej}(\mathbf{n} - \mathbf{e}_k)$, into two components, the time waiting in the queue, $Q_{mej}(\mathbf{n} - \mathbf{e}_k)$, and the remaining time for the customer in service at the station,

r_{mej}, which is calculated as a Mean Residual Life (MRL):

$$W_{mek}(\mathbf{n}) = s_{mek} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij}Q_{mij}(\mathbf{n}-\mathbf{e}_k) + \sum_{j=1}^{K}\sum_{i=1}^{E_m} r_{mij}U_{mij}(\mathbf{n}-\mathbf{e}_k) \quad (3.20)$$

$$r_{mij} = \frac{s_{mij}}{2} + \frac{\sigma_{mij}^2}{2s_{mij}} \quad (3.21)$$

$$Q_{mij}(\mathbf{n}-\mathbf{e}_k) = L_{mij}(\mathbf{n}-\mathbf{e}_k) - U_{mij}(\mathbf{n}-\mathbf{e}_k)$$

The variance, σ_{mij}, is calculated using an auxiliary model, either using the method in [110] or in [152], based on the phase type.

Equation (3.22) [73, (3.54)] finds the waiting time for an open model.

$$W_{me0} = s_{me0} + \frac{\rho_m[\overline{S}_{m0} + \sigma/\overline{S}_{m0}]}{2(1-\rho_m)} \quad (3.22)$$

$$\overline{S}_{m0} = \frac{\sum_{e=1}^{E_m}\lambda_{me0}s_{me0}}{\sum_{e=1}^{E_m}\lambda_{me0}}$$

3.4.7 Conclusions

The new solver LQNS includes elements from many sources, which have been referenced. The author's contribution is the new software design expressed by the class relationship in Section 3.4, the batched and squashed layering strategies described in Section 3.3.2, and from the forwarding transformation described in Section 3.3.1.

The author also defined a new input grammar for the model language, based on the earlier (SRVN) language. Additional features of the solver for algorithms which are new to this thesis are presented in the chapters that follow. A simulator and a translator (to the input language for GreatSPN) have also been written so that the execution time and accuracy of the solver can be compared to simulations and exact results where possible.

Chapter 4

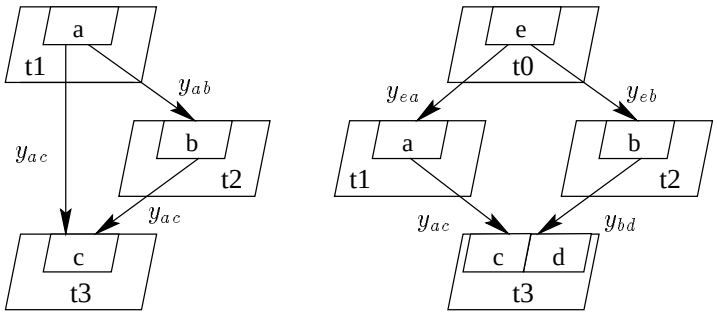
Traffic Dependencies in Layered Queueing Networks

Interlocking is a phenomenon that occurs in the solution of a layered queueing network when a client and its server share a common resource; the resource may be either another software server or a hardware device. Traffic dependencies at the lower-level servers in multipley layered systems in turn affect the prediction of delays. These dependencies arise because the client task can be queued on, blocked on or executing within only one task or device at a time. Solutions which ignore interlocking tend to be pessimistic because arrivals from interlocked clients are treated as independent events when in fact they are correlated. Furthermore, these solutions may show that the lower level servers and devices are not fully utilized when in fact they are. Other authors have recognized that interlocking is a phenomenon which must be considered. However, the SRVN solver [152] only deals with "send" interlocking (see Figure 4.1) while the MOL solver [110] only accounts for interlocking at the device level. The improvement described here generalizes the algorithm of Woodside et. al. to handle a broader range of systems by searching more deeply in the call graph to find these dependencies.

The material in this chapter was mostly published in [39].

4.1 Interlock Phenomena

This section describes the two forms of interlocking found in client-server systems: *Send* (shown in Figure 4.1(a)) and *Split* (Figure 4.1(b)). Interlocking arises when a common parent task makes requests to a server through independent nested RPC's or *call paths* which may pass through intermediate servers in the model. The point where the flow splits along independent paths is called the *"split point"*. For example, in Figure 4.1(a), the first path consists of the arc labeled y_{ac}; the second path consists of the arcs labeled y_{ab} and y_{bc}. In Figure 4.1(b), the first path is comprised of arcs labeled y_{ea} and y_{ac} and the second y_{eb} and y_{bd}. In both cases the set of clients labeled `t0` are the common parents at the split point in the flow and the task labeled `t3` is the common server.



(a) Send Interlock (b) Split Interlock

Figure 4.1: Interlocking arises when traffic flows from intermediate servers (eg. tasks *t1* and *t2*) are correlated due to stimulus from a common source. In both cases, the client cannot be making a request to the server while blocked on the task *t1*.

Send interlocking arises when there are one or more direct paths between the common parent and the common server (for example the arc labeled y_{ab} in Figure 4.1(a)). Both

the set of client tasks at the split point for the flows, `t0`, and the intermediate interlocked task, `t2`, are direct parents of the common server. Split interlocking arises when a common parent task calls independent intermediate tasks which in turn share a common resource. In this case, the split point is not a direct parent of the common resource. In both cases, the common resource may be a device or another task.

Figures 4.1(a) and 4.1(b) show the very simplest forms of interlocking. More complicated patterns can arise, either by adding more tasks in series along a path, or by adding more paths in parallel. An example of a complex interaction pattern can be found in Section 4.2 below.

4.1.1 Factors That Affect Performance Estimation

Four factors affect the amount of error introduced into the solution of the model:

1. The number of clients at a common parent split point. Increasing the number of common parents decreases the effect of interlocking.

2. The number of independent paths from the split point to the common resource. Increasing the number of independent paths increases the effect from interlocked flows.

3. The utilization of the interlocked tasks and devices. The error increases with overall utilization.

4. The amount of included service time in the interlocked tasks. Systems with large amounts of included service time close to the split point will show the largest effects from interlocking.

These factors are illustrated in Section 4.3 with examples.

4.2 Calculation of Contention with Interlocked Flows

The algorithm to estimate and correct for the interlocked flow rates is split into three parts:

1. the location of interlocking *paths*,
2. the calculation of interlocked flow components, and
3. queue adjustments in the MVA solver

described below.

Path Finder

The purpose of the path finder is to identify calls which have a common source, like *t1* in Figure 4.1(a). Paths consist of nested remote procedure calls from an originating client task to a server. They are located by following the requests made from one task to the next using a depth-first recursive search. By incorporating the call rate information associated with each request while tracing a path, the flow component originated by any parent to any entry can be determined. This information is stored by the solver in an $n_e \times n_e$ *path* matrix where n_e is the total number of entries in the model. Each entry in the matrix stores the number of calls to the destination entry caused by an invocation of the source entry. Elements in the matrix are denoted by path(a, b), where a is the source entry and b is the destination. By definition, the diagonal of the matrix is set to 1 indicating that each call to the associated entry causes it to execute once.

Interlocked Flows

The purpose of the interlocked flow finder is to locate all common sources of traffic to a particular *task* in a model since interlocked flows, in general, do not have to go to the same entry. The path matrix table defined in the preceding section is used to meet this objective. Common parents to a task j are located by sequencing through all pairings of incoming arcs that originate from different tasks. The originating entries for each arc, say a and c, are then used as the destination entries in the path table. The table is then searched for

source entries belonging to a common task that calls both entries. This algorithm is shown in Figure 4.2 below.

{ Find parents common to entries a and c. }

```
common_parent( a, c )
   P ← ∅                                        {Set of common entries}
   for i ∈ T do                                 {Search over all tasks, T}
      for e ∈ E(i) do                           {Search over all entries, E(i)}
         for f ∈ E(i) do
            if path(e, a) × path(f, c) > 0 then
               P ← P ∪ e
            endif
         endfor
      endfor
   endfor
   return P                                     {Return the set of common entries}
end common_parent
```

Figure 4.2: Common parent finder for interlocking.

As the path table stores call rates from one entry to another rather than by entry to arc, it is necessary to identify the arcs to the common server by their associated source entries. The flow component from the originating source entry e to the ultimate destination entry c of task j is then found using:

$$\lambda_{ec} = \lambda_e \cdot \text{path}(e, a) \cdot y_{ac} \tag{4.1}$$

where y_{ac} denotes the number of calls from entry a on task i to entry c on task j, $i \neq j$. Furthermore, the throughput at any particular entry a due to calls at e can be found by the sum:

$$\lambda_a = \sum_{e \in \mathcal{C}} \lambda_e \cdot \text{path}(e, a) \tag{4.2}$$

where $\mathcal{C}$ is the set of entries belonging to pure client tasks. In fact, any unique cut-set to the task j can be used in place of $\mathcal{C}$ in (4.2). By knowing the total throughput at an entry a and the throughput at a due to requests from a particular entry e, it is possible to separate flow caused by common parents from flow from other sources.

The common parent finder shown in Figure 4.2 returns a set of entries (denoted as $\mathcal{P}$) that generate non-zero flow to the entries labeled a and c. This set must be pruned of all entries that do not result in the split in flow. The interlocked flow component, λ^{IL}, is then found by using the expression:

$$\lambda_{ac}^{\mathrm{IL}} = y_{ac} \sum_{e \in \mathcal{P}} \lambda_e \cdot \mathrm{path}(e, a) \tag{4.3}$$

Interlock Adjustment

The interlock adjustment is an adjustment to the contention seen by a request to a certain entry. It is accomplished by removing flow which originates from common parents that arrives by different paths, in the calculation of queue lengths for Mean Value Analysis. Equation (4.4) shows the queue length calculation where L_{mk} is the queue length for chain k at station m, $\mathbf{N}$ is the population vector by chain, W_{mk} is the waiting time for chain k at station m and λ_{mk} is the flow to station m for chain k. λ_{mk} us divided into two parts using (4.3), so:

$$L_{mk}(\mathbf{N}) = \lambda_{mk} W_{mk}(\mathbf{N}) \tag{4.4}$$

$$\lambda_{mk} = \lambda_{mk}^{\mathrm{IL}} + \lambda_{mk}^{\mathrm{NOIL}} \tag{4.5}$$

Client classes in each submodel of a layered queueing model are represented by separate chains in the underlying MVA model. Flow from a client task corresponding to chain k to

the serving station m is found using:

$$\lambda_{mk} = \sum_{a \in \mathcal{E}(k)} \sum_{c \in \mathcal{E}(m)} \lambda_{ac} \tag{4.6}$$

where the expression $\mathcal{E}(k)$ denotes the set of entries corresponding to task k.

The interlock adjustment separates the flow from common parents from flow from other sources in λ_{mk}. The interlocked flow is then reduced in proportion to the total number of sources (labeled as n_s in (4.7)) of flow along all chains with interlock. These sources are the tasks at the point where the interlock paths diverge plus the immediate parents of the tasks that lie along the interlocked paths that source non-interlocked flow. In effect, one source out of several that generate the flow rate λ_{mk} is being removed. Equation (4.4) is then replaced with:

$$L_{mk}(\mathbf{N}) = \left(\frac{n_s - 1}{n_s} \lambda_{mk}^{\text{IL}} + \lambda_{mk}^{\text{NOIL}} \right) W_{mk}(\mathbf{N}) \tag{4.7}$$

Interlocking has its greatest effect on accuracy when there are a small number of ultimate sources creating a wide diversity of requests at lower levels. The need to adjust for interlock is a side effect of the layered solution strategy which only models the immediate source of a request.

4.2.1 Path Finder Example

Figure 4.3 shows an example with multiple interlocking paths. The corresponding interlocking path table is shown in Table 4.1.

Consider task `S1`. Its entries are `h`, `i` and `j`. Entry `h` is called by entry `c`, entry `i` is called by entry `d`, and entry `j` is called by both entries `f` and `g`. As entries `c` and `d` belong to a common task, the arcs y_{ch} and y_{di} are not separate paths for interlocked flow (task `I5` serializes the requests), therefore they are not paired when locating common sources. The set of pairings that must be considered is $\{(\texttt{c},\texttt{f}),(\texttt{c},\texttt{g}),(\texttt{d},\texttt{f}),(\texttt{d},\texttt{g}),(\texttt{f},\texttt{g})\}$. For $(\texttt{c},\texttt{f})$ the

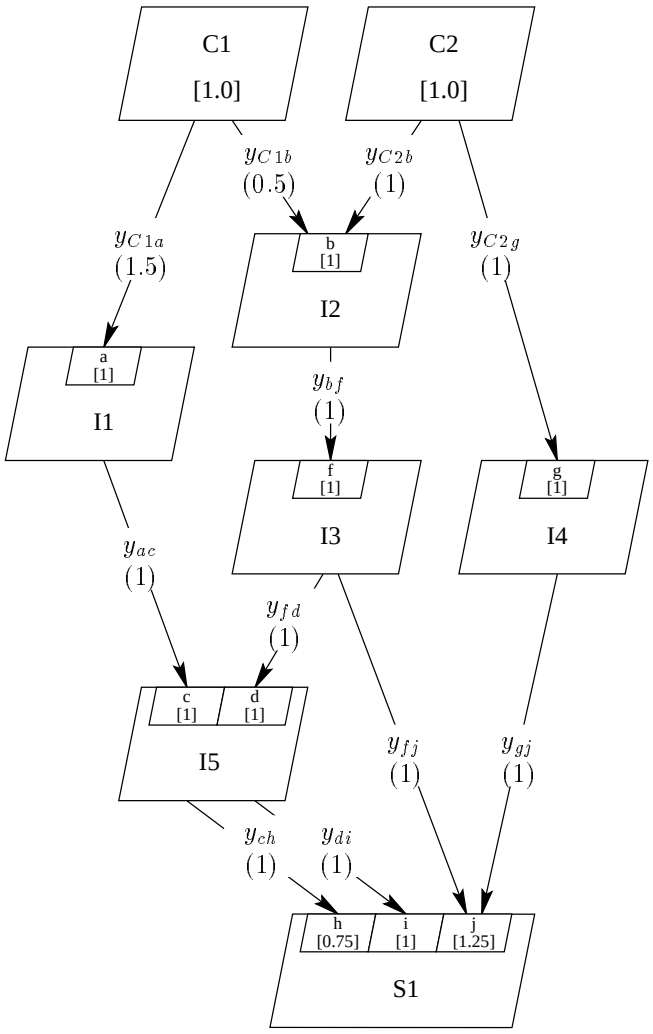


Figure 4.3: Complex Interlock Case. The number immediately below an arc labels is the arc's call rate. The number associated with an entry label is service time.

Complex Interlock											
Src	Destination Entry										
Ent	C1	C2	a	b	c	d	f	g	h	i	j
C1	1.	0.	1.5	0.5	1.5	0.5	0.5	0.	1.5	0.5	0.5
C2	0.	1.	0.	1.	0.	1.	1.	1.	0.	1.	2.
a	0.	0.	1.	0.	1.	0.	0.	0.	1.	0.	0.
b	0.	0.	0.	1.	0.	1.	1.	0.	0.	1.	1.
c	0.	0.	0.	0.	1.	0.	0.	0.	1.	0.	0.
d	0.	0.	0.	0.	0.	1.	0.	0.	0.	1.	0.
f	0.	0.	0.	0.	0.	1.	1.	0.	0.	1.	1.
g	0.	0.	0.	0.	0.	0.	0.	1.	0.	0.	1.
h	0.	0.	0.	0.	0.	0.	0.	0.	1.	0.	0.
i	0.	0.	0.	0.	0.	0.	0.	0.	0.	1.	0.
j	0.	0.	0.	0.	0.	0.	0.	0.	0.	0.	1.

Table 4.1: Path table for complex example of path finding.

only row that has non-zero entries for the columns `c` and `f` is `C1`. There are no rows that meet the criteria for (`c`, `g`). For (`d`, `f`), matches are found for rows `C1`, `C2`, `b` and `f`. This set is pruned to row `f` only as this entry spans the least number of layers to task `S1`. Finally, `C2` is the only common entry for (`d`, `g`) and (`f`, `g`). After pruning entries that do not split flow, the path finding algorithm returns the set {`C1`, `f`, `C2`}.

4.3 Examples

The following examples illustrate the factors identified earlier that affect performance estimation identified earlier and how compensating for interlocked flows improves the solution accuracy. This section concludes with a discussion on how these factors affect performance estimation.

4.3.1 Example 1: Common Server System (Send Interlock)

Figure 4.4 shows a client-server system with a database and a file system running on a common processor. Each client runs on its own processor. The central database computer has two tasks running on it called `Database` and `FileSys` (disks are not shown). The service time for both tasks is 1.0 units. The service time at the client tasks is varied from 0.01 units to 100.0 units by powers of 10. The number of clients is varied from 1 to 50. Each client makes one request per cycle on average. This request causes the `DataBase` tasks to make one request to the `FileSys` task. The request parameters are shown next to the appropriate arc on the figure.

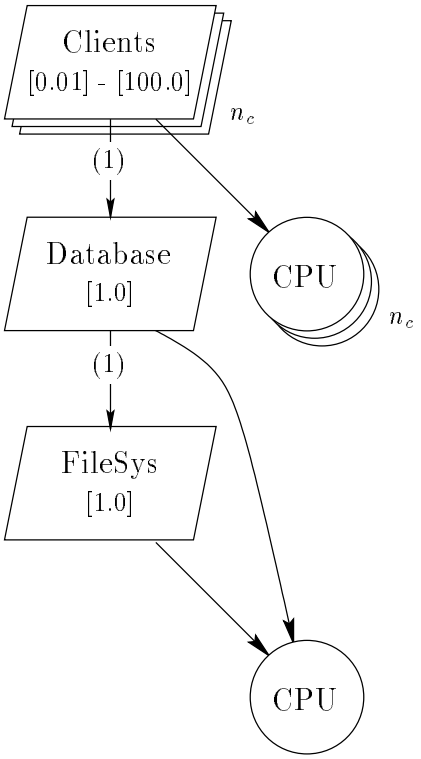


Figure 4.4: Parameters for Example 1 (§4.3.1). Entries are not shown.

The throughputs (among other performance results) were found by an exact Markovian

analysis using the GreatSPN [17] Petri Net solver for the cases with one to four clients, by simulation for the cases with ten and fifty clients and by the layered queuing network solver using MVA for all cases with and without the queue length correction for interlocked flows. Without the interlock correction, the relative error ranges up to a maximum of 32.63% as the demand on the database computer increases. However, with the interlocking correction enabled, the maximum error is only 1.61%, with the typical error less than 0.1%. The relative error for all test cases is shown in Table 4.2.

n_c	Client Service Time									
	% Error, No interlock correction					% Error, With interlock correction				
	0.01	0.10	1.0	10.0	100.0	0.01	0.10	1.0	10.0	100.0
1	32.33	30.78	19.46	1.69	0.02	0.0	0.00	0.00	0.00	0.00
2	32.51	32.39	27.97	3.63	0.04	0.0	0.09	1.47	0.51	0.01
3	32.51	32.51	31.44	6.52	0.06	0.0	0.01	0.48	1.02	0.02
4	32.51	32.51	32.34	10.39	0.08	0.0	0.00	0.04	1.52	0.03
10	32.60†	32.63†	32.55†	30.84†	0.39†	0.16†	0.26†	0.07†	0.67†	0.16†
50	32.60†	32.63†	32.55†	32.59†	24.06†	0.16†	0.26†	0.06†	0.16†	1.61†

Table 4.2: Relative Error in client throughput versus number of clients, n_c, and varying client service time for the system in Figure 4.4. The results marked with '†' were compared against simulation runs with a 95% confidence interval of ±1%.

From Table 4.2, the largest error occurs with the highest customer demand on the database computer. The reason for this relationship can be seen easily from Figure 4.5. This graph shows the waiting time versus utilization for the database system when the number of clients was varied from one to four and the client service time was set to 1.0. Without the interlock correction (the points marked with `NOIL`), the included service introduced by the nested RPC limits the utilization of the processor to about 0.67 because the `DataBase` task spends 1 unit of time at the processor and 1 unit of delay for included service. With the interlock correction, the waiting times calculated by the layered queuing network solver and the Markovian analyzer are nearly identical.

The lesson from this example is that errors can be very large if interlocking is ignored.

In fact, they can be arbitrary large, as is shown next.

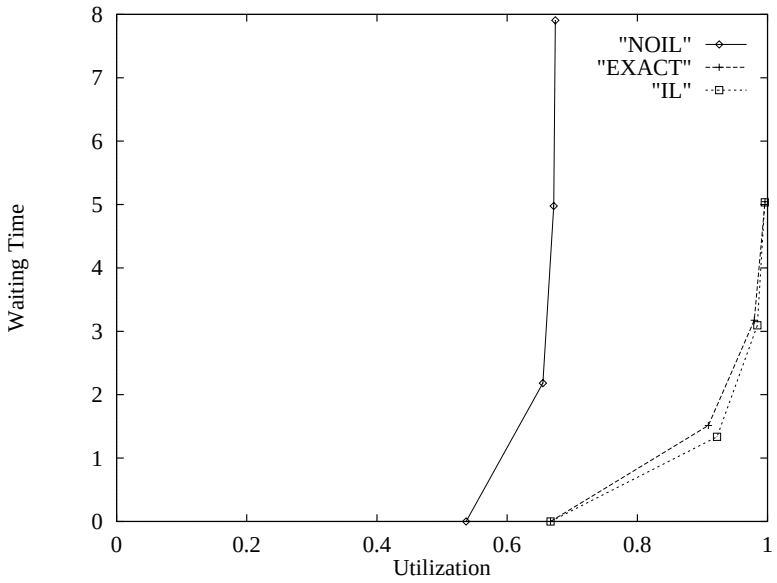


Figure 4.5: Waiting time versus utilization for three clients. Exact results generated using Markovian analysis are labeled `EXACT`, and analytic results using layered queueing networks with and without interlock compensation are labeled `IL` and `NOIL` respectively.

4.3.2 Example 2: Send Interlock

This example is based on the example in the preceding section. However, the service time at the `FileSys` and `DataBase` tasks were varied instead of varying the demand generated by the clients. The number of clients was fixed at 3, and the service time for each client was set to 1 unit. Table 4.3 shows the service times used and the results for the interlocked and non-interlocked layered queueing network solutions using Mean Value Analysis. As the service time of the `FileSys` task was increased, the relative error in the non-interlocked solution also increased. The utilization at the processor was close to 1.0 for all cases.

Srv. Time		Exact	No Interlock		Interlock	
s_{DB}	s_{FS}		Util.	% Δ	Util.	% Δ
100.0	0.01	1.	0.7321	26.79	1.	0.00
10.0	0.1	0.9999	0.7348	26.51	0.9994	0.04
1.0	1.0	0.9796	0.6716	31.44	0.9843	0.48
0.1	10.0	0.9922	0.5033	49.28	0.9994	0.73
0.01	100.0	0.9992	0.5	49.96	1.	0.08

Table 4.3: Utilization and relative error in utilization for the database computer for the service times s_{DB} and s_{FS} of the `DataBase` and `FileSys` tasks respectively. Exact results were generated using Markovian analysis.

4.3.3 Example 3: Split Interlock

Figure 4.6 shows the example used earlier in Section 4.1.1. The parameters for the intermediate tasks, `t1` through `t4` and common server are shown on the figure. The call rates from the common parent `Client` to the intermediate level tasks are shown as $[1/r]$ where r is the number of intermediate tasks (this call rate caused the client throughput to be constant regardless of the number of intermediate tasks for a system with only one client).

Table 4.4 shows the relative error in throughput for each case (the non-interlocked case is shown in the graph in Figure 4.7). As the number of clients was increased, the relative error in the layered queueing network solution without the interlocking correction decreased. As the number of independent paths was increased, the error increased.

n_c	Number of Intermediate Tasks (r)					
	No interlock			Interlock		
	2	3	4	2	3	4
1	5.15	7.15	8.23	0.00	0.00	0.00
2	3.59	6.37	8.11	1.20	0.59	0.12
3	2.61	5.27	7.19	1.65	1.20	0.64
4	2.00	4.26	6.01	1.69	1.62	1.22

Table 4.4: Relative error in client throughput versus number of clients and number of intermediate tasks for the non-interlocked and interlocked solutions for Figure 4.6.

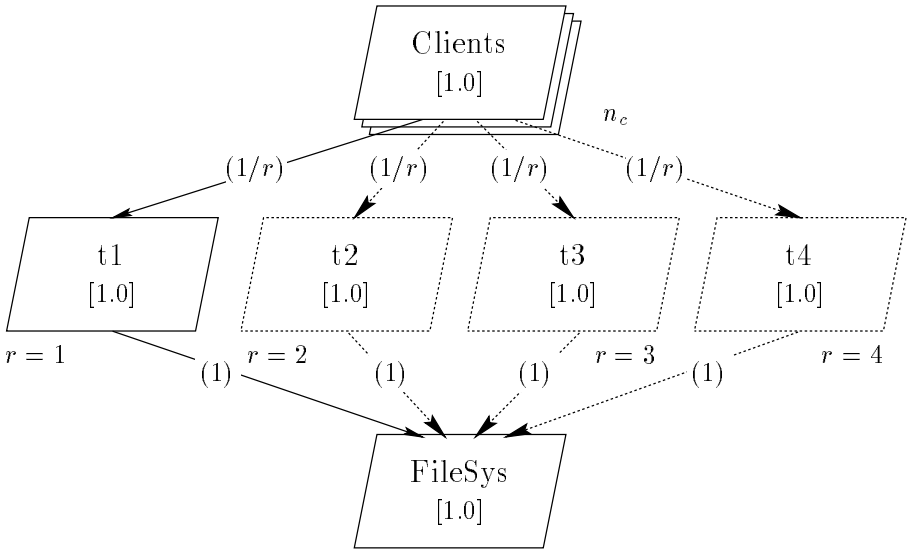


Figure 4.6: Parameters for split interlock model.

4.3.4 Factors that Affect Performance Estimation

Example 3 above demonstrates the effect on solution accuracy caused by the first two factors (common parents and interlocked paths) listed earlier in Section 4.1.1. The results for the non-interlocked solution are plotted in the graph shown in Figure 4.7. Increasing the number of clients at the common parent split point decreased the error in the solution without the interlock compensation because, as the number of clients increased, the likelihood of a request arriving at a common server meeting another request from the same common parent but along a different path decreased. Increasing the number of independent paths increased the solution error caused by interlock flows. When the submodel for the common server is solved without compensating for interlocking, each interlocked call path represents an arrival from an independent source. Increasing the paths increases the number of independent sources.

Example 1 above demonstrates the effect of utilization on a system with send interlocking. As the utilization of the database computer increased, the relative error in the solution

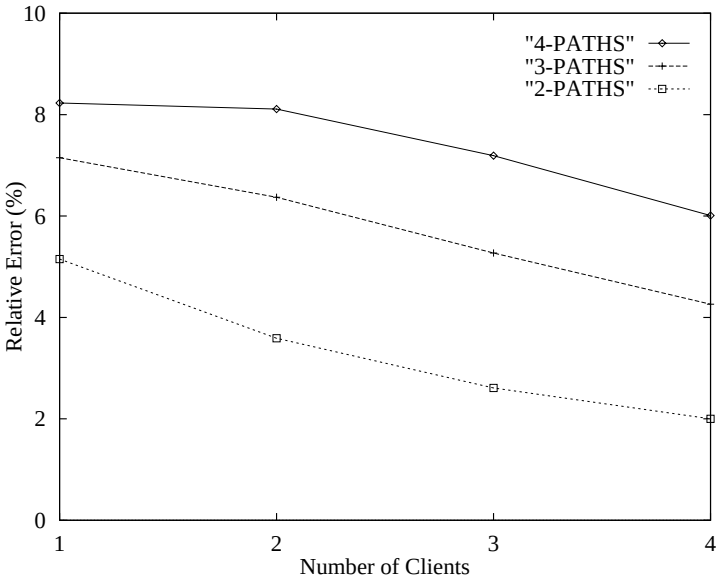


Figure 4.7: Relative Error versus Number of Clients with two, three and four call paths for the split interlock case shown in Figure 4.1(b) and 4.6.

also increased. The upper limit of the utilization for the non-interlocked solution, 0.675 in Figure 4.7, is due to the effect of included service.

Example 2 above shows the effect of included service on the error in the solution. If the bulk of the service time in the interlocked tasks is located at or near the common server (i.e. `DataBase`), then the included service delay will be small and the overall utilization high. However, if the bulk of the service time is in the intermediate task (i.e. `FileSys`), then the included service time at the common parent will be high which will create low utilizations.

4.4 Conclusions

Ignoring interlocking effects when solving multilayer client-server performance models can have a large impact on the accuracy of the solution of the performance model. There are four factors which affect the relative error: the number tasks involved in the interlock, the

number of interlocked paths in the model, the utilization of the interlocked tasks and devices, and the amount of included service in the tasks acting as clients. Increasing the number of tasks associated with the interlocked flow paths, n_s, decreases the relative error as the flow components are adjusted in proportion to $\frac{n_s-1}{n_s}$. Consequently, the error introduced by ignoring interlocking will be quite small as the number of parent tasks increases. Conversely, increasing the number of interlocked paths increases the relative error in the solution because an arrival from an interlocked client task may now encounter its own requests from additional sources. The utilization of the devices and the tasks involved in the interlock also affects solution error. As the utilization goes up, the relative error increases. Finally, systems that have large amounts of included service time in the tasks where the interlocked flows split will also tend to show larger relative amounts of error. This effect will manifest itself with lower than expected utilizations at the common server.

Interlocking can occur on both tasks and devices. The effect occurs whenever requests from a task split then rejoin at a lower level in the model. Tasks running on a common processor that communicate through remote procedure calls can often be a source of solution error.

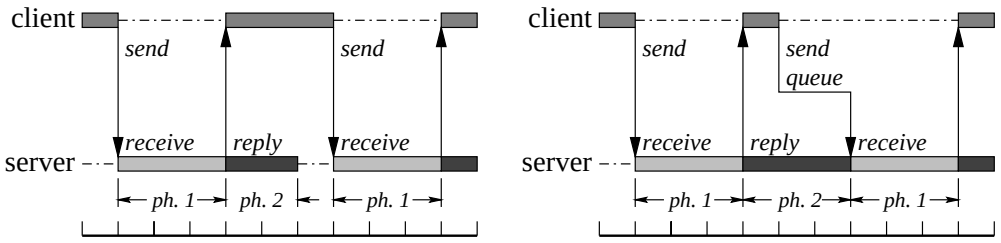
Chapter 5

Second Phases and Phased Fixed-Rate Servers

Second phases arise naturally in the send-receive-reply pattern of communication, which is widely used in distributed systems, often with a Remote Procedure Call (RPC). This pattern imposes waiting on the sender or client in the interaction, which can be reduced by sending the reply as early as possible, referred to as an *early reply*. To perform an early reply, any server operations which can be done later are postponed until after replying; the deferred work is termed a *second phase of service* [92, 76]. In some cases, the early reply will improve performance, since it lets the two processes proceed in parallel. Second phases are common in practice. They perform "cleanup" operations such as delayed writes to stable storage in database systems, logging of non-critical data, and deallocation of resources [26, page 131]. They also may provide autonomous operations by the server, triggered by the client but executed in parallel. Smith and Williams [132] describe a real-time system which includes a second phase. Second phases can also be used to model file systems with delayed writes (see Chapter 9). Many existing distributed operating systems support second phase execution, including Ameoba [136], Chorus [114], V [15], and Sun RPC [26]. Second phase service is

also directly supported in the programming language Ada, in code segments following an "`end accept`" [2].

The two phases of service are illustrated in Figure 5.1(a). The time interval from the instant of the receive at the server to the time of the reply is described as phase one; the server execution after the reply, is phase two. From the standpoint of the clients, the server's phase two is a *vacation* – a client cannot receive service during this period. A request made by a client while the vacation caused by the client's own previous service request is still underway is called an *overtaking* arrival (see Figure 5.1(b)).



(a) Non-overtaking arrival

(b) Overtaking arrival

Figure 5.1: Overtaking and non-overtaking arrivals from a client task i to a server task j.

The use of a second phase improves performance most at lightly loaded servers, as is demonstrated below. If a certain server is the dominant system bottleneck then putting some of its work in second phase will not improve its capacity (if the amount of work is still the same); the bottleneck and associated waiting will remain the same.

For open systems, a single queue with a second phase server has been thoroughly analyzed as a type GNENP (General Non-Exhaustive Service, Non-Preemptive), SV (Single Vacation) server [32, §2]. An early version of the model, the "walking server" [44, Chapter 2] has been used to model drum-based memories [72] and other computer devices. However, in general there has been little work on closed models.

For a single finite source queue with two-phase service, Petriu and Woodside described an approximation based on Markov Chain decomposition, and showed how it can be extended to approximate a closed queueing network and a layered queueing network [100]. This kind of server was incorporated into the SRVN [152] and MOL [113] solvers. Both of these implementations give large errors for some cases, as will be shown. The errors are particularly serious when the client process also has two phases and they have different service interactions.

This work describes an improved approximation which takes into account the phase of the client when it sends a request, its request behaviour in each phase, and the effect of second phases on the queue length calculation in Mean Value Analysis (MVA). This reduced the worst-case errors by up to an order of magnitude in the examples reported below.

5.1 Performance Enhancement through Early Server Replies

This section will explore how system throughput is improved by using the second phase in a design. Three cases are considered: a very simple system consisting of a set of clients calling a single server, a more complex case where two-phase clients call three two-phase servers, and a deeply layered system with four tiers of two-phase servers.

5.1.1 Example 1: Single Server, Single Phase Client

The simple client-server system system shown in Figure 5.2 is the first example. The set of client tasks i execute a cycle, and have only one phase (i.e. $s_{i1} = 0, y_{ij1} = 0$) in each cycle. At the set of clients, the CPU demands were $s_{i2} = 4$. At the server $s_{j1} + s_{j2} = 1$, and the fraction $\alpha = s_{j1}/(s_{j1} + s_{j2})$ was varied from 0 to 1. When the phase-one fraction α equals 0, all the service time is in phase 2 and the model is representative of messaging employed in transputers [53]. When α equals 1, the model represents a standard RPC client-server interaction which has a simple product-form queueing model with m_i customers, that

alternate between two classes for phase 1 and phase 2. The mean number of requests from client i to server j per client cycle, y_{ij2}, was varied from 0.5 to 20 in order to vary the load on the server. Finally, the number of client tasks m_i was varied from 1 to 10.

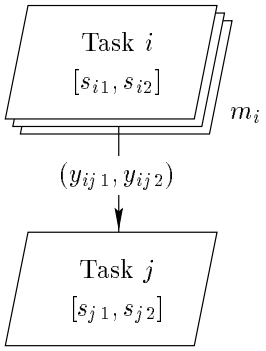
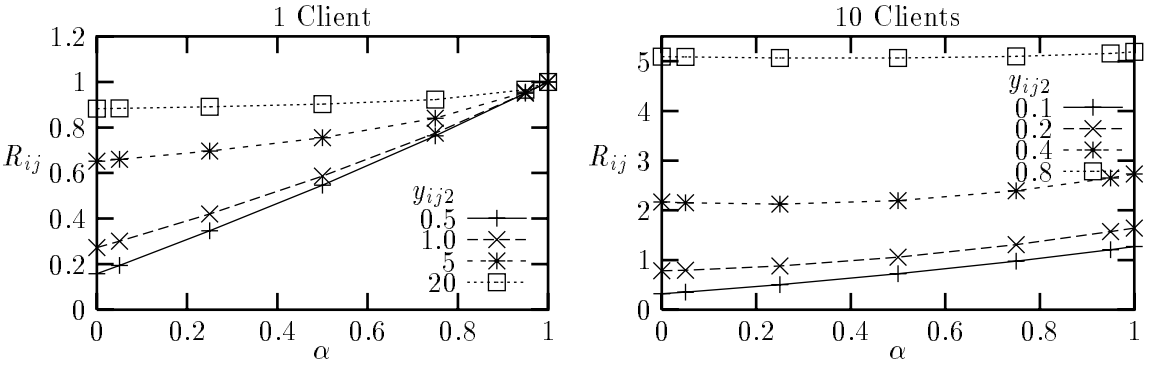


Figure 5.2: Example 1: Simple two-level client-server system. Service times for task i in phase p are shown as s_{ip}. The number of calls from task i in phase p to task j is shown as y_{ijp}.

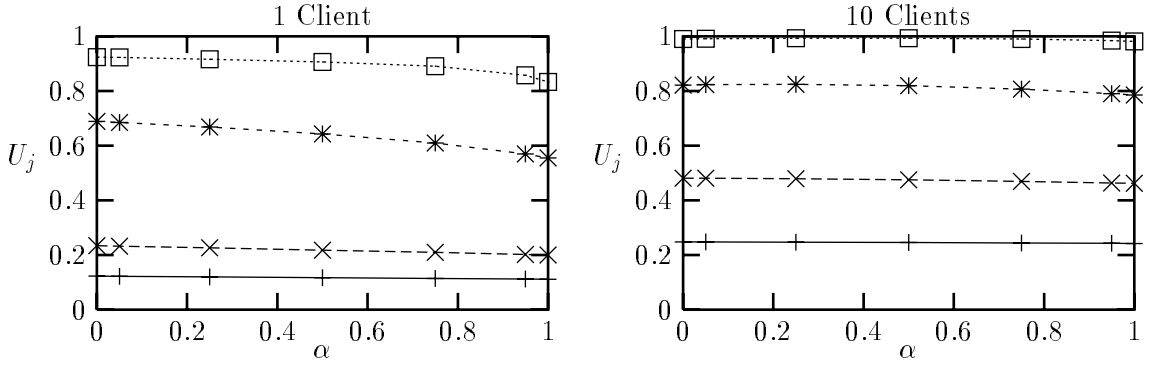
Results were obtained for cases with exponential service time distributions and random service requests, by solving the underlying Markov chain for the entire system, using the GreatSPN Petri net performance tool [17]. Figure 5.3 shows the response time and utilization results for the configurations with one and ten client tasks. When server task utilization is low and the number of clients is small, replying as early as possible can have substantial performance benefits. When the server is saturated the second phase is immaterial. When there are many clients the impact of the second phase is reduced.

5.1.2 Example 2: Multiple Servers and Two Phase Clients

In this example, shown in Figure 5.4 the client cycle goes through two phases with different server demands, and there are three servers. The number of clients, m_0, the fraction of phase-one to total service time at each of the servers, α, and the number of visits from the client task to the server tasks, y, were varied. The total number of requests from client task



(a) Per-call response time for each request from client to server



(b) Server utilization

Figure 5.3: Performance results for Example 1 shown in Figure 5.3. α varies from 9 (all phase 2) to 1 (all phase 1). Response times, R, show the most improvements when utilization at the server is low.

0 to each of the servers was the same, however, the fraction of requests from phase one and phase two of the client was different for each server. The service time at the client task was fixed at 0.5 for phases one and two.

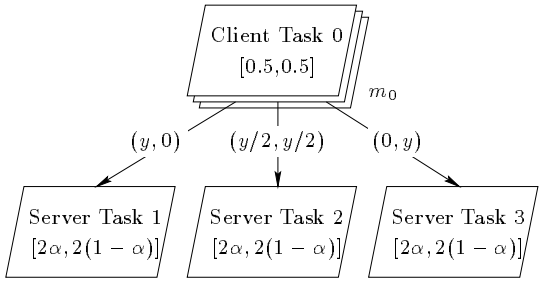


Figure 5.4: Example 2: Complex Client-Server system.

Figure 5.5 shows the response time and utilizations for the system in Figure 5.4 with $m_0 = 1, 3, 5$ and 10 client tasks. The request rate, y, from the client task 0 to the servers was varied to examine the effects of server load and is shown on each of the graphs in Figure 5.5. Exact results were obtained from a Markov model for 1, 3 and 5 client tasks. Simulation was used for the 10 client case because of state space problems with the Markovian solver. The simulation results were generated with 95% confidence intervals of $\pm 0.5\%$.

In all the cases studied, the worst response times occurred when the fraction of phase one service time $\alpha = 1$ (i.e., no phase two). At low utilizations (small y), the best response time occurred when $\alpha = 0$ (i.e., immediate reply to client). At high utilizations, the best response times occurred at some intermediate value of α. For the other servers the curves are not shown but the same observations hold.

5.1.3 Example 3: Deeply Layered System

Figure 5.6 shows a deeply-layered client-server system with multiple phases at all levels. The total service time at each of the tasks was fixed at 1.0, but the fraction of phase-one service time, α, was varied from 0 to 1. Each task made one request to its immediate

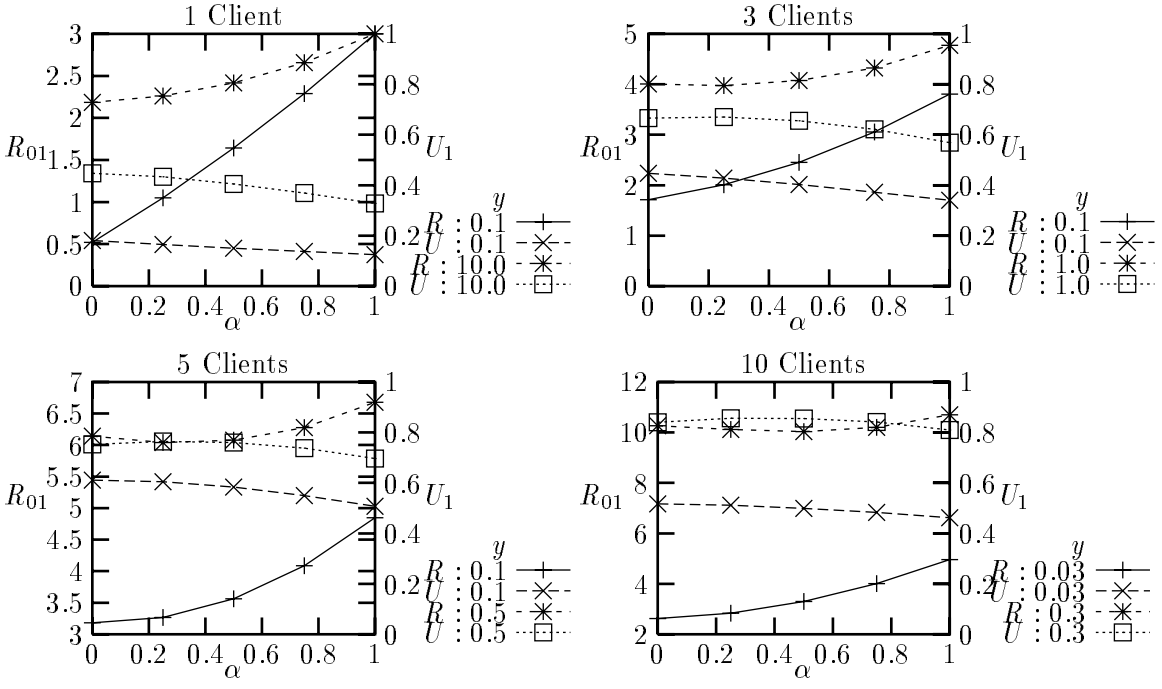


Figure 5.5: Per-call response time for a request from Client task 0 to Server task 1 and Utilization at Server Task 1 for the model in Figure 5.4.

lower-level server from phase one only. The number of client tasks, m_1, was set at 1, 2 and 3 again to study the effect of utilization on throughput and response time.

Figure 5.7 shows response time verses the number of client tasks. This model shows that early replies always improve performance.

5.1.4 Conclusions

To get more insight into the influence of server utilization and number of clients, the results of Example 1 were analyzed further. A naive ideal effect of a factor α is a reduction in response time by the same factor. Ideally then we might hope that $R(\alpha) = \alpha R(1)$. The

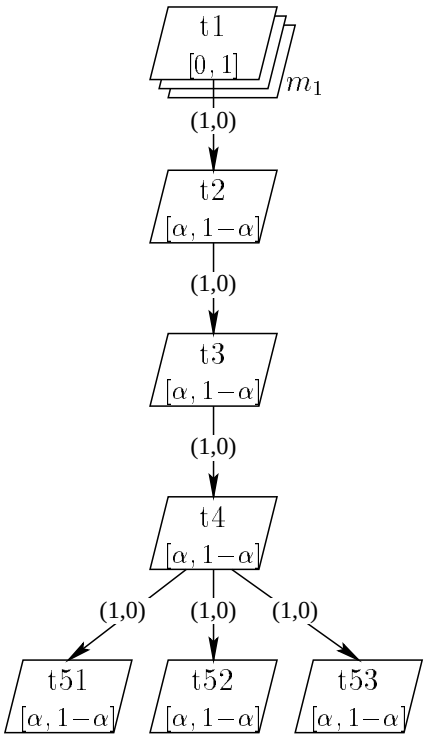


Figure 5.6: Example 3: Deeply layered (multiple tier) client server system.

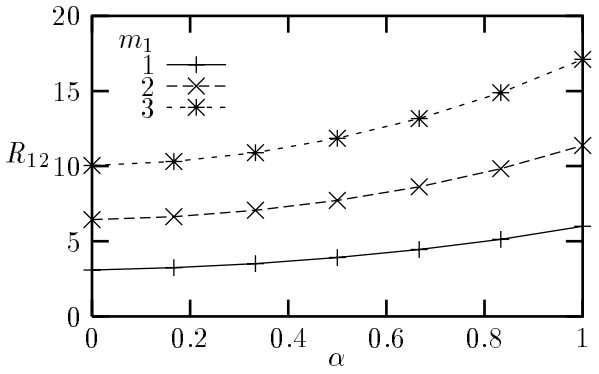


Figure 5.7: Per-call response time for client task t1 for the model in Figure5.6.

following factor F measures how close the actual $R(\alpha)$ comes to the ideal:

$$F = \frac{R_{ij}(\alpha) - \alpha R_{ij}(1)}{(1-\alpha)R_{ij}(1)} \tag{5.1}$$

where $F = 0$ at the ideal improvement ($R(\alpha) = \alpha R(1)$), and $F = 1$ for no speedup ($R(\alpha) = R(1)$). Figure 5.8 show the results for Example 1 when $\alpha = 0.5$. Clearly the improvement is close to ideal at low server utilization and vanishes as utilization increases. At 80% utilization one obtains about 30% of the ideal improvement.

From the figure, the number of clients using the server has little effect on the improvement, except as it affects utilization.

5.2 Analytic Approximations for Two Phase Queues

The results from the previous section demonstrate that early replies can shorten the response times of remote procedure calls. This section addresses the question of analytic approximations for two-phase servers within a layered queueing model for the software. The earlier MOL and SRVN approximations for second phases are described and shown to give large errors in certain cases, and then in Section 5.3 a new and better approximation

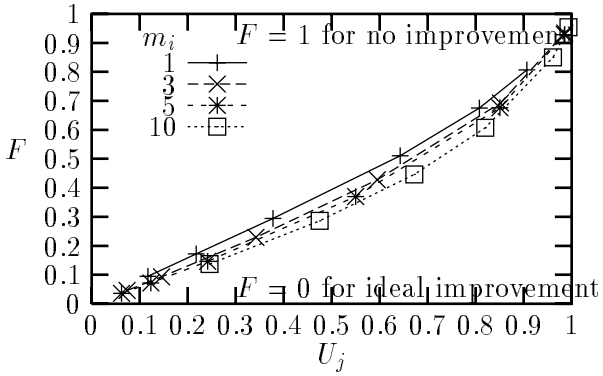


Figure 5.8: Improvement factor F for Example 1 in Figure 5.2 when $\alpha = 0.5$.

is described.

5.2.1 Elementary Analysis of Overtaking

This section summarizes briefly the overtaking approximations used by the Method of Layers and Stochastic Rendezvous Networks. The argument centers on the mean-value equation for the mean total delay $W_j(N)$ of a customer arriving at a queueing station labeled j, with a load-independent server (e.g. a FIFO or PS server),

$$W_j(N) = s_j + s_j L_j(N-1) \tag{5.2}$$

where s_j is the delay for the customer's own service, and $s_j L_j(N-1)$ is the delay due to a random number of customers present at the arrival instant. For a two phase server, the second term is unchanged, but the first term is replaced to give:

$$W_j = s_{j1} + \Pr\{\text{OT}\}_j s_{j2} + s_j L_j(N-1) \tag{5.3}$$

where:

- s_{j1} is the mean delay for service, which is just the phase-one service time, and
- $\Pr\{\mathrm{OT}\}_j s_{j2}$ is the probability of the customer arriving during the phase-two vacation caused by its own previous visit to the server multiplied by the phase-two service time.

Both s_{j1} and s_{j2} are random variables with exponential distributions. Equation (5.3) is the MOL calculation [113]. The SRVN waiting calculation is more complex, using estimates of the state of each client that may contribute to the waiting of the arriving client; it is described in [152];

Equation (5.3) is approximate, since two phase service does not satisfy the product-form assumptions. The overtaking term also assumes exponentially-distributed phase-two times, to make the remaining phase-two time equal to its mean.

5.2.2 Server-Based Approximation for $\Pr\{\mathrm{OT}\}$

After a departure of a customer from the server, the probability of it overtaking depends on the outcome of a race between the completion of phase two (mean delay of s_{j2}) and the return of the customer, shown in Figure 5.1(b). We will assume that the return delay has a mean rate of τ_j^{-1} and is also exponentially distributed for solution convenience. The probability of the return event winning the race and causing an overtaking event is then:

$$\Pr\{\mathrm{OT}\}_j = \frac{\tau_j^{-1}}{(\tau_j^{-1} + s_j^{-1})}$$

which simplifies to

$$\Pr\{\mathrm{OT}\}_j = \frac{s_{j2}}{\tau_j + s_{j2}}$$

The mean return time τ_j can be calculated if we know the mean arrival rate of requests at the server from this particular customer, λ_j, and the mean total delay at the server, W_j, as

$$\tau_j = \frac{1}{\lambda_j} - W_j$$

giving:

$$\Pr\{\mathrm{OT}\}_j = \frac{s_{j2}}{s_{j2} + 1/\lambda_j - W_j} \tag{5.4}$$

Equations (5.3) and (5.4) make up the server-based overtaking approximation used by the MOL algorithm, and by the SRVN algorithm as described in [152].

5.2.3 Accuracy of the Server-Based Approach

Now consider client tasks which themselves have second phases, because they are servers to higher-level tasks in a deeply layered system, as discussed earlier in this chapter. Example 2 has two-phase clients, and Figure 5.9 shows that using (5.4) may give errors up to 50% in the throughput calculation. The principle reason for the large error is that (5.4) does not consider the variation in the rate of arrivals between the phases of the client. The exact Markov solution for this example with $y = 3$ and $\alpha = 0.5$ has overtaking probabilities of 0.535 at Server tasks 1 and 3, and 0.354 at Server task 2. Using the service times found from the exact solution in (5.4) gives only 0.231 for all three servers.

Figure 5.9 shows the percent relative error[1] in throughput using (5.3) and (5.4) for the example system when the proportion of phase-one to phase-two service time was varied from 0 to 100% for a variety of clients. Very large errors occur even for moderate amounts of phase 2 service.

5.3 Client-Server Based Approximation

The new Client-Server Based overtaking approximation is based on a transient sequence of states following a reply from a server j to a client i which is in phase p, shown in Figure 5.10. This start state is called p, S. The purpose of the analysis is to find the probability of exiting the chain at the absorbing state "**OT,**x", which designates overtaking while the server is in

[1] Percent relative error $= \frac{\text{estimate}-\text{exact}}{\text{exact}} \times 100$.

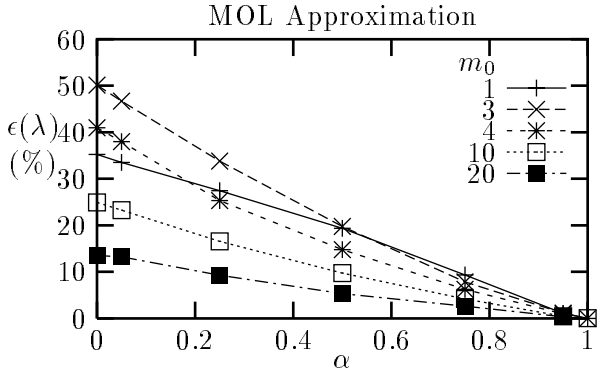


Figure 5.9: Percent Relative Error in throughput, $\epsilon(\lambda)$ for the multiple servers, two-phase client system in Figure 5.4 using the server-based overtaking approximation. The request rate was fixed at $y = 1$.

phase x, or in state "**Done**" which designates the end of the server's execution.

The states are labeled (p, x, k) according to the client phase p, the server phase x, and a value k:

$k = 0$**:** client i executing,

$k = 1$**:** client i making a choice among possible servers, including the server j being analyzed

$k = 2$**:** client i blocked on the chosen server $k, k \neq j$, waiting for a reply.

The states $(0, 2, 0), (0, 3, 0)$ ...$(0, x, 0)$, represent the states in which the client itself is idle.

This is a discrete-time or jump chain for the process, with probabilities q for each transition. The probabilities q are found from the service times and request rates from client i to server j and to all other servers called by client i, represented by k. The parameters from the solution of a submodel of a layered queueing network are:

y_{ijp} = the mean number of calls from client i to server j during client phase p.

Y_{ip} = the mean number of calls from client i to any server j or k during client phase p, $Y_{ip} = \sum_k y_{ikp}$.

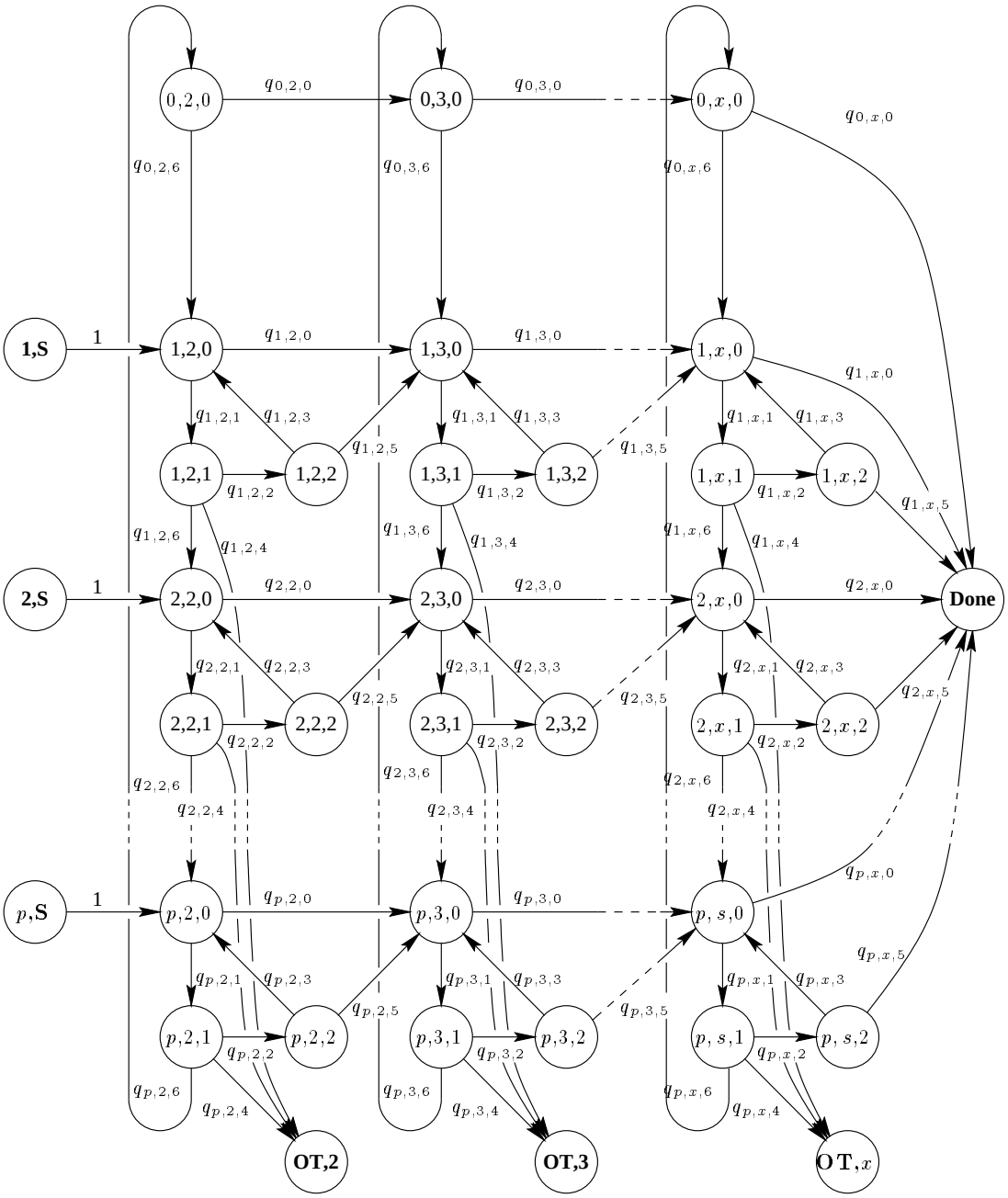


Figure 5.10: Jump chain for finding overtaking probabilities. The states are named with the three digit combination p, x, k where p represents the phase of the client, x designates the phase of the server, $k = 0$ represents client execution, $k = 1$ represents the client sending to a server or going to its next phase, and $k = 2$ designates server execution.

w_{ijp} = the mean waiting time for the client i to the server j during phase p.

λ_{ij} = the mean throughput from the client i to the server j during phase p.

μ_{ip} = the mean execution rate of the client i in phase p between requests to servers.

μ_{kp} = the mean delay of the client i when blocked on a server other than server j in phase p during a remote procedure call, $\mu_{kp} = 1/\sum_k w_{ikp}$.

μ_{jx} = the mean execution rate of server j in phase $x, x \geq 2$, $\mu_{jx} = 1/s_{jx}$.

μ_{i0} = the inverse of the mean idle time of the client i.

The transition probabilities are labeled (p, x, k) according to the client phase p, the server phase x, and a value k:

$q_{p,x,0} = \mu_{jx}/(\mu_{kp} + \mu_{jx})$: the probability that server j completes its own phase-two service before the client i finishes its execution interval.

$q_{p,x,1} = \mu_{ip}/(\mu_{ip} + \mu_{jx})$: the probability that client i completes its own execution interval before server j completes its phase two service.

$q_{p,x,2} = (Y_{ip} - y_{ijp})/(Y_{ip} + 1)$: the probability that client i calls some other server other than server j.

$q_{p,x,3} = \mu_{kp}/(\mu_{kp} + \mu_{jx})$: the probability that some other server k replies to client i before server j completes its phase-two service.

$q_{p,x,4} = y_{ijp}/(Y_{ip} + 1)$: the probability that client i calls server j and overtakes.

$q_{p,x,5} = \mu_{jx}/(\mu_{ip} + \mu_{jx})$: the probability that server j completes its phase two service before some other server k replies to client i.

$q_{p,x,6} = 1/(Y_{ip} + 1)$: the probability that client i completes phase p and goes to phase $p+1$.

The chain shown in Figure 5.10 has a product form solution and is evaluated from left to right by column using:

$$\begin{aligned}
\Pr(\mathrm{OT}_p|S_{r,x}) &= \frac{c_p \prod_{y=r}^{p-1} \frac{b_y}{1-d_y}}{1-\left(b_p \prod_{x\neq p} \frac{b_x}{1-d_x} + d_p\right)} \\
b_p &= \frac{1}{1+Y_{ip}+\mu_{jx}s_{ip}} \\
c_p &= \frac{y_{ijp}}{1+Y_{ip}+\mu_{jx}s_{ip}} \\
d_p &= \frac{Y_{ip}-y_{ijp}}{1+Y_{ip}+\mu_{jx}s_{ip}} \cdot \frac{\mu_{kp}}{\mu_{kp}+\mu_{jx}}
\end{aligned} \tag{5.5}$$

The solution to this equation, $\Pr(\mathrm{OT}_p|S_{r,x})$, gives the absorption probability for overtaking with the client in phase p, when the server, while in phase x, is completing a previous request made from the client's phase r.

The final overtaking probability, when the server is in phase x, is:

$$\Pr\{\mathrm{OT}(x)\} = \sum_{p=1}^{P_i} \sum_{r=0}^{P_i} \Pr\{S_{p,x,0}\} \Pr(\mathrm{OT}_p|S_{r,x}) \tag{5.6}$$

where P_i is the total number of phases of the client i and $\Pr\{S_{p,x,0}\}$ is the probability of starting from state $(p, x, 0)$.

To start the analysis, the initial starting probabilities for the states $(1, S), (2, S), ..., (p, S)$ are found using:

$$\Pr\{S_{p,2,0}\} = \frac{\lambda_i y_{ijp}}{\lambda_{ij}} \tag{5.7}$$

These probabilities are simply the ratio of the flow from client i in phase p to server j over the total flow from client i to server j. These probabilities are then used in (5.6).

To continue the analysis for $x > 2$, the following equation is used to find the absorption

probabilites for the states $(p, x, 0)$:

$$\Pr(\text{NEXT}_p|S_{r,x}) = \frac{a_p \prod_{y=r}^{p-1} \frac{b_y}{1-d_y}}{1 - \left(b_p \prod_{x \neq p} \frac{b_x}{1-d_x} + d_p \right)} \tag{5.8}$$

The solution to this equation, $\Pr(\text{NEXT}_p|S_{r,x})$, gives the absorption probability for overtaking with the client in phase p, when the server, while in phase x, is completing a previous request made from the client's phase r[2]. To find the probability of ending in state $(p, x+1, 0)$, when the server is in phase x:

$$\Pr\{S_{p,x+1,0}\} = \sum_{p=1}^{P_i} \sum_{r=0}^{P_i} \Pr\{S_{p,x,0}\} \Pr(\text{NEXT}_p|S_{r,x}) \tag{5.9}$$

where P_i is the total number of phases of the client i and $\Pr\{S_{p,x,0}\}$ is the probability of starting from state $(p, x, 0)$. The results from (5.9) are then used in (5.5) and (5.6) to find the solution for the next column in the chain.

5.3.1 Improved Waiting Time Calculation

The MVA equation (5.3) for the waiting time also has a flaw buried in the queue length calculation. $L_j(N-1)$ is the mean number of clients at the server, which are either in the queue or in phase-one service. At an overtaking arrival the server is conditioned to be in phase two by definition and this term needs to be modified.

The approximation developed here is based on the assumption that, at the instant of a non-overtaking arrival, the distribution of states of the server is given by its steady state distribution in a system with one less client in the same chain. Notice that the probability that the server is in phase p is a fraction of the total server utilization, $U_j(N-1)$, and is

[2]Equation (5.8) is almost the same as (5.5) except a_c is used in the numerator instead of c_p.

given by $U_j(N-1)s_{jp}/s_j$ (where $s_j = s_{j1} + s_{j2}$).

The waiting time at station j is one's own service, s_j plus the service time for jobs in the queue ahead of oneself, plus the mean residual life (MRL) of the job in service at the time of one's arrival:

$$W_j = s_{j1} + s_j Q_j(N-1) + \mathrm{MRL}_j \tag{5.10}$$

The MRL term is:

$$\mathrm{MRL}_j = \Pr\{\mathrm{OT}\}_j s_{j2} + (1 - \Pr\{\mathrm{OT}\}_j) \begin{bmatrix} \Pr(\mathrm{idle}_j) \times 0 \\ \Pr(p_j = 1) \times \mathrm{MRL}_1 \\ \Pr(p_j = 2) \times \mathrm{MRL}_2 \end{bmatrix} \tag{5.11}$$

$$\begin{aligned} \mathrm{MRL}_1 &= s_{j1} + s_{j2} = s_j \\ \mathrm{MRL}_2 &= s_{j2} \end{aligned}$$

If:

$$\Pr(p_j = 1) + \Pr(p_j = 2) = U_j(N-1)$$

$$\Pr(p_j = 1) = \frac{s_{j1}}{s_j} U_j(N-1) \tag{5.12}$$

$$\Pr(p_j = 2) = \frac{s_{j2}}{s_j} U_j(N-1) \tag{5.13}$$

Collecting (5.12), (5.13) and (5.11) into (5.10):

$$\begin{aligned} W_j = {} & s_{j1} + s_j Q_j(N-1) + \Pr\{\mathrm{OT}\}_j s_{j2} \\ & + (1 - \Pr\{\mathrm{OT}\}_j) \left[s_{j1} + \frac{s_{j2}^2}{s_j} \right] U_j(N-1) \end{aligned} \tag{5.14}$$

Phase-two service is *extra* utilization. Therefore, the number of customers in queue and in service, not counting the phase-two "customers" is:

$$s_j L_j(N-1) = s_j Q_j(N-1) + s_{j1} U_j(N-1) \tag{5.15}$$

Substituting (5.15) into (5.14) results in (5.16), shown below:

$$\begin{aligned} W_{ij} = {} & s_{j1} + s_j L_j(N-1) + \Pr\{\mathrm{OT}\}_j s_{j2} \\ & - s_{j1} U_j(N-1) + (1 - \Pr\{\mathrm{OT}\}_j) \left[s_{j1} + \frac{s_{j2}^2}{s_j} \right] U_j(N-1) \end{aligned} \tag{5.16}$$

Equations (5.6) and (5.16) are used in by the LQNS solution for servers with two phases. Equation (5.6) has also been incorporated into the SRVN solver.

5.3.2 Entries

To evaluate a system with entries, the overtaking probabilities are computed on an entry to entry basis using the Markov chain in Figure 5.10. In all of the expressions in Section 5.3, the indecies i and j are changed to refer to entry i on the source task and entry j on the destination task. The calculation of the starting probabilities, (5.7), is changed to:

$$\Pr\{S_{p,2,0}\} = \frac{\lambda_a y_{ijp}}{\lambda_{ab}} \tag{5.17}$$

where a is the source task and owner of entry i, and b is the destination task and owner of entry j. λ_a is the total throughput at the task a, and λ_{ab} is the total throughput from task a to task b.

5.4 Improved Accuracy of the LQNS Approximation

The three examples presented earlier in Section 5.1 plus the examples from [152] will be used to show the improvements in accuracy using the new Client-Server Based LQNS approximation. Exact results were from a Markovian analysis using GreatSPN [17, 18].

5.4.1 Example 1: Single Server, Single Phase Client

Figure 5.11 contains twelve graphs, all of which plot the relative error in client throughput, weighted by throughput, for the four cases in Section 5.1.1. The three analytic approximations, MOL, SRVN and LQNS, are compared against exact results for all cases. The graphs show that the LQNS approximation is superior, especially when the server is heavily utilized (note the difference in the vertical scales!)

Both the MOL and SRVN solutions suffer when more than one client is present. Since the client task in this example only has one phase, the improvements in accuracy are due to the improved waiting time expression (5.16). The reduction in the relative error ranges up to an order of magnitude in the heavily loaded cases.

The accuracy of the response time calculation improves by an even greater percentage because it is more sensitive to the change.

5.4.2 Example 2: Multiple Servers and Two Phase Clients

Figure 5.12 shows the relative error in throughput for the system in Section 5.1.2. The approximations were compared against exact results except for the 10-customer case where simulation was used. The graphs show that the relative error for the LQNS approximation is much better than the MOL and SRVN algorithm, especially when the server is heavily utilized. The SRVN algorithm, coupled with the client-server-based overtaking approximation (5.6) is also superior to the MOL when the fraction of phase-two service is high. The improvements again range up to nearly an order of magnitude reduction in the error.

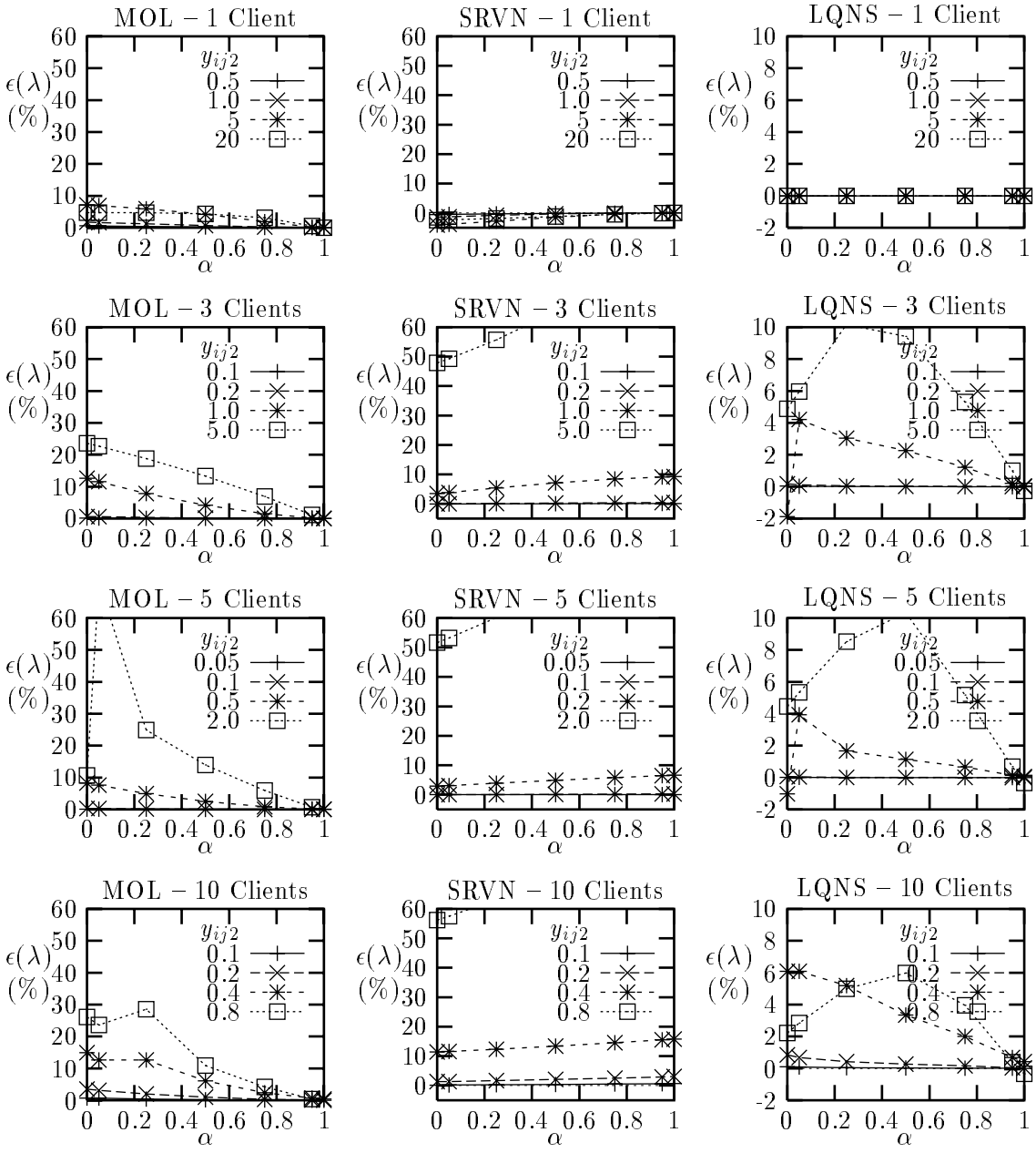


Figure 5.11: Relative error in client throughput, ϵ versus phase-one service time fraction α for the Single-Server, Single Phase Client system in §5.1.1.

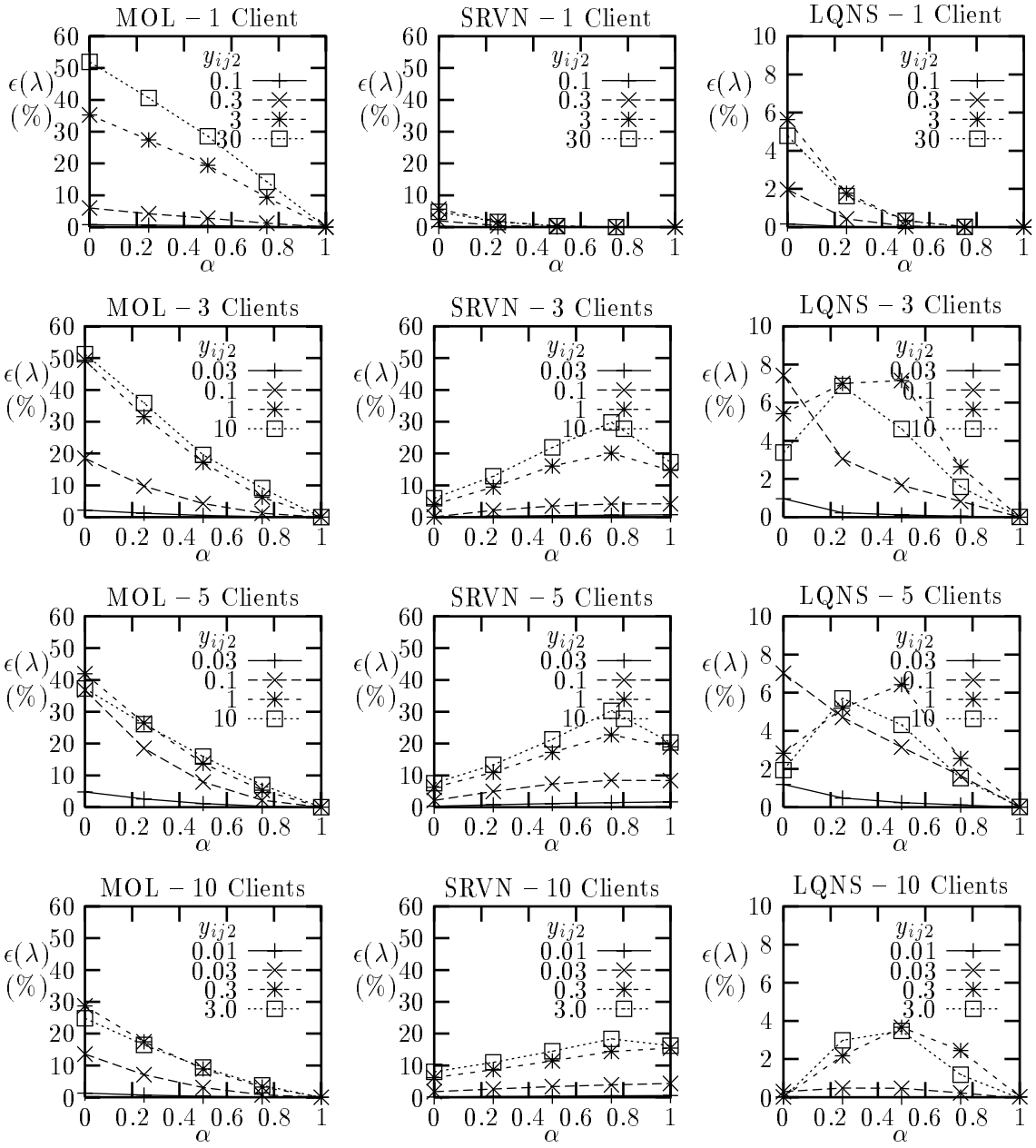


Figure 5.12: Relative error in client throughput, ϵ versus phase-one service time fraction α for the Three Server Two Phase Client system in §5.1.2.

5.4.3 Example 3: Deeply Layered System

Figure 5.1.3 shows the relative error in throughput for the deeply layered system in Section 5.1.3. The LQNS solvers is more accurate than the MOL and SRVN solutions for low phase-one service time fractions. In this case also the relative error is reduced by up to an order of magnitude.

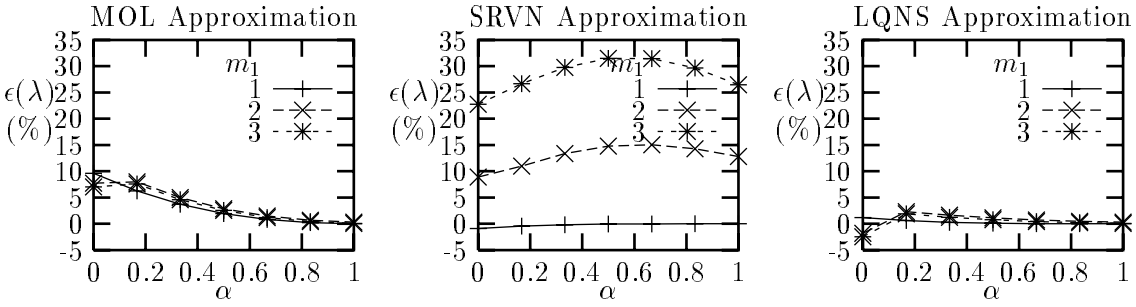


Figure 5.13: Relative error in client throughput, ϵ versus phase-one service time fraction α for the deeply layered system in §5.1.3.

5.4.4 Example 4: Woodside et. al. Test Case

Tables 5.1 and 5.2 and show the throughput of the reference tasks of the twenty test cases presented in [152]. The throughputs found using the LQNS, SRVN and MOL approximations were compared against exact results (except as noted with '†'). The SRVN results reported in these tables are from the algorithms in 5.2 and do not include the client-server-based overtaking approximation.

Table 5.3 summarizes the relative error results from Tables 5.1 and 5.2. The server-based SRVN and MOL approximations have higher average errors than the newer client-server-based LQNS approximation; the SRVN approximation tends to underestimate throughput while the MOL approximation does the reverse.

Case	Task	Throughput						
		Exact	LQNS		SRVN		MOL	
		λ	λ	ϵ_λ	λ	ϵ_λ	λ	ϵ_λ
A01	t1	0.0254	0.0274	8.01	0.0255	0.39	0.0284	11.88
	t2	0.0149	0.0145	−2.59	0.0133	−10.51	0.0167	12.49
A02	t1	0.0229	0.0253	10.24	0.0225	−1.87	0.0260	13.29
	t2	0.0154	0.0140	−8.66	0.0130	−15.30	0.0150	−2.08
A03	t1	0.0225	0.0260	15.18	0.0232	2.73	0.0263	16.43
	t2	0.0151	0.0144	−4.86	0.0128	−14.93	0.0161	6.88
A04	t1	0.0251	0.0274	8.84	0.0254	0.98	0.0284	13.06
	t2	0.0150	0.0146	−3.16	0.0134	−11.15	0.0168	11.65
A05	t1	0.0259	0.0284	9.64	0.0258	−0.21	0.0292	12.96
	t2	0.0208	0.0202	−2.76	0.0184	−11.37	0.0234	12.60
A06	t1	0.0461	0.0530	14.82	0.0482	4.51	0.0553	19.93
	t2	0.0310	0.0294	−5.04	0.0267	−13.87	0.0338	9.01
A07	t1	0.0324†	0.0312	−3.71	0.0297	−8.38	0.0350	8.06
	t2	0.0100†	0.0100	−0.04	0.0100	−0.04	0.0100	−0.04
A08	t1	0.0722†	0.0671	−6.99	0.0700	−2.99	0.0760	5.29
	t2	0.0100†	0.0100	−0.03	0.0100	−0.03	0.0100	−0.03
A09	t1	0.0669	0.0786	17.48	0.0724	8.13	0.0896	33.82
	t2	0.0622	0.0589	−5.32	0.0518	−16.74	0.0679	9.08
A10	t1	0.0740	0.0874	18.19	0.0779	5.36	0.0953	28.88
	t2	0.0638	0.0582	−8.77	0.0529	−17.05	0.0619	−2.97

Table 5.1: The 'A' series of test cases from [152]. The model, corresponding to case 'A01' is shown in Figure 3.2.

Case	Task	Throughput						
		Exact	LQNS		SRVN		MOL	
		λ	λ	ϵ_λ	λ	ϵ_λ	λ	ϵ_λ
B01	t1	0.0245	0.0273	11.57	0.0246	0.39	0.0279	13.92
	t2	0.0149	0.0146	−2.52	0.0132	−11.36	0.0167	11.66
B02	t1	0.0217	0.0247	13.92	0.0213	−1.51	0.0252	16.45
	t2	0.0154	0.0141	−8.73	0.0129	−16.37	0.0150	−2.64
B03	t1	0.0212	0.0253	19.37	0.0219	3.00	0.0255	20.19
	t2	0.0153	0.0145	−4.83	0.0128	−16.05	0.0162	6.38
B04	t1	0.0242	0.0272	12.37	0.0244	0.79	0.0278	14.95
	t2	0.0150	0.0147	−2.62	0.0133	−11.69	0.0168	11.37
B05	t1	0.0249	0.0282	13.49	0.0249	−0.05	0.0287	15.21
	t2	0.0210	0.0203	−3.15	0.0184	−12.51	0.0234	11.35
B06	t1	0.0427	0.0491	14.76	0.0410	−3.98	0.0499	16.70
	t2	0.0298	0.0299	0.34	0.0262	−12.26	0.0336	12.85
B07	t1	0.0317†	0.0311	−2.06	0.0284	−10.57	0.0342	7.64
	t2	0.0100†	0.0100	−0.07	0.0100	−0.07	0.0100	−0.07
B08	t1	0.0648†	0.0611	−5.73	0.0561	−13.47	0.0654	0.87
	t2	0.0100†	0.0100	−0.05	0.0100	−0.05	0.0100	−0.05
B09	t1	0.0582	0.0608	4.40	0.0518	−11.14	0.0651	11.76
	t2	0.0494	0.0582	17.71	0.0478	−3.29	0.0659	33.28
B10	t1	0.0643	0.0642	−0.15	0.0531	−17.46	0.0663	3.05
	t2	0.0535	0.0553	3.33	0.0469	−12.33	0.0579	8.28

Table 5.2: The 'B' series of test cases from [152]. The model, corresponding to case 'A01' is shown in Figure 3.2.

Metric	LQNS	SRVN	MOL
Mean	3.96	−6.05	11.56
Std. Dev.	1.03	0.69	1.21
Min	−6.14	−15.13	0.75
Max	10.51	−2.63	21.64

Table 5.3: Summary of Tables 5.1 and 5.2.

5.5 Solver Construction

Figure 5.14 shows the class diagram of the fixed-rate multi-phased server classes. It is an extension of the class diagram shown earlier in Figure 3.23.

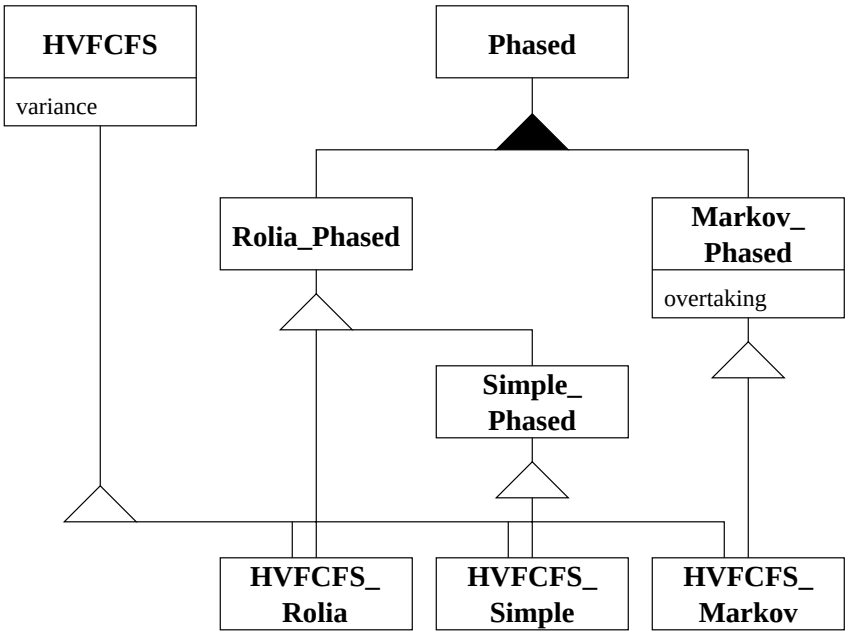


Figure 5.14: Class hierarchy for phased fixed-rate servers.

Two branches in the class hierarchy are present. The left branch (*Rolia_Phased*) implements the simpler client-based overtaking approximation. Its classes are described below in §5.5.1. The right branch (*Markov_Phased*) implements the new client-server-based overtaking approximation. Its classes are described below in §5.5.2. The notation for the expressions is described in the Glossary on page xxi.

Class Phased_Server

Class *Phased_Server* is an abstract superclass that defines the waiting time expression for an open model. The expression used in LQNS is for an M/G/1 queueing system with forced

idle periods [73, (3.87)], shown below (5.18).

$$\begin{aligned} W_{me0} &= s_{me01} + \frac{\lambda_{m0}\overline{S^2_{m0}}}{2(1-\rho_m)} \\ \overline{S^2_{m0}} &= \frac{\sum_{e=1}^{E_m} \lambda_{me0} s^2_{me0}}{\sum_{e=1}^{E_m} \lambda_{me0}} \end{aligned} \tag{5.18}$$

The term s_{me01} is the phase 1 service time of the open class at entry e of station m.

For closed models, the waiting time expressions are all defined by subclasses.

5.5.1 Server-Based Overtaking

Equation (5.19) is used to find the overtaking probabilites for the classes described in this section. $, _{mek}(\mathbf{n}-\mathbf{e}_k)$ is the overtaking probability defined earlier in §5.2.2 on page 103.

$$\begin{aligned} , _{mek}(\mathbf{n}) &= \frac{v_{mek}s_{mek2}}{v_{mek}s_{mek2} + z_k + R_k(\mathbf{n}) - R_{mk}(\mathbf{n})} \\ R_k(\mathbf{n}) &= \sum_{m=1}^{M} R_{mk}(\mathbf{n}) \\ R_{mk}(\mathbf{n}) &= \sum_{e=1}^{E_m} v_{mek} W_{mek}(\mathbf{n}) \end{aligned} \tag{5.19}$$

Class Rolia_Phased_Server

This class implements a simplified version of the two-phase server described in [113]. It does not include the effects of variance nor does it include the correction for utilization described in §5.3.1.

$$W_{mekp}(\mathbf{n}) = s_{mek1} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n}-\mathbf{e}_k) + s_{mek2}\, , _{mek}(\mathbf{n}-\mathbf{e}_k) \tag{5.20}$$

Class HVFCFS_Rolia_Phased_Server

This class implements the two-phase server described in [113]. The residual term, r_{mij}, is inherited from class *HVFCFS_Server* and is found using (3.21) on page 3.21. It does not include the correction for forked customers described in §5.3.1.

$$\begin{aligned} W_{mek}(\mathbf{n}) &= s_{mek} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} Q_{mij}(\mathbf{n}-\mathbf{e}_k) + \sum_{j=1}^{K}\sum_{i=1}^{E_m} r_{mij} U_{mij}(\mathbf{n}-\mathbf{e}_k) \\ &\quad + s_{mek2}\,,\ {}_{mek}(\mathbf{n}-\mathbf{e}_k) \end{aligned} \tag{5.21}$$

Class Simple_Phased_Server

This class uses the simpler server-based overtaking, but compensates for the forked customers using the method outlined earlier in §5.3.1. It is used for the Processor Sharing queueing discipline and other cases when variance is not needed.

$$\begin{aligned} W_{mekp}(\mathbf{n}) &= s_{mek1} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n}-\mathbf{e}_k) + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij2}\,,\ {}_{mij}(\mathbf{n}-\mathbf{e}_k) \\ &\quad + \sum_{j=1}^{K}\sum_{i=1}^{E_m}(1 - \,,\ {}_{mij}(\mathbf{n}-\mathbf{e}_k)) \left[s_{mij1} + \frac{s_{mij2}^2}{s_{mij}} \right] U_{mij}(\mathbf{n}-\mathbf{e}_k) \end{aligned} \tag{5.22}$$

Class HVFCFS_Simple_Phased_Server

This class uses the simpler server-based overtaking approximation for servers that use FIFO queueing.

$$W_{mek}(\mathbf{n}) = s_{mek} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} Q_{mij}(\mathbf{n}-\mathbf{e}_k) + \sum_{j=1}^{K}\sum_{i=1}^{E_m} r_{mij} U_{mij}(\mathbf{n}-\mathbf{e}_k)$$

$$+\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij2},_{mij}(\mathbf{n}-\mathbf{e}_k)$$
$$+\sum_{j=1}^{K}\sum_{i=1}^{E_m}(1-,_{mij}(\mathbf{n}-\mathbf{e}_k))\left[s_{mij1}+\frac{s_{mij2}^2}{s_{mij}}\right]U_{mij}(\mathbf{n}-\mathbf{e}_k) \quad (5.23)$$

5.5.2 Client-Server-Based Overtaking

Equation (5.24) is used to find the overtaking probabilities for the classes described in this section.

$$,_{mek}(p) = \Pr\{\text{overtake}(e,k,p)\} \quad (5.24)$$

Class Markov_Phased_Server

This class assumes that all phases are exponentially distributed.

$$\begin{aligned} W_{mekp}(\mathbf{n}) &= s_{mek1}+\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij}L_{mij}(\mathbf{n}-\mathbf{e}_k) \\ &\quad +\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij2},_{mij}(\mathbf{n}-\mathbf{e}_k) \\ &\quad +\sum_{j=1}^{K}\sum_{i=1}^{E_m}(1-,_{mij}(\mathbf{n}-\mathbf{e}_k))\left[s_{mij1}+\frac{s_{mij2}^2}{s_{mij}}\right]U_{mij}(\mathbf{n}-\mathbf{e}_k) \end{aligned} \quad (5.25)$$

Class HVFCFS_Markov_Phased_Server

This class implements that Reiser approximation to account for variance at the server.

$$\begin{aligned} W_{mek}(\mathbf{n}) &= s_{mek}+\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij}Q_{mij}(\mathbf{n}-\mathbf{e}_k)+\sum_{j=1}^{K}\sum_{i=1}^{E_m} r_{mij}U_{mij}(\mathbf{n}-\mathbf{e}_k) \\ &\quad +\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij2},_{mij}(\mathbf{n}-\mathbf{e}_k) \end{aligned}$$

$$+\sum_{j=1}^{K}\sum_{i=1}^{E_m}(1-,_{mij}(\mathbf{n}-\mathbf{e}_k))\left[s_{mij1}+\frac{s_{mij2}^2}{s_{mij}}\right]U_{mij}(\mathbf{n}-\mathbf{e}_k) \qquad (5.26)$$

5.6 Conclusions

Early replies give performance gains by unblocking a client earlier. However at a saturated server the client loses this advantage by having to queue for longer. The examples given here, though limited, show that early replies give worthwhile gains for servers which are not close to saturation. The improvement in response time is close to the ideal value at low server utilization. The number of clients competing for the server appears to have little effect on the improvement.

Analytic modeling of early replies requires careful approximation of the queueing effects, since exact queueing solutions do not apply. Previous approximations were oversimplified and sometimes give large errors, in the set of test cases used here. The new approximation accounts for two aspects of the client behaviour that were ignored before. First, it represents the difference between client phases. Second, it also models the client delay before its next request as a sum of a number of exponential delays, instead of by one exponential delay with the same mean. It also adjusts the queue-length calculation for the early departures.

The new approximation has errors which are always smaller, and are up to an order of magnitude smaller in relative terms, in the worst cases. The worst error in accuracy of the new approximation is about 10%, and occurs for a saturated server that has about half the service in the second phase.

The examples were chosen to have large errors and to stress the approximation. The impact of the approximation on an entire performance model is usually watered down by other important delays such as client think time and network latency, shown in Table 5.3.

Chapter 6

Multiservers

Multithreaded tasks, which are common and neccesary in distributed and parallel computing are naturally modelled as multiserves. The most common software architecture uses remote procedure calls to a variety of servers to execute the bulk of the processing. This is seen in transaction processing [60, 128], in business client-server systems [37, 130], and in the various versions of Open Distributed Processing such as OSI-ODP [153], DCE [36], and CORBA [93]. In all these areas it has been found to be imperative, for performance reasons, to have multi-threaded servers. Thus DCE and CORBA have defined their own thread packages, and most operating systems have introduced thread support.

Multi-threading is required for several reasons. First, a service may block to wait for some other operation such as a disk I/O to complete, and another thread (or even several threads) may be able to execute in this interval. Second, with multiple threads the server can exploit a multiprocessor for additional processor capacity (used extensively in web servers and file servers, for instance). Third, a single request can spawn parallel threads to manage parallel subtransactions. Without multithreading a server may have a long queue and be idle (blocked), and yet be unable to take another request. These requirements lead to a standard pattern where a task has a "listener" thread which dequeues requests, and a pool

of identical "worker" threads which each operate on one request [74]. The pool of threads may have a fixed size or a new thread may be created for every request, which provides an *infinite server* pool.

The analysis of multi-threaded software servers is complicated by some aspects of the usual distributed systems. The service time or holding time of a thread depends on the blocking times it may experience for disk I/O or other services, (i.e. *nested services*); these blocking times may in turn depend on contention at the nested server. Servers often defer some of their work until after replying to the request, such as delayed writes or buffer clean-up (*second-phase* service). Servers typically offer different classes of service, with quite different operations and service times. These factors all put software multiservers well outside the descriptions of servers for which elementary queueing solutions can be used to predict performance.

There has been little research reported on software multiservers. Neilson et. al. discussed their importance in certain simple situations, and gave a "bottleneck strength" measure for a server, to identify cases in which the threading level could usefully be increased [89]. They included multi-tier systems. Rolia and Sevcik described a queueing approximation in [110], which will be examined below. Dilley et. al. applied it to web servers in [30] (and shown in Figure 6.1), where long latencies in the network could block a thread; they found that the number of threads made an enormous difference. Concern about the performance effect of threading levels has been expressed in some papers, and experimental and simulation results have been quoted to show their effectiveness [80, 50, 52, 61].

This chapter evaluates seven approximations for multiservers for accuracy and speed, when used in layered models. A robust approximation must be sufficiently accurate in a large number of cases covering a great variety of systems, and this work has examined a large number of randomly constructed cases, as well as a number of systems gathered from the literature.

This chapter also demonstrates the performance improvements obtained by introducing

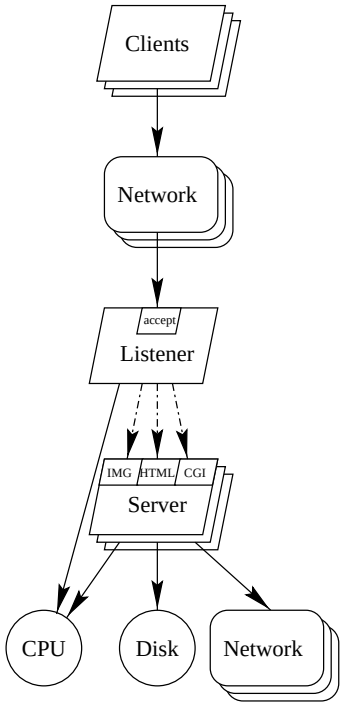


Figure 6.1: Layered Queueing Network Model used to examine the effect of threading a web server [30].

second phases to multithreaded tasks. The intent is to reduce the blocking time of a client at a multithreaded server. Some of the Mean Value Analysis expressions described later in this paper for multiclass first-come first-served multiservers have been modified to account for this behaviour. An example based loosely on a model in [86] is used to demonstrate the execution time enhancement brought on by multiple phases. Other examples show the performance of the solution and its accuracy. Second phase service in multiserver queues has not been modelled before this.

The material in this chapter was mostly published in [39].

6.1 Performance Implications of Multiple Layers

Figure 6.2 shows a simple layered queueing model, similar to the Web server example shown in Figure 6.1, showing multiple clients, a multithreaded application and two lower-level servers. The networking delay component has been dropped, but lower-level servers which represent databases and other value-added services have been added. It might represent an electronic commerce server.

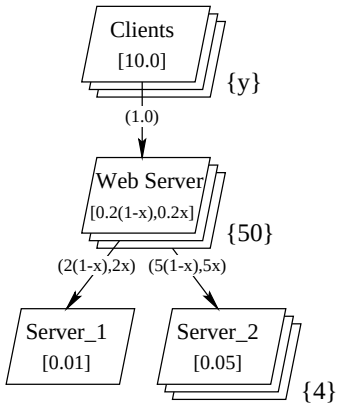


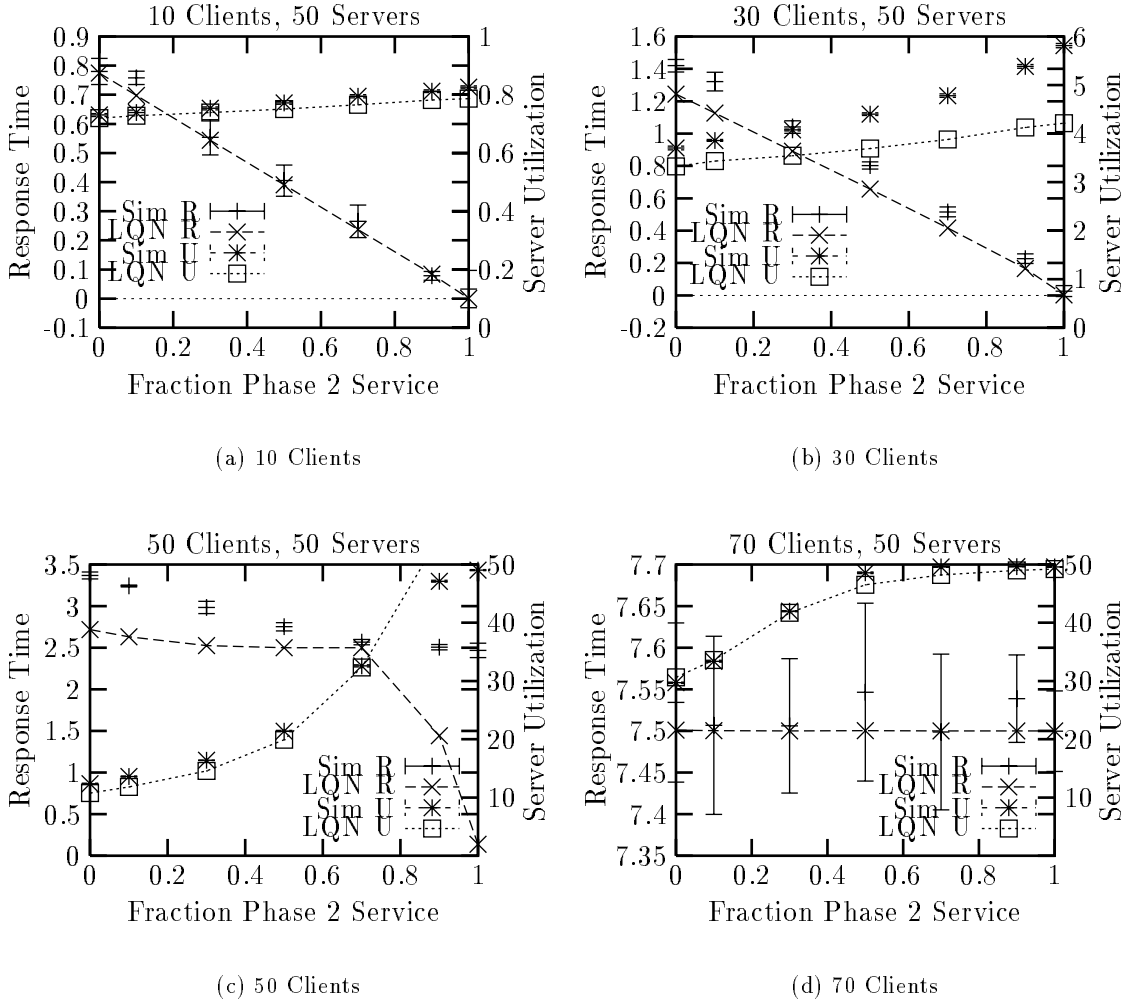
Figure 6.2: A simplified layered queueing network model of web server connected to two lower-level database servers. The value x represents the fraction of phase-two service and y represents the number of concurrently active customers.

From a performance standpoint, the important aspect of service requests is that a client is blocked while a server processes its requests. Unacceptable delays may result at an upper-level client if the intermediate-level applications are single threaded as they also may be blocked waiting for replies from their own lower-level servers. By allowing for multiple concurrent threads at intermediate servers, additional services for other clients may be initiated while the server is waiting for responses from lower level servers [89, 30, 77].

A second technique to improve response time to clients is to split execution at lower-level servers into two phases, in Figure 1.1(b). Figure 6.3 shows the performance effects of exploiting a second phase in the model of Figure 6.2. The number of concurrent users, y, and the fraction of phase two service time, x, is varied. When the number of available threads exceeds the number of potential customers, two-phase multiservers can offer substantial performance benefits. However, when heavy queueing occurs at the two-phase multiserver, as in Figure 6.3(d), second phase service offers little to no performance improvement. This result is similar to the conclusion for second phases in a single server described in the previous chapter.

6.2 Multiclass Multiservers

Submodels that arise from layered queueing networks often have sets of clients that make different demands on the servers (represented by entries on the servers). Further, software servers usually have first-come, first-served (FCFS) queueing disciplines. The corresponding queueing network model must therefore have FCFS service centers with service demands that vary by chain. Queueing networks with these characteristics do not have a "product form" [11, 91]. Fortunately, approximations exist for multiclass FCFS single- and multiple-servers which are sufficiently accurate for practical purposes. The following section describes seven multiclass multiserver waiting time expressions and compares their accuracy and execution time to simulation. The notation used is shown in Table 1.



(a) 10 Clients

(b) 30 Clients

(c) 50 Clients

(d) 70 Clients

Figure 6.3: Impact of moving operations into the second phase on the client response time and Web Server utilization versus fraction of phase 2 service time for the model in Figure 6.2. The phase 2 fraction and the number of clients correspond to x and y respectively. Simulations were conducted with 95% confidence intervals of $\pm 1\%$. The analytic solutions were performed using (6.9).

6.2.1 MVA Waiting Time Expressions

The approximations listed below all give different expressions for waiting time W_{mk} for a class k customer at queue m which is a J_m-fold multiservers. Many of the expressions rely on the marginal probability of finding i customers in service, $P_m(i, \mathbf{N} - \mathbf{e}_k)$, and the probability of finding all servers busy, $PB_m(\mathbf{N} - \mathbf{e}_k$.

- Reiser and Lavenberg [107] gave the exact MVA expression for a multiserver based on special product-form networks with multiclass multiservers:

$$W_{mk}(\mathbf{N}) = s_{mk} + \frac{s_{mk}}{J_m}\left[\sum_{j=1}^{K} Q_{mj}(\mathbf{N} - \mathbf{e}_k) + PB_m(\mathbf{N} - \mathbf{e}_k)\right] \tag{6.1}$$

  where the first term is the customer's own service, the second term is the time the customer spends in the queue waiting for service and the third term is the busy probability. For FCFS scheduling, the service time terms s_{mk} must be equal for each class. Equation (6.1) is often expressed as

$$W_{mk}(\mathbf{N}) = \frac{s_{mk}}{J_m}\left[1 + \sum_{j=1}^{K} L_{mj}(\mathbf{N} - \mathbf{e}_k) + \sum_{i=0}^{J_m-2}(J_m - 1 - i)P_m(i, \mathbf{N} - \mathbf{e}_k)\right] \tag{6.2}$$

- Ruth [119] evaluated a version of (6.2) where each of the individual terms is multiplied by the service time. The class dependent service times are used for the customers in queue (the first summation) and the overall mean service time $\overline{s_m}$ is used for the customers in service (the second summation).

$$W_{mk}(\mathbf{N}) = \frac{1}{J_m}\left[s_{mk} + \sum_{j=1}^{K} s_{mj}L_{mj}(\mathbf{N} - \mathbf{e}_k) + \overline{S_m}\sum_{i=0}^{J_m-2}(J_m - 1 - i)P_m(i, \mathbf{N} - \mathbf{e}_k)\right] \tag{6.3}$$

- de Souza e Silva and Muntz [29] incorporated class-dependent FCFS scheduling into multiservers by applying the approximation by Reiser for load-independent servers to

(6.1) [108]. This approximation was incorporated into Linearizer approximate MVA with good results [24]. The complete waiting time expression, describing the terms xe and xr, is given in §6.6.1.

$$W_{mk}(\mathbf{N}) = s_{mk} + \sum_{j=1}^{K} xe_{kcj}(\mathbf{N}) Q_{mj}(\mathbf{N} - \mathbf{e}_k) + PB_m(\mathbf{N} - \mathbf{e}_k) xr_{mk}(\mathbf{N}) \tag{6.4}$$

- Bruell, Baldo and Afshari [7] list several well-known expressions used for mixed, multiple class BCMP networks with load dependent service stations, including

$$W_{mk}(\mathbf{N}) = \sum_{\mathbf{n} \in \mathbf{N}} \frac{|\mathbf{n}| s_{mk}}{\alpha_m(\mathbf{n})} P_m(\mathbf{n} - \mathbf{e}_k, \mathbf{N} - \mathbf{e}_k) \tag{6.5}$$

  which has class-dependent service times. Restrictions exist on the rate multiplier terms, $\alpha_m(\mathbf{n})$. Further, the queueing discipline, called *composite queuing*, is not FCFS. The marginal probabilities P_m are computed for every customer population $\mathbf{n}$.

- Schmidt [124] also uses the marginal probabilities based on customer population but approximates FCFS queueing.

$$W_{mk}(\mathbf{N}) = \sum_{\mathbf{n} \in \mathbf{N}} \left[s_{mk} + \frac{\max(0, |\mathbf{n}| - M)}{M(|\mathbf{n}| - 1)} \left(\sum_{j=1}^{K} \frac{n_c V_{mj} s_{mj} - V_{mk} s_{mk}}{V_{mk}} \right) \right] P_m(\mathbf{n} - \mathbf{e}_k, \mathbf{N} - \mathbf{e}_k) \tag{6.6}$$

- Rolia [110] has developed a multiserver approximation for the Method of Layers that does not require computation of marginal probabilities at all:

$$W_{mk}(\mathbf{N}) = s_{mk} \left[1 + \frac{U_m^{(1)}(\mathbf{N} - \mathbf{e}_k)^M}{J_m} \sum_{j=1}^{K} L_{mj}(\mathbf{N} - \mathbf{e}_k) \right] \tag{6.7}$$

- Equation (6.7) is modified here by multiplying the terms inside the brackets by the class-dependent service times, to better represent the delays caused by customers of

each class:

$$W_{mk}(\mathbf{N}) = s_{mk} + \frac{U_m^{(1)}(\mathbf{N}-\mathbf{e}_k)^M}{J_m} \sum_{j=1}^{K} s_{mj} L_{mj}(\mathbf{N}-\mathbf{e}_k) \tag{6.8}$$

Equations (6.2) through (6.6) all use marginal probabilities, either for the total number of customers at the station, $P_m(i, \mathbf{N}-\mathbf{e}_k)$, or for each population vector, $P_m(\mathbf{n}-\mathbf{e}_k, \mathbf{N}-\mathbf{e}_k)$. When using exact MVA to solve a queuing network, these terms are all computed recursively starting with zero customers, and with $P_m(0, \mathbf{0}) = 1$ and $P_m(\mathbf{0}, \mathbf{0}) = 1$. When using approximate MVA, the usual backwards MVA approximation step to find the value of $P_m(i, \mathbf{N}-\mathbf{e}_k)$ is to simply let $P_m(i, \mathbf{N}-\mathbf{e}_k) = P_m(i, \mathbf{N}-\mathbf{e}_k)$. This assignment often leads to infeasible marginal probabilities. Better approximations, given by Krzesinski and Greyling [70] for $P_m(i, \mathbf{N}-\mathbf{e}_k)$, and by Schmidt [124] for $P_m(\mathbf{n}-\mathbf{e}_k, \mathbf{N}-\mathbf{e}_k)$ have been used in this work. The exact and approximate expressions are shown in Appendix B.

6.2.2 Accuracy and Performance Comparisons

Figure 6.4 shows the layered queueing network, based on the model in [23], used to evaluate the accuracy and run-time performance of each of the residence time expressions shown above. Each server gives four classes of service to each of the four classes of user. Service times at each of the lower-level servers were chosen randomly from a range of values between 1.57 and 72.39. The number of visits to each of the servers by each class of customer was varied from 0.35 to 2.30. Two hundred different sets of parameters were tested.

Table 6.1 shows the percent Mean Relative Error (MRE) in throughput at the clients, MRE = $\frac{\text{apprx}-\text{sim}}{\text{sim}} \times 100$, the standard deviation in MRE, the percent mean Absolute Relative Error (ARE) in throughput at the clients, ARE = $\frac{|\text{apprx}-\text{sim}|}{\text{sim}} \times 100$, and the average run time for the test cases. Equations (6.2), (6.5) and (6.7) all give large errors because they do not represent FCFS queueing with chain-dependent service times. Based on the MRE metric, (6.8) performs the best, and based on ARE, (6.4) is actually the best. The MRE error for (6.4) is slightly larger because it gives a slight bias, underestimating throughput.

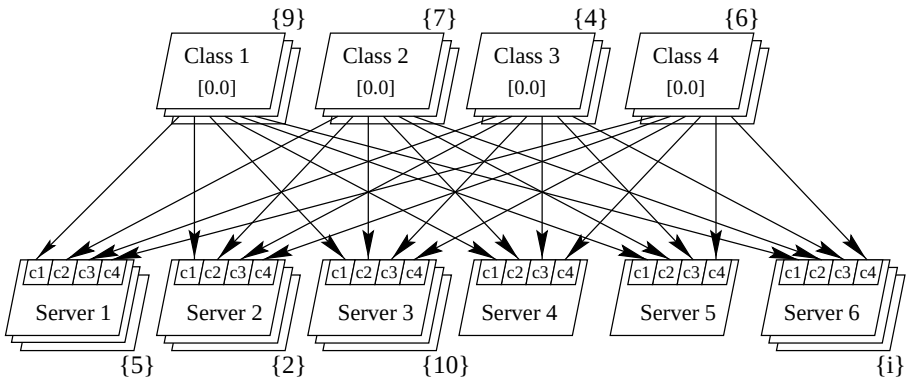


Figure 6.4: Test Network

Of these two, Equation (6.8) is clearly superior in run time because it does not need to compute marginal probabilities nor does it have multiple product terms like those used by the de Souza e Silva and Muntz approximation. Equation (6.8) is a good compromise of accuracy and run-time.

Expression	MRE	σ	ARE	Run Time
Simulation				42:37.9
Reiser (6.2)	11.18	0.41	13.91	2.0
Ruth (6.3)	-1.68	0.01	3.52	2.2
de Souza e Silva (6.4)	-0.59	0.00	1.55	1:25.9
Bruell (6.5)	11.27	0.42	15.07	2:55.4
Schmidt (6.6)	-3.64	0.07	8.51	4:38.5
Rolia (6.7)	11.99	0.38	14.16	2.1
Franks (6.8)	0.48	0.01	2.91	2.3

Table 6.1: Mean Relative Error and mean Absolute Relative Error of client throughput when compared to simulation. All of the simulations have 95% confidence intervals of $\pm$ 1.0%.

6.3 Multiphase Multiclass Multiservers

This section describes the impact of second-phase service on the calculations for multiclass multiservers which were explored for simple load-independent servers in [151, 110]. These servers need new waiting time expressions to compensate for two factors. First, the client is not held for the entire period of execution of its request at the server, shown in Figure 5.1(a). Second, a client making a subsequent request to a server may block while the server completes its own preceding request. This phenomenon is called *overtaking* as discussed in Chapter 5.

The next section describes the waiting time expressions for multiphase, multiclass multiservers. Three approximations were selected based on the accurarcy and solution run time speed shown in Table 6.1: Ruth (6.3), de Souza e Silva (6.4) and Franks (6.8).

6.3.1 MVA Waiting Time Expressions

The delay to a customer at a multiphase, multiclass multiserver has the following components:

1. its own phase 1 service time, s_{mk1},
2. time spent waiting in the queue (effective backlog), and
3. time spent waiting for a server to become free (residual service).

The new version of waiting time will be found for the de Souza e Silva approximation (6.4) first since it is the most detailed of the expressions that are suitable for multiple classes. The first term of (6.4), s_{mk}, is changed to s_{mk1}, the phase one service time. The second term of (6.4), $\sum_{j=1}^{K} xe_{kcj}(\mathbf{N})Q_{mj}(\mathbf{N}-\mathbf{e}_k)$, is the delay arriving customers see prior to being serviced. With a multiphase server, servers in phase two are equivalent to "new" customers that are created when replies to clients are sent. New arrivals must also wait behind these

pseudo customers so only the phase one utilization, U_{mk1}, is subtracted. The third term of (6.4), the departure time, includes both phases of service, so it is unchanged.

The second effect that must be taken into consideration is overtaking, described earlier in Chapter 5.

After these changes, Equation (6.4) becomes:

$$\begin{aligned} W_{mk}(\mathbf{N}) &= s_{mk1} + \sum_{j=1}^{K} xe_{kcj}(\mathbf{N})[Q_{mj}(\mathbf{N}-\mathbf{e}_k) + U_{cj2}(\mathbf{N}-\mathbf{e}_k)] \\ &\quad + PB_m(\mathbf{N}-\mathbf{e}_k)xr_{mk}(\mathbf{N}) + \frac{\Pr\{OT\}_{mk}}{J_m} \cdot s_{mk2} \end{aligned} \tag{6.9}$$

Equations (6.3) and (6.8) are modified in a similar fashion:

$$\begin{aligned} W_{mk}(\mathbf{N}) &= \frac{1}{J_m}\left[s_{mk1} + \sum_{j=1}^{K} s_{mj}\left[L_{mj}(\mathbf{N}-\mathbf{e}_k) + U_{cj2}(\mathbf{N}-\mathbf{e}_k)\right] \right. \\ &\quad \left. + \overline{s_m} \sum_{i=0}^{J_m-2} (J_m - 1 - i)P_m(i, \mathbf{N}-\mathbf{e}_k) + \frac{\Pr\{OT\}_{mk}}{J_m} \cdot s_{mk2} \right] \end{aligned} \tag{6.10}$$

$$\begin{aligned} W_{mk}(\mathbf{N}) &= s_{mk1} + \frac{U_m^{(1)}(\mathbf{N}-\mathbf{e}_k)^M}{J_m} \sum_{j=1}^{K} s_{mj}\left[L_{mj}(\mathbf{N}-\mathbf{e}_k) + U_{cj2}(\mathbf{N}-\mathbf{e}_k)\right] \\ &\quad + \frac{\Pr\{OT\}_{mk}}{J_m} \cdot s_{mk2} \end{aligned} \tag{6.11}$$

6.3.2 Accuracy and Performance Comparisons for Multiphase Servers

The layered queueing network of Figure 6.4 was again used to test the accuracy and run-time performance of Equations (6.9), (6.11) and (6.10) with different second phase loads. Service times for each phase of service at each of the lower-level servers were chosen randomly from a range of values between 1.275 and 36.195. Two hundred different networks were tested with the fraction of phase two varying from 0.04 to 0.96. Another two hundred test cases were run with phase two fractions fixed at 0.25, 0.50, 0.75 and 1.00 for fifty cases each. The number of visits to each of the servers was varied from 0.35 to 2.30. The results of the runs

are shown in Table 6.2.

Expression	%Ph2	MRE	σ	ARE	Run Time
Simulation					42:44.5
de Souza e Silva (6.9)	rand	-0.66	0.00	1.34	7:05.1
Ruth (6.10)		0.10	0.07	5.09	15.0
Franks (6.11)		0.96	0.07	4.92	5.9
de Souza e Silva (6.9)	25	-0.80	0.03	1.59	3:43.0
Ruth (6.10)		-0.59	0.14	4.08	5.6
Franks (6.11)		0.60	0.18	4.41	8.8
de Souza e Silva (6.9)	50	-0.83	0.03	1.62	3:43.5
Ruth (6.10)		0.93	0.17	4.99	5.1
Franks (6.11)		0.64	0.20	4.67	6.1
de Souza e Silva (6.9)	75	-0.84	0.02	1.66	3:47.3
Ruth (6.10)		1.87	0.48	6.30	5.4
Franks (6.11)		1.12	0.36	5.33	6.2
de Souza e Silva (6.9)	100	-0.79	0.03	1.77	4:22.0
Ruth (6.10)		3.62	0.53	7.24	12.0
Franks (6.11)		0.94	0.54	5.88	4.4

Table 6.2: Performance Results for two phase multiclass multiservers. Simulations were conducted with 95% confidence intervals of $\pm 1\%$.

As with the single phase test cases in Table 6.1, based on the ARE metric, (6.9) is more accurate that either (6.10) or (6.11), but much more computationally expensive. Run times for the analytic solutions are also higher than the models with only single phase multiserver because multiple iterations are required – the overtaking probabilities depend on the waiting times at the lower level servers, hence may change after a submodel is solved.

Equation (6.11), the equation originated in this thesis, is a strategic compromise of accuracy and effort.

6.4 Industrial Example 1: Telephone Inquiry System

This section describes two different tele-operator models, derived loosely on the client-server tele-marketing model presented by Menascé [86]. In each case there is a set of operators

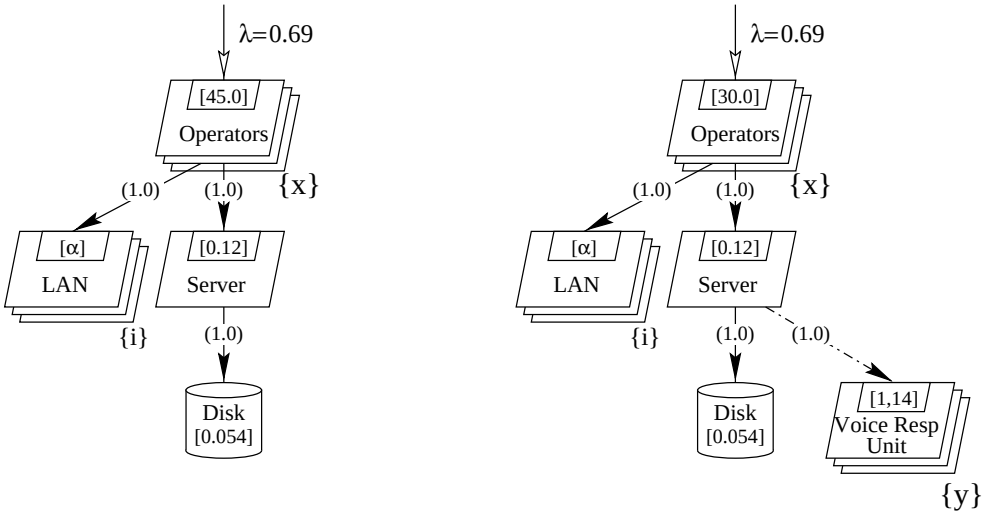
who field calls arriving at a rate of 2,500 calls per hour. In the first system, each operator handles all aspects of a call and takes, on average, 45 seconds to do so. In the second system, the operator processes the first phase of the call for 30 seconds, then hands off the remaining 15 seconds of service to an automated voice response unit. The voice response unit takes one second to accept the operation from the operator; the remaining 14 seconds are processed asynchronously in second phase. In both cases, the operators interact with a database server connected to a LAN. The delay imposed by the LAN is dependent on the number of operators due to collisions. The delay requirement for the system is that the average wait for an operator should be five seconds or less.

The layered queueing network models for the operators-only and voice response unit configurations are shown in Figures 6.5(a) and 6.5(b) respectively. Service times for each of the tasks and devices, except for the LAN, are shown in the figure. The service time for the LAN, shown as α, depends on the number of operators present, x, and was taken from the parameters in [86]. The number of operators present was varied to find a configuration where the queueing time for a customer was no more than five seconds.

The voice response unit model augments the Operators Only model by adding a two-phase multiserver task to deliver responses. The database server *forwards* requests to the voice response unit multiserver which takes one second of service time before replying directly to the originating operator. The voice response unit then takes an additional fourteen seconds to respond to the customer's request.

Figure 6.6 shows the performance results for the Operators Only system in Figure 6.5(a). From the graph, 35 operators are needed to meet the performance requirements.

Figure 6.7 shows performance results as waiting time contours, for the Voice Response Unit system. Both simulation and the analytic LQN solution shows the minimum number of operators needed is 25. The models differ slightly in the number of voice response units needed – the analytic solution is somewhat optimistic in estimating the need for 14 voice response units while the simulation results estimate the need for 15.



(a) Operators Only

(b) With Voice Response Units

Figure 6.5: Telephone Operators models. The values of x (the number of operators) and y (the number of voice response units in (b)) need to be found so that customer requests arriving at a rate of $\lambda = 0.69$ wait no longer than five seconds on average before being serviced. α is a parameter that depends on the number of operators, x.

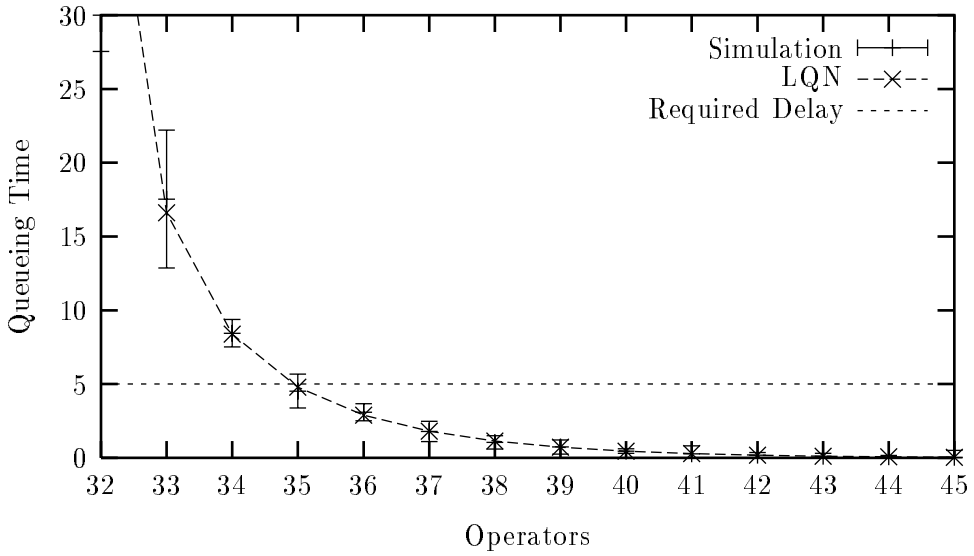
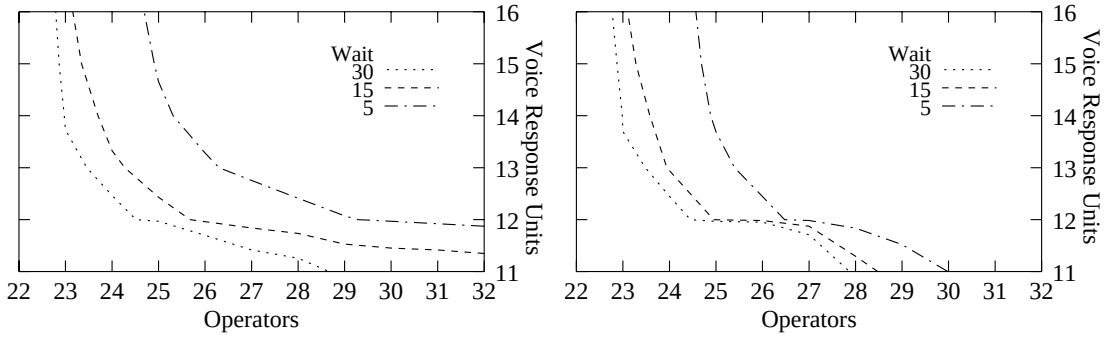


Figure 6.6: Waiting time for the Operators Only model shown in Figure 6.5(a). Simulations were conducted with 95% confidence intervals of $\pm 1\%$.

Both the simulation and the analytic model have difficulty with nearly-saturated situations (e.g., with 11 voice response units). The simulation becomes unstable and takes a long time to give adequate accuracy. The analytic model also becomes unstable due to infeasible intermediate solutions with infinite queues, during intermediate iterations of the solver.

The figures show that the analytic solution is sufficiently accurate for practical purposes. Further, for the Operators Only case, the simulations took on average, 41 minutes to complete, whereas the analytic solutions took less than 1 second each. For the voice response unit model, the simulations took over 75 minutes while the analytic solutions took approximately 8 seconds each.



(a) Simulation (b) Analytic

Figure 6.7: Waiting time contours for the Voice Response Unit system shown in Figure 6.5(b). Simulations were conducted with 95% confidence intervals of ±1%.

6.5 Industrial Example 2: Transaction Processing System

To demonstrate how the calculations scale up to a system with many multiservers, many layers and many customers, they will be applied to a moderately large transaction processing system.

The example shown in Figure 6.8 represents a system designed to manage the installation, operational status and repair of equipment in a large communications system and is loosely based on the system analyzed in [54]. The system is only partially represented with the transactions *addItem* and *delItem*; these are sufficient to demonstrate the scalability of the layered queueing network solution technique to a large system.

The design was targeted to a Tandem NonStop™ multiprocessor system [31]. Task are statically allocated to processors. Redundancy features of the target system are not modelled, however, the overhead needed to update shadow tasks is included. The performance of the system is therefore calculated for a period without faults.

The system was evaluated with from 1 to 250 users who think for ten seconds each, with the parameters shown in Figure 6.8. The response time to users, found both analytically

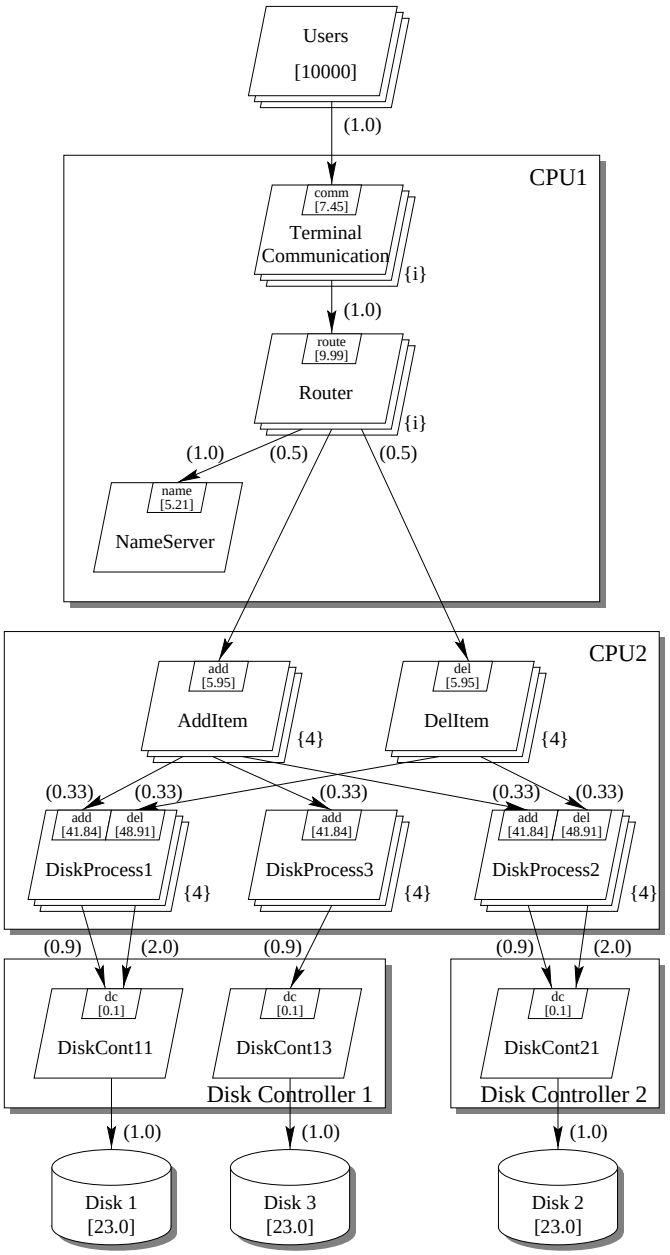


Figure 6.8: Transaction Processing System Model

and through simulation, is shown in Figure 6.9(a).

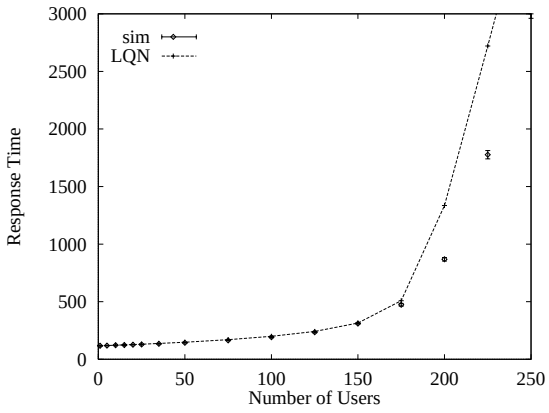
The overall accuracy of the analytic solution of layered queueing network models depends on the number of submodels used, as errors at lower layers tend to be magnified as they percolate upwards. The problem is worse when as tasks at intermediate layers approach saturation because small changes in their service times arising from errors in lower-level submodels result in large changes in waiting times to tasks in higher-level submodels.

Figure 6.9(b) shows the solution time of the layered queueing network solver using the multiserver expressions of (6.4) and (6.8). The solution accuracy of the two analytic multiserver versions is almost exactly the same, so only one result is plotted in Figure 6.9(a).

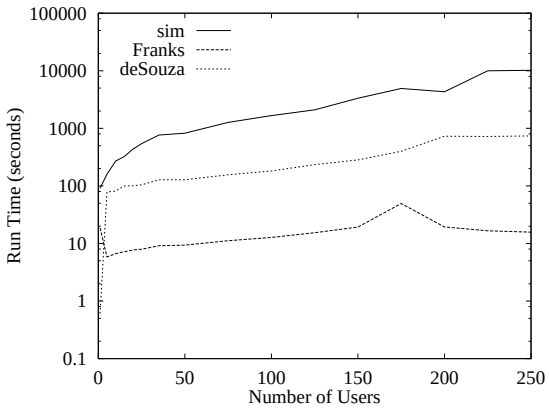
Figure 6.9(b) shows that the analytic solver scales very well, particularly when there are no bottlenecks in the system. When bottlenecks do arise, more iterations between layer submodels are needed because small changes in one submodel may cause large changes in another.

The overall speed of the solution is governed primarily by the "width" of the widest layer, counted in client tasks. Each client in a particular layer submodel corresponds to a chain in the submodel's queueing network. The number of customers in each chain corresponds to the number of replicas of the task in the layered queueing network model. In Linearizer, the solution speed is not affected by the number of customers in a chain. However, its run-time performance is of order JK^3 where J is the number of clients and K is the number of chains.

This example shows that the analytic solution of layered queueing models is both fast and sufficiently accurate for large industrial-scale models. Approximation errors are only large when the system being approaches saturation.



(a) Response Time



(b) Solution Run Time

Figure 6.9: Transaction Processing System Model Response Times and solution run times. Simulations were run with 95% confidence intervals of ±5%.

6.6 Solver Design

This section describes briefly the implementation of the various multiserver expressions described earlier in §6.2. Figure 6.10 shows the class diagram of the various server types. The classes that inherit from class *Markov_Phased* implement multiphase multiservers and use the client-server based overtaking approximation from Section 5.3. The notation for the expressions is described in the Glossary on page xxi. The equations used to find the marginal probabilities using either exact or approximate MVA are found in Appendix B.

6.6.1 Single Phase Multiservers

These classes, in the upper row of Figure 6.10, implement multi servers with one phase of service.

Class Reiser_Multi_Server

This class implements the original algorithm for exact MVA by Reiser and Lavenberg [107], with the modifications of Ruth [119] for class-dependent service times for multiservers with FIFO queueing. The solver does not incorporate the enhancement to account for numerical stability problems in exact MVA for load-dependent service centers [109]. The marginal probabilities, $P_m(j|\mathbf{n})$, are computed using the method by Krzesinski and Greyling [70] when using approximate MVA.

$$
\begin{aligned}
W_{mck}(\mathbf{n}) &= \frac{1}{J_m}\left[s_{mck} + \sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n}-\mathbf{e}_k) \right. \\
&\qquad \left. + \overline{S}_{mj} \sum_{j=0}^{J_m-2} (J_m - 1 - j) P_m(j|\mathbf{n}-\mathbf{e}_k) \right] \\
P_m(0|\mathbf{n}) &= 1 - \sum_{j=1}^{J} P_m(j|\mathbf{n})
\end{aligned}
\tag{6.12}
$$

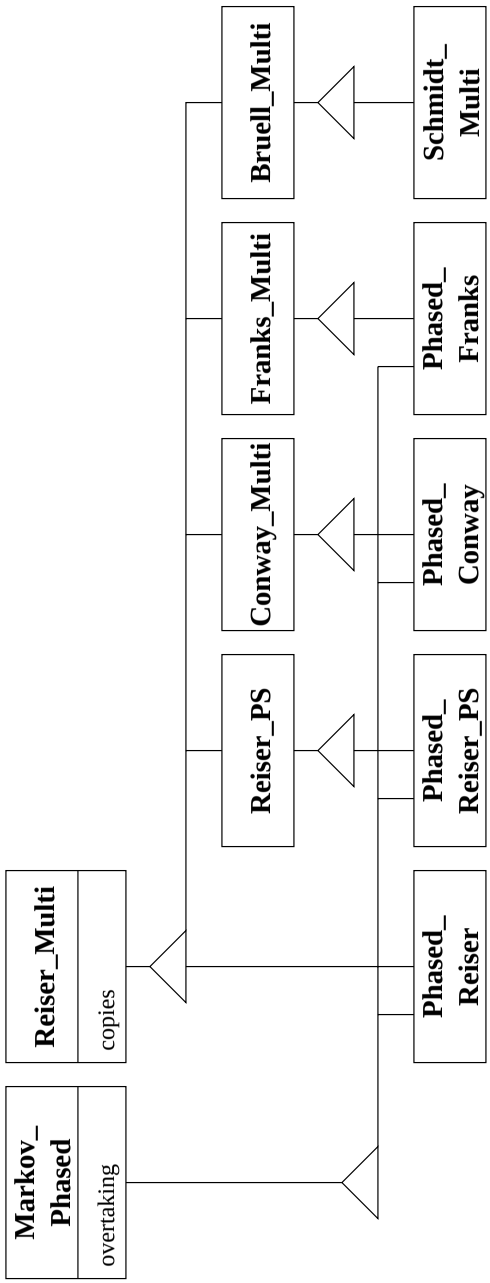


Figure 6.10: Class hierarchy for multiservers.

$$
\begin{aligned}
P_m(j|\mathbf{n}) &= \frac{1}{j}\sum_{k=1}^{K}\rho_{mk}(\mathbf{n})P_m(j-1|\mathbf{n}-\mathbf{e}_k), 0<j<J_m \\
P_m(Jm|\mathbf{n}) &= \frac{1}{J_m}\sum_{k=1}^{K}\rho_{mk}(\mathbf{n})\left(P_m(J_m|\mathbf{n}-\mathbf{e}_k)+P_m(J_m-1|\mathbf{n}-\mathbf{e}_k)\right) \\
\overline{S}_{mj} &= \frac{1}{\lambda_m(\mathbf{n}-\mathbf{e}_j)}\sum_{k=1}^{K}\lambda_{mk}(\mathbf{n}-\mathbf{e}_j)\sum_{e=1}^{E_m}v_{mek}s_{mek} \\
\lambda_m(\mathbf{n}-\mathbf{e}_j) &= \sum_{k=1}^{K}\lambda_{mk}(\mathbf{n}-\mathbf{e}_j)\sum_{e=1}^{E_m}v_{mek}
\end{aligned}
$$

For open models, (6.13) [73, (3.128)] is used. This expression is used by all of the other multiserver classes.

$$
\begin{aligned}
W_{me0} &= \overline{S}_{m0}\left(1+\frac{\rho_m(J_m\rho_m)^{J_m-1}}{J_m!A_m(1-\rho_m)^2}\right) \\
A_m &= \sum_{i=1}^{J_m-1}\frac{(J_m\rho_m)^i}{i!}+\frac{(J_m\rho_m)^{J_m}}{J_m!(1-\rho_m)}
\end{aligned}
\tag{6.13}
$$

Class Reiser_PS_Multi_Server

This class implements the original algorithm for exact MVA by Reiser and Lavenberg [107]. Most of the functions are inherited from class *Reiser_Multi_Server*.

$$
\begin{aligned}
W_{mek}(\mathbf{n}) = \frac{s_{mek}}{J_m}\Bigg[&1+\sum_{j=1}^{K}\sum_{i=1}^{E_m}L_{mij}(\mathbf{n}-\mathbf{e}_k) \\
&+\sum_{j=0}^{J_m-2}(J_m-1-j)P_m(j|\mathbf{n}-\mathbf{e}_k)\Bigg]
\end{aligned}
\tag{6.14}
$$

Class Conway_Multi_Server

The approximate multi-chain solution for load dependent service centers by de Souza e Silva et. al. [29] modified by Conway for Linearizer [24]. This technique, while accurate, is

expensive computationally.

$$W_{mek}(\mathbf{n}) = s_{mek} + \sum_{j=1}^{K} xe_{mkj}(\mathbf{n}) Q_{mj}(\mathbf{n}-\mathbf{e}_k) + P_m(J_m|\mathbf{n}-\mathbf{e_k}) xr_{mk}(\mathbf{n}) \tag{6.15}$$

$$\begin{aligned}
xe_{mkj}(\mathbf{n}) &= \sum_{\mathbf{i}\in\mathcal{A}_{jk}} \overline{S}_m(\mathbf{i}) ps_{mk}(\mathbf{i}, j, \mathbf{n}) \\
xr_{mk}(\mathbf{n}) &= \sum_{\mathbf{i}\in\mathcal{B}_k} \overline{S}_m(\mathbf{i}) ps_{mk}(\mathbf{i}, \mathbf{n}) \\
\mathcal{A}_{jk} &= \{\mathbf{i} | \sum_{x=1}^{K} i_x = J_m, \mathbf{i} \le \mathbf{n}-\mathbf{e}_k, i_j \ge 1\} \\
\mathcal{B}_k &= \{\mathbf{i} | \sum_{x=1}^{K} i_x = J_m, \mathbf{i} \le \mathbf{n}-\mathbf{e}_k\} \\
\overline{S}_m(\mathbf{i}) &= \left(\sum_{k=1}^{K} \frac{i_k}{s_{mk}} \right)^{-1} \\
ps_{mk}(\mathbf{i}, j, \mathbf{n}) &= \frac{a_m(\mathbf{i}, \mathbf{n}-\mathbf{e}_k)}{c_{mj}(\mathbf{n}-\mathbf{e}_k)} \\
c_{mj}(\mathbf{n}-\mathbf{e}_k) &= \sum_{\mathbf{i}\in\mathcal{A}_{jk}} a_m(\mathbf{i}, \mathbf{n}-\mathbf{e}_k) \\
ps_{mk}(\mathbf{i}, \mathbf{n}) &= \frac{a_m(\mathbf{i}, \mathbf{n}-\mathbf{e}_k)}{c_m(\mathbf{n}-\mathbf{e}_k)} \\
c_m(\mathbf{n}-\mathbf{e}_k) &= \sum_{\mathbf{i}\in\mathcal{B}_k} a_m(\mathbf{i}, \mathbf{n}-\mathbf{e}_k) \\
a_m(\mathbf{i}, \mathbf{n}-\mathbf{e}_k) &= \frac{J_m!}{\prod_{j=1}^{K} n_j!} \prod_{j=1}^{K} F_{mj}^{n_j}(\mathbf{n}-\mathbf{e}_k) \\
F_{mj}(\mathbf{n}-\mathbf{e}_k) &= \frac{\rho_{mj}(\mathbf{n}-\mathbf{e}_k)}{\rho_m(\mathbf{n}-\mathbf{e}_k)} \\
\rho_m(\mathbf{n}-\mathbf{e}_k) &= \sum_{j=1}^{K} \rho_{mj}(\mathbf{n}-\mathbf{e}_k)
\end{aligned}$$

Class Rolia_Multi_Server

The approximate multi-chain solution for load dependent service centers by Rolia [113, §3.3], which is surprisingly simple, fast, and accurate. Rolia's technique was modified so that it would work for exact MVA.

$$\begin{aligned} W_{mek}(\mathbf{n}) &= s_{mek} + pb_m(\mathbf{n}-\mathbf{e}_k)\sum_{j=1}^{K}\sum_{i=1}^{E_m} s_{mij}L_{mij}(\mathbf{n}-\mathbf{e}_k) \\ pb_m(\mathbf{n}-\mathbf{e}_k) &= \frac{1}{J_m}\left(\frac{\rho_m(\mathbf{n}-\mathbf{e}_k)}{J_m}\right)^{J_m} \end{aligned} \tag{6.16}$$

Class Bruell_Multiserver

This class implements the mulitserver expression (6.5). The marginal probabilities are computed in the `step()` function shown earlier in Figure 3.18 using (B.2) for exact MVA and (B.4) for approximate MVA. Equations (B.2) and (B.4) are found in Appendix B.

$$\begin{aligned} W_{mek}(\mathbf{n}) &= \sum_{\mathbf{i}\in\mathbf{n}} \mu(\mathbf{i},k)P_m(\mathbf{i}-\mathbf{e}_k|\mathbf{n}-\mathbf{e}_k) \\ \mu(\mathbf{i},k) &= S_{mk}\frac{|\mathbf{n}|}{\mu(|\mathbf{n}|)} \\ P_m(\mathbf{0}|\mathbf{n}) &= 1-\sum_{\mathbf{0}<\mathbf{i}\le\mathbf{n}} P_m(\mathbf{i}|\mathbf{n}) \\ P_m(\mathbf{i}|\mathbf{n}) &= \frac{1}{\mu(|\mathbf{i}|)}\sum_{k=1}^{K}\rho_{mk}(\mathbf{n})P_m(\mathbf{i}-\mathbf{e}_k|\mathbf{n}-\mathbf{e}_k) \end{aligned} \tag{6.17}$$

Class Schmidt_Multiserver

This class implements the approximation by Schmidt (6.6).

$$\begin{aligned} W_{mek}(\mathbf{n}) &= \sum_{\mathbf{i}\in\mathbf{n}} \mu(\mathbf{i},k) P_m(\mathbf{i}-\mathbf{e}_k|\mathbf{n}-\mathbf{e}_k) \\ \mu(\mathbf{i},k) &= S_{mk} + \frac{\max\{0,|i|-J_m\}}{J_m(|\mathbf{i}|-1)} \left(\sum_{j=1}^{K} i_j S_{mj} - S_{mk} \right) \end{aligned} \tag{6.18}$$

6.6.2 Multi Phase Multiservers

The classes described in this section implement multiservers with two or more phases. The overtaking probabilities are found using the client-server-based approximation described earlier in Section 5.3.

Class Markov_Phased_Reiser_Multi_Server

This class implements the original algorithm for exact MVA by Reiser and Lavenberg [107], with the modifications of Ruth [119] for class-dependent service times for multiservers with FIFO queueing and with modifications for two-phase service.

$$W_{mekp}(\mathbf{n}) = \frac{1}{J_m} \left[s_{mek1} + \sum_{j=1}^{K} \sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n}-\mathbf{e}_k) \right. \tag{6.19}$$

$$\left. + \overline{S}_{mj} \sum_{j=0}^{J_m-2} (J_m - 1 - j) P_m(j|\mathbf{n}-\mathbf{e}_k) + s_{mek2,\ mek}(p) \right] \tag{6.20}$$

Class Markov_Phased_Reiser_PS_Multi_Server

Not implemented.

Class Markov_Phased_Conway_Multi_Server

The approximate multi-chain solution for load dependent service centers by de Souza e Silva et. al. [29] modified to allow for two phases of service.

$$W_{mekp}(\mathbf{n}) = s_{mek1} + \sum_{j=1}^{K} xe_{mkj}(\mathbf{n}) Q_{mj}(\mathbf{n} - \mathbf{e}_k) + P_m(J_m|\mathbf{n} - \mathbf{e_k}) xr_{mk}(\mathbf{n}) + \frac{s_{mek2,\ mek}(p)}{J_m} \tag{6.21}$$

Class Markov_Phased_Rolia_Multi_Server

This class implements Rolia's multiserver expression, modified for two phases of service.

$$W_{mekp}(\mathbf{n}) = s_{mek1} + pb_m(\mathbf{n} - \mathbf{e}_k) \sum_{j=1}^{K} \sum_{i=1}^{E_m} s_{mij} L_{mij}(\mathbf{n} - \mathbf{e}_k) + \frac{s_{mek2,\ mek}(p)}{J_m} \tag{6.22}$$

6.7 Conclusions

Fast, accurate approximations now exist for multiclass multiservers with first-come, first-served queueing and chain-dependent service times. This is the first careful comparison we know of, of all the different approximations that have been given for these multiservers. Seven approximations were evaluated here in the context of layered queueing network models.

Rolia's approximation scales exceptionally well because it does not rely on the marginal probabilities, rather, it uses a simple expression using the utilization of the server. This expression has been generalized here to allow for chain-dependent service times while still retaining its impressive performance characteristics. From the tests in this paper, analytic solutions using new approximation are two to three orders of magnitude faster than simulation.

This chapter has introduced the analysis of two phase multiservers with one or more classes. Two-phase multiservers can yield significant performance improvements provided that the server is not saturated. Three of the seven multiclass multiserver waiting time expressions described in this paper (those which allowed for chain-dependent service times and FCFS scheduling) were modified to allow for two-phase server operations. The new approximations have approximation errors of less than 7% on average. They are also quite fast. Both properties mean that multi-tier client server systems with multiphase servers can be solved analytically both quickly and with sufficient accuracy for practical applications.

These new waiting time expressions broaden the scope of layered queueing networks. Further, layered queueing networks can be used to solve client/server systems with multiple tiers of intermediate servers both quickly and accurately.

The remaining limitations in the algorithms presented here are captured in the assuming made at the end of Section 5.2. In particular, it would be useful to model non-exponential or non-geometric distributions of demands.

Chapter 7

General Precedence Extensions to the Performance Model

The existing analytical performance modelling tools and improvements, described earlier, are useful for modelling a wide variety of computer systems with client-server-like behaviour. However, software designers are exploiting parallelism in application to improve performance. Some of these techniques include parallelism within tasks, asynchronous RPC, and early replies. None of these techniques are supported by conventional mean value analysis, even with hierarchical decomposition. Further, Stochastic Rendezvous Networks and the Method of Layers can only model systems with early replies.

This chapter describes extensions to the layered queueing network model to accommodate these and other forms of application parallelism. First, the concept of phases from SRVN and MOL models is generalized to objects called "activities" which can be connected together in a variety of patterns. Second, the rules for the behaviour of tasks with activities are described. Third, the new input grammar is described followed by sample input files. Finally, ways to reduce complicated patterns into simpler ones are described.

7.1 Activity Patterns

Activities are components in the performance model that represent the lowest level of detail necessary. Activities are connected together to form a directed graph which represents one or more execution scenarios. Execution may branch into parallel concurrent threads of control which may or may not execute in parallel on the target system. Execution may also choose randomly between different paths. The nomenclature used to show how activities are connected together is adopted from Chu et. al. [21] and is shown in Table 7.1. Smith [131] also has a notation which expresses the same information, but in more detail than is necessary for the performance models considered here.

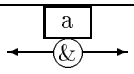
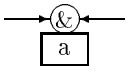
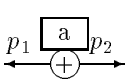
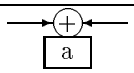
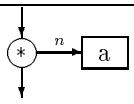
Name	Icon	Description
Activity		Basic unit of modelling detail.
Activity with Reply		An activity that generates a reply to entry e after it executes.
Connecting Arc		Transfer of control.
And-Fork		Start of concurrent execution. There can be any number of forked paths.
And-Join		A Synchronization point for concurrent activities.
Or-Fork		A branching point where one of the paths is selected with probability p. There can be any number of branches.
Or-Join		
Repetition		Repeat the activity an average of n times.

Table 7.1: Activity graph notation.

7.1.1 Sequential Execution

The simplest way to connect activities together is in a series, as shown in Figure 7.1.1.

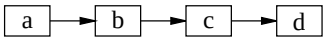


Figure 7.1: A sequence of activities.

As an example on the use of activities, consider a transaction that updates a database. Activity *a* represents data input, *b* transaction logging, *c* database update, and *d*, output confirmation.

7.1.2 Remote Procedure Calls

The client-server paradigm breaks up application processing into parts which run on a *client*, often situated on a user's desk, and a *server* located at another location. A client makes a request to a server through a remote procedure call [143, 5]. The server can then either respond to the request directly, or hand it off to other servers.

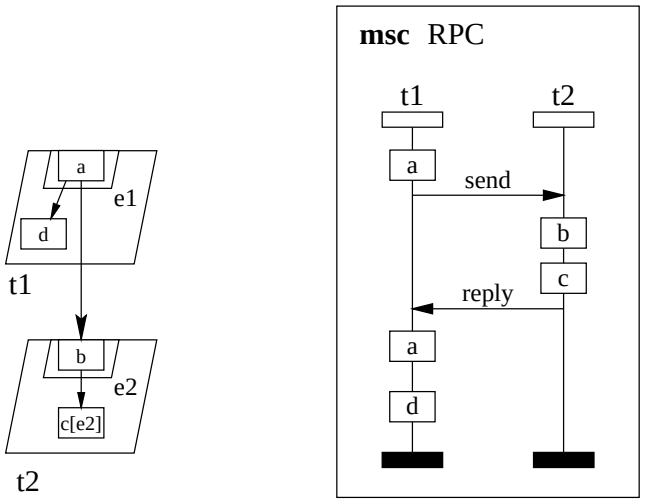
The sections that follow describe different ways of performing remote procedure calls. Immediately thereafter are techniques that can help shorten the blocking time.

Simple Remote Procedure Calls

Figure 7.2(a) shows the activity diagram for a simple remote procedure call based on the database example used earlier. In this figure, the the Layered Queueing Network notation described earlier in Section 3.1 is combined with the activity notation shown in Table 7.1 to show where each activity executes. In this example, the database logging activity, *b*, and update activity, *c*, run on task, *t2*. The Input and output operations, activities *a* and *d* respectively, run at the client task *t1*. Activities that send messages to other tasks always do so before continuing on to the next activity or activities. This behaviour matches that of phases.

Figure 7.2(b) shows the message sequence chart (MSC) for the remote procedure call

using the MSC'92 notation [154, 49, 115, 116]. The activities in Figure 7.2(a) have been transferred to the chart to show where program execution takes place and a particular order shown. Activity a is split into two activities on the chart because the actual remote procedure call takes place during the input operation from the user, and not between the activities. Notice the important differences between the arrow styles for a rendezvous/RPC, with a "barbed" arrowhead, and for precedence of activities (a smaller ordinarily filled arrowhead) in Figure 7.2(a). On the right, the MSC conventions for arrows are followed and all messages are asynchronous.



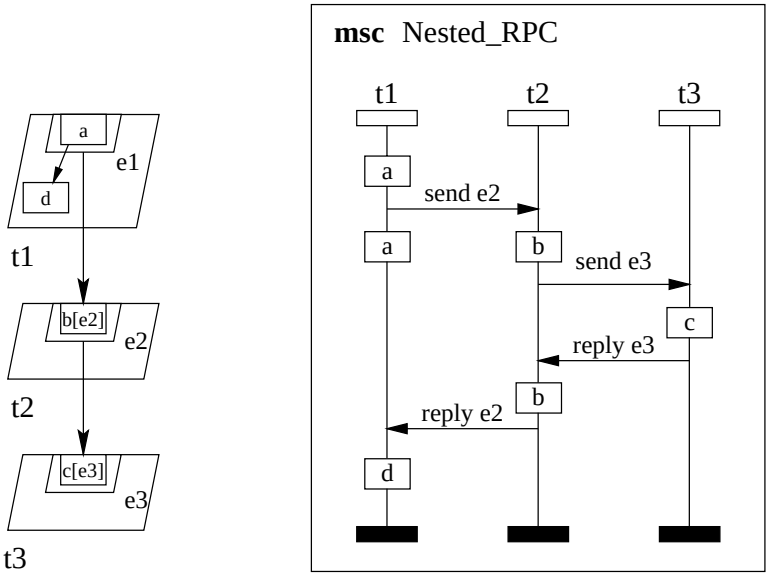
(a) Activity Diagram (b) Message Sequence Chart

Figure 7.2: Remote procedure call defined using activities.

Nested Remote Procedure Calls

Suppose next that the transaction logging and database update activities, b and c respectively, also can run on separate devices – the so called "three tiered" architecture [34]. Figure 7.3 shows one way to accomplish this goal by nesting remote procedure calls. With

this technique, both the client task, *t1*, and the intermediate server task, *t2*, are blocked during the RPC request to *t3*.

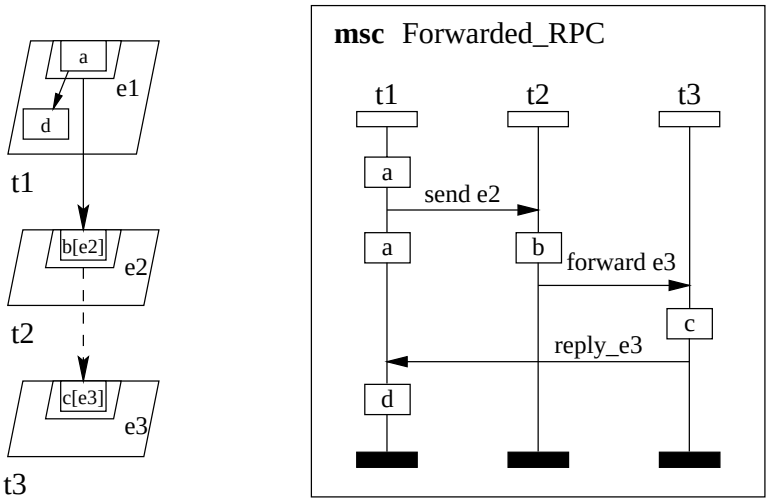


(a) Activity Diagram (b) Message Sequence Chart

Figure 7.3: Nested remote procedure call. The client, *t1*, calls *t2* who in turn calls *t3*. Both *t1* and *t2* are blocked until *t3* replies.

Forwarded Remote Procedure Calls

Remote procedure calls can also be *forwarded*, shown in Figure 7.4. In this example, the intermediate server task, *t2*, forwards the RPC request to the lower level server, *t3*. *T2* is then able to continue execution, unlike the nested RPC case described earlier. The client, *t1*, remains blocked until the last server replies.



(a) Activity Diagram (b) Message Sequence Chart

Figure 7.4: Forwarded Remote Procedure Call. The intermediate server, *t2*, forwards the RPC request to the lower level server, *t3* which in turn replies to the client, *t1*.

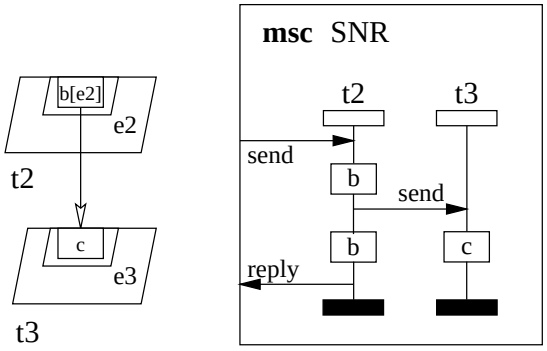
7.1.3 Patterns that Fork

Forking describes the phenomena where a thread of control splits into two or more concurrent subthreads. There are three ways forking may occur, described below.

Asynchronous Messages

The simplest form of starting a separate thread of control is by sending an asynchronous message (that is, a message to which no reply is expected). Both the client and server tasks can run in parallel.

Figure 7.5 shows the activity diagram and message sequence chart for an asynchronous send. Continuing with the simple database transaction example, task *t2* continues to execute the transaction logging activity *b* (note that since activity *b* generates a reply, it still blocks any callers). However, the database disk update activity, *c*, need not be executed while the original client waits, so the update can be accomplished using an asynchronous message to a separate task. The original client can therefore proceed as soon as the transaction is logged.



(a) Activity Diagram

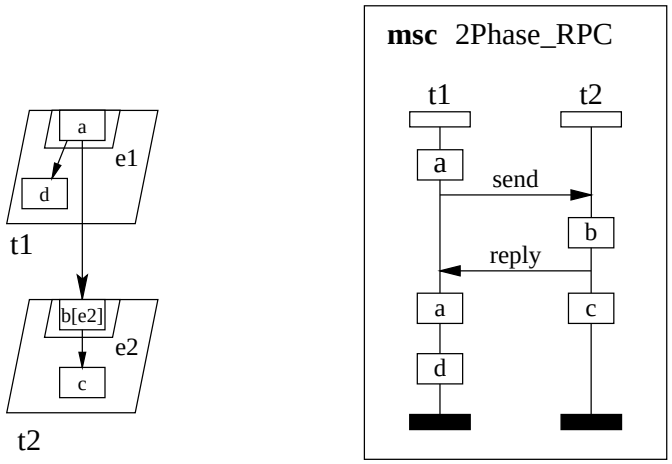
(b) Message Sequence Chart

Figure 7.5: Asynchronous send.

Two-Phase Execution

A second way of performing a fork is through an early reply at the server (see §5). An server using an early reply will reply to the client prior to completing all the of the work due to the client's request. The client and server can then proceed in parallel.

Figure 7.6 shows the activity diagram and message sequence chart for a server with an early reply. This Figure is very similar to Figure 7.2 except that the reply takes place after activity b instead of activity c.



(a) Activity Diagram (b) Message Sequence Chart

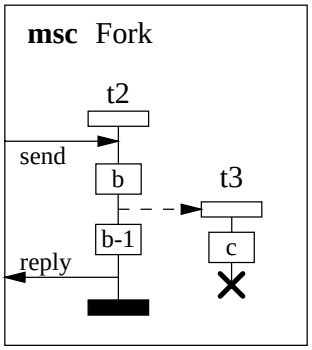
Figure 7.6: A remote procedure call with a second phase. The second phase at the server, labeled c, can run in parallel with the client.

From the client's perspective, the modified database transaction systems shown here and in Figure 7.5 are exactly the same in that the reply is generated as soon as activity b finishes. However, there is one important difference. A server with an early reply cannot accept new requests until the second phase execution completes (in this example, activity c). Furthermore, the second phase activity must run on the same processor. The server using asynchronous messages to update the database disk file can accept more requests before

the database update takes place and can also distribute the update and logging operations on separate processors, which also can improve performance. However, a system that uses asynchronous messages can overload the server executing activity *d*, possibly losing requests.

Thread Creation

The third way to fork a new thread of control is to actually create a new task. Referring to the Message Sequence Chart (Figure 7.7(a)), this method for forking is almost the same as using asynchronous messages. Differences arise in implementations when the cost of sending a message to a task that is already created is compared to the cost of creating a new task.



(a) Message Sequence Chart

Figure 7.7: Simple fork. Task *t3* is created and destroyed for each request.

7.1.4 Patterns that Join

Patterns that join arise when two or more tasks synchronize with a third. There are two cases, barrier synchronization and guarded accepts, described below. The differences in the two cases is based on when the reply is generated to the caller. In both cases, the task performing the synchronization operation must be single threaded. Furthermore, if the

sourcing tasks are not reference tasks, the messages must originate from a single fork point.

Barrier Synchronization

Barrier synchronization occurs when several threads of control wait for one and another; it is often found in scientific computing when array operations are farmed out to multiple processors. It can be implemented with a *synchronization server*, a task that waits for all of its client tasks to rendezvous before replying to them all, shown in Figure 7.8. Synchronization servers can only accept synchronous messages from dependent traffic sources because random asynchronous arrivals from independent sources cause unbounded queues. Asynchronous messages that originate from a common source are feasible and are described in Section 7.1.5 below.

Guarded Execution

Figure 7.9 shows a *guarded accept*. Both entries reply prior to the join taking place but the join must take place before entries can accept subsequent messages.

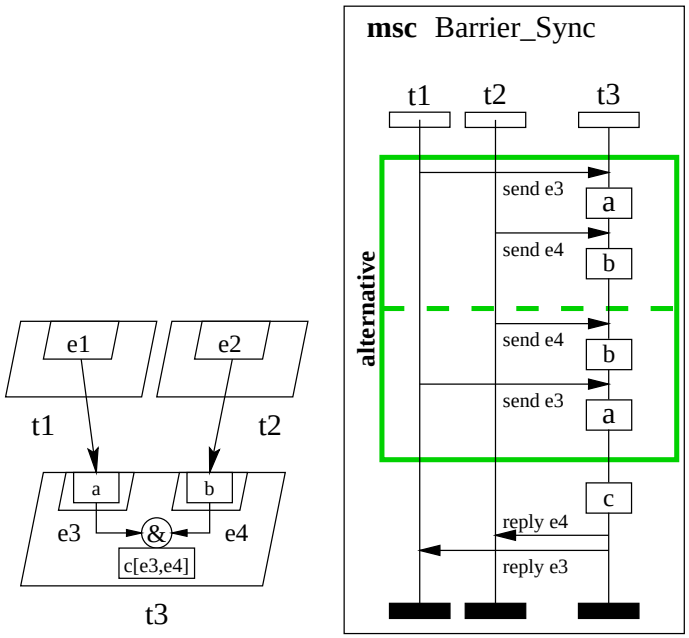
7.1.5 Patterns that Fork and Join

There are two forms of fork-join behaviour which are based on whether the fork and join take place within the same task, or in two separate tasks, described below.

Intra-Task Fork-Join

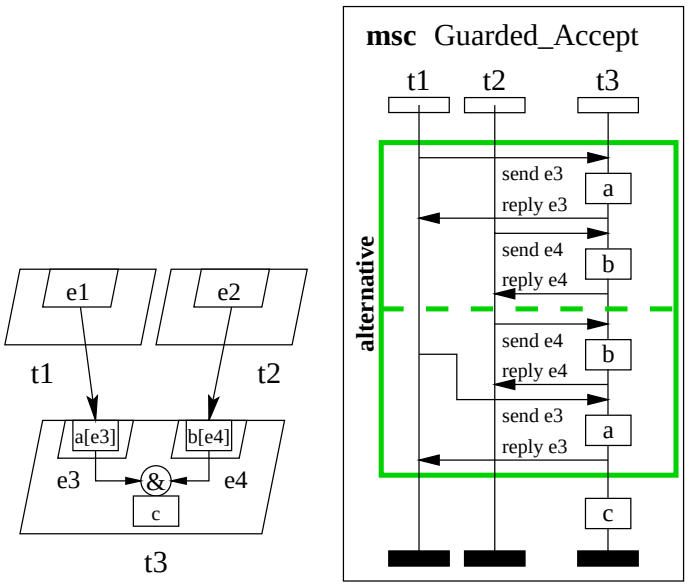
Intra-task fork-join behaviour occurs when the fork and join take place within the same task. This pattern is particularly useful for improving performance if parallelism in an application can be exploited.

An *Asynchronous RPC* is a technique which attempts to improve the performance of a remote procedure call by sending the request to the server as early as possible so that



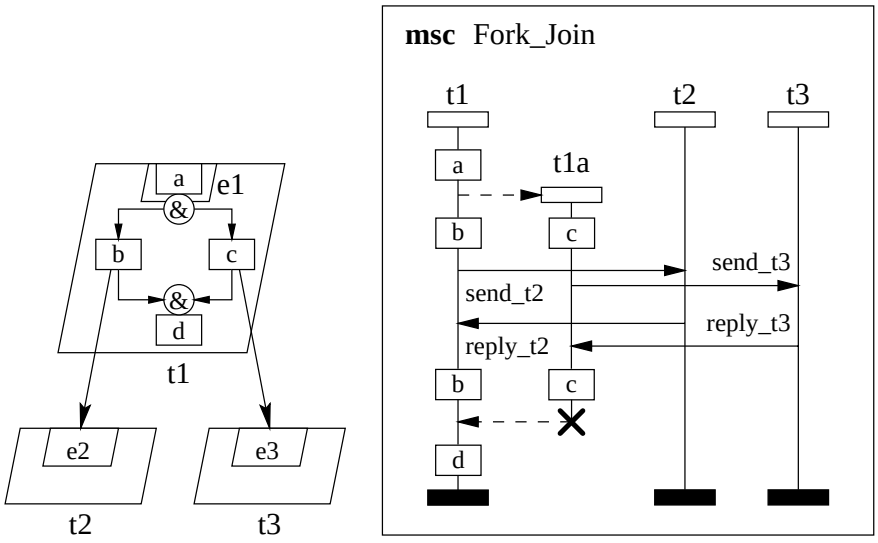
(a) Activity Diagram (b) Message Sequence Chart

Figure 7.8: Barrier synchronization. The shaded box denotes alternatives, i.e., the order of messages from *t1* and *t2* does not matter.



(a) Activity Diagram (b) Message Sequence Chart

Figure 7.9: Guarded accept.



(a) Activity Diagram (b) Message Sequence Chart

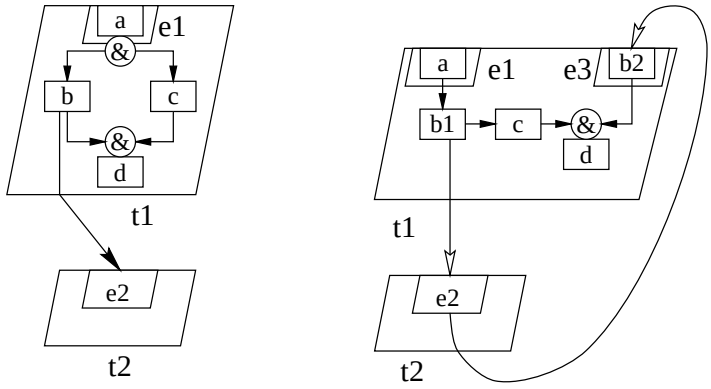
Figure 7.10: Fork-join within a task.

that the reply is waiting when the results are needed. There are two ways to model this phenomena:

1. Use a fork-join in the client with one thread making a RPC request while the other path continues with the main line processing (see Figure 7.11(a)).

2. Use deterministic asynchronous messages, then "join" the reply (see Figure 7.11(b)).

Of the two techniques, the first is preferable because it retains the semantics of the remote procedure call.



(a) Asynchronous RPC by Fork-Join with a remote procedure call.

(b) Asynchronous RPC by asynchronous messages with join.

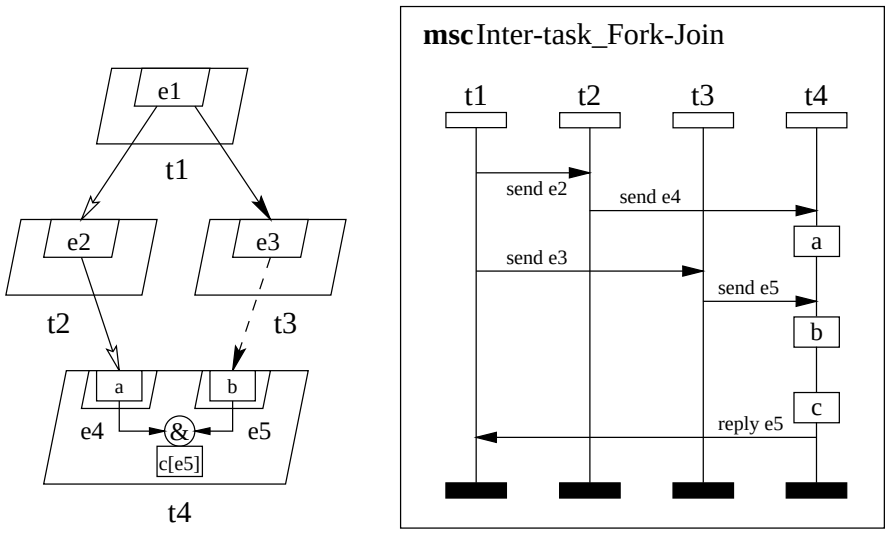
Figure 7.11: Asynchronous remote procedure calls using synchronous and asynchronous messages.

Inter-Task Fork-Join

Inter-task fork-join combines the behaviours of the fork and join, described earlier in Sections 7.1.3 and 7.1.4. Messages originate from a common client task, follow independent routes, then join at a common server task, shown in Figure 7.12. There are two cases:

1. Asynchronous threads joining with a (possibly forwarded) remote procedure call. All asynchronous messages must be sent prior to RPC request, or the asynchronous messages must be emitted from separate threads.

2. Multiple asynchronous threads.



(a) Activity Diagram (b) Message Sequence Chart

Figure 7.12: Inter-task fork-join. The left path is made up of an asynchronous request from *t1*. The right path is a forwarded rendezvous.

Synchronizing asynchronous messages from non-synchronized tasks is not practical (mathematically, one queue or the other will grow to infinity even if the arrival rates have exactly the same average value). Furthermore, all messages making up the paths from the fork point to the join point must be sent exactly once per fork-join. Forwarded RPC requests also must have a forwarding probability of 1.0 at all forwarding points.

7.2 Task Semantics

The previous section described various ways the activity extension can be used to enhance the modelling power of layered queueing networks. The extensions necessarily change the semantics of the original model. The semantics of the new model are described next.

7.2.1 Activity Execution

The unit of modelling in layered queueing networks is the *activity*. Activities consume time on the processor on which their task runs. By default, service time demand at processors is exponentially distributed. However purely deterministic, and hyper- and hypo-exponential service time distributions are also allowed. Non-deterministic service time distributions are modelled with either series or parallel exponential stages. Activities can also have zero service time in which case no request is made to the processor.

Activities can make requests to other tasks by way of synchronous (rendezvous) or non-synchronous (send-no-reply) messages. By default, an activity makes a random number of requests to its serving tasks. For this case, the user specifies the mean number of requests; the value need not be integral. An activity can also make a deterministic number of requests, but in this instance the number of requests made must be a positive integer. This case has been useful for modelling pipelines.

Requests sent to tasks are served in first-come, first-served order. Messages are received by *entries*, which differentiate task behaviour. Entries invoke activities which, in turn, pass control on to other activities through forks and joins (the complete list of interactions is shown in Table 7.1). After an AND-fork, all successor activities can execute in parallel; it is assumed that additional threads are available or created for this. After an OR-fork, only one of the successor activities is executed, which probability p. Sequential execution is a special case of an OR-fork with only one branch. Joins perform the reverse operation – multiple threads of control and connected together. AND-joins are special in that

they introduce synchronization delays because multiple independent threads of control are connected together.

7.2.2 AND Fork-Join

AND fork and join introduce a new set of behaviours to tasks simply because tasks now have multiple threads of control. Intra-task fork and join do not pose significant difficulty because the fork-join behaviour is contained within the task. This behaviour can take place within single, multi- and infinite server type tasks. Inter-task fork-join is a another matter altogether because synchronization now takes place at a task boundaries, i.e., the entries. Messages originating from a common source that must ultimately join can only do so at a single task. Also, deadlock can arise in a system that has more servers that join independent RPC request streams than tasks that originate the requests.

Tasks will only accept messages when all internally forked threads have completed. This constraint also implies that all threads that are forked, but not explicitly joined, must complete before the task can receive new messages. It also duplicates the existing second-phase behaviour.

Entries used to synchronize input streams also have new constraints. These entries can accept only one message during each join cycle. Messages sent to entries in which an activity has blocked on a join are not accepted. However, messages to other entries not blocked on joins will be accepted. Once a join takes place, messages will be received in time-stamp (first-come, first served) order.

Forked threads proceed independently if possible, otherwise they are processed from left to right in the input specification. Forked activities with zero service time are always processed first (it is assumed that these threads are asynchronous RPC requests).[1]

There is no limit on the number of internal threads forked. The user must limit threading

[1] Asynchronous RPC needs this requirement so that the send takes place as early as possible.

levels explicitly by serializing activities.

7.3 Grammar

In order to handle paradigms such as asynchronous remote procedure calls, forking with tasks, and synchronization, the Stochastic Rendezvous Network model described in Chapter 3 must be extended to describe these interactions. This section first describes briefly the major sections of the old grammar. Next, the extensions to the grammar for "activities" is described. The extensions are intended to be upward-compatible with this grammar to retain compatibility with the existing performance modelling tool set described in [40]. Examples showing the use of the extensions follow.

7.3.1 Abbreviated SRVN Input Grammar

The existing SRVN input file is divided into four sections as shown in Figure 7.3.1 (the complete input file grammar is found in Appendix A). The four sections are used for the following purposes:

General information: Sets the upper limit on the number of iterations the analytic solver will perform, the desired level of precision, the under-relaxation coefficient and print interval. These parameters are described in Appendix A.

Processor declarations: Declares the processors and their scheduling discipline.

Task declarations: Declares the tasks and entries in the model. Task-specific parameters such as priority and think times for "reference tasks" are defined here.

Entry declarations: Sets the parameters for all of the entries. These parameters include request rates to other entries and service times.

All identifiers are scoped globally within the input file. Furthermore, all identifiers must be declared before they are used.

$$
\begin{array}{lcl}
\langle SRVN\_input\_file \rangle & \rightarrow & \langle general\_info \rangle \ \langle processor\_info \rangle \ \langle task\_info \rangle \ \langle entry\_info \rangle \\
\langle general\_info \rangle & \rightarrow & \texttt{G} \ \langle comment \rangle \ \langle conv\_val \rangle \ \langle it\_limit \rangle \ \langle print\_int \rangle_{\text{opt}} \\
 & & \langle underrelax\_coeff \rangle_{\text{opt}} \ \langle end\_list \rangle \\
\langle processor\_info \rangle & \rightarrow & \texttt{P} \ \langle np \rangle \ \{\langle proc\_decl \rangle\}_1^{np} \ \langle end\_list \rangle \\
\langle task\_info \rangle & \rightarrow & \texttt{T} \ \langle nt \rangle \ \{\langle task\_decl \rangle\}_1^{nt} \ \langle end\_list \rangle \\
\langle entry\_info \rangle & \rightarrow & \texttt{E} \ \langle ne \rangle \ \{\langle entry\_decl \rangle\}_1^{ne} \ \langle end\_list \rangle
\end{array}
$$

Figure 7.13: Abbreviated SRVN input file grammar.

7.3.2 Activity Extensions

The existing SRVN input grammar is to be extended by adding new sections devoted to specifying activities. The *SRVN_input_file* rule is modified by adding the *activity_info* term begining with 'A' (see Figure 7.14). Activities are associated with tasks and not entries so the term is repeated for as many tasks that need an activity specification. Entries may be defined using the new activity grammar or the existing entry specification grammar[2]. However, only one method may be used for a particular entry.

An activity specification begins with the key-letter '`A`' followed by a *task_id*, a set of resource demands and connections to other tasks, and a set of activity connections within the task. Activity specifications follow the definition of all processors, tasks and entries.

Activities are scoped within tasks meaning that the name space is not shared among tasks[3].

The activity specification for a task is divided into two sections. The first section is used to define the activity's parameters such as service demand, and outgoing requests (messages) to other tasks. This section is similar to the entry specification section in the existing grammar, except that there is no phase information so each record accepts only

[2]The existing input grammar for an entry is somewhat more compact than the activity specification. Each phase defined using the existing grammar is considered to be an activity. The two specification techniques for a rendezvous to a two-phase server are shown together in Figure 7.16.

[3]This specification style differs from that of entries in that the name space used by entries is global among all tasks.

$\langle SRVN\_input\_file \rangle \rightarrow \langle general\_info \rangle \ \langle processor\_info \rangle \ \langle task\_info \rangle \ \langle entry\_info \rangle \ \{ \langle activity\_info \rangle \}_0^*$

$\langle activity\_info \rangle \rightarrow$ `A` $\langle task\_id \rangle \ \langle activity\_defn\_list \rangle$ `:` $\langle activity\_conn\_list \rangle \ \langle end\_list \rangle$

/* *Entry definition.* */

$\langle entry\_decl \rangle \rightarrow$ `A` $\langle entry\_id \rangle \ \langle activity\_id \rangle$ /* *Initial activitiy* */

/* *Activity definition.* */

$\langle activity\_defn\_list \rangle \rightarrow \{ \langle activity\_defn \rangle \}_1^{na}$

$\langle activity\_defn \rangle \rightarrow$ `s` $\langle activity\_id \rangle \ \langle ph\_serv\_time \rangle$ /* *Service time* */

| `c` $\langle activity\_id \rangle \ \langle coeff\_of\_variation \rangle$ /* *Sqr. Coef. of Var.* */

| `f` $\langle activity\_id \rangle \ \langle ph\_type\_flag \rangle$ /* *Phase type* */

| `y` $\langle activity\_id \rangle \ \langle to\_entry \rangle \ \langle ph\_RNV\_nb \rangle$ /* *Rendezvous* */

| `z` $\langle activity\_id \rangle \ \langle to\_entry \rangle \ \langle ph\_SNR\_nb \rangle$ /* *Send-no-reply* */

| `Z` $\langle activity\_id \rangle \ \langle think\_time \rangle$ /* *Think time* */

/* *Activity Connections.* */

$\langle activity\_conn\_list \rangle \rightarrow \langle activity\_conn \rangle \ \{ \texttt{;} \ \langle activity\_conn \rangle \}_1^{na}$

$\langle activity\_conn \rangle \rightarrow \langle join\_list \rangle$

| $\langle join\_list \rangle$ `->` $\langle split\_list \rangle$

| $\langle repeat\_list \rangle$ `->` $\langle split\_list \rangle$

$\langle join\_list \rangle \rightarrow \langle reply\_activity \rangle$

| $\langle and\_join\_list \rangle$

| $\langle or\_join\_list \rangle$

$\langle split\_list \rangle \rightarrow \langle activity\_id \rangle$

| $\langle and\_split\_list \rangle$

| $\langle or\_split\_list \rangle$

$\langle and\_join\_list \rangle \rightarrow \langle reply\_activity \rangle \ \{ \texttt{\&} \ \langle reply\_activity \rangle \}_1^{na}$

$\langle or\_join\_list \rangle \rightarrow \langle reply\_activity \rangle \ \{ \texttt{+} \ \langle reply\_activity \rangle \}_1^{na}$

$\langle and\_split\_list \rangle \rightarrow \langle activity\_id \rangle \ \{ \texttt{\&} \ \langle activity\_id \rangle \}_1^{na}$

$\langle or\_split\_list \rangle \rightarrow \langle prob\_activity \rangle \ \{ \texttt{+} \ \langle prob\_activity \rangle \}_1^{na}$

$\langle repeat\_list \rangle \rightarrow \langle real \rangle$ `*` $\langle activity\_id \rangle \ \langle next\_activity \rangle_{\text{opt}}$

$\langle prob\_activity \rangle \rightarrow$ `(` $\langle real \rangle$ `)` $\langle activity\_id \rangle$

$\langle reply\_activity \rangle \rightarrow \langle activity\_id \rangle \ \langle reply\_list \rangle_{\text{opt}}$

$\langle next\_activity \rangle \rightarrow$ `,` $\langle activity\_id \rangle$

$\langle reply\_list \rangle \rightarrow$ `[` $\langle entry\_id \rangle \ \{ \texttt{,} \ \langle entry\_id \rangle \}_0^{ne}$ `]`

Figure 7.14: Activity BNF.

one input term. Activities may make both rendezvous and send-no-reply requests to other tasks. The precise time at which a rendezvous or send-no-reply request is made from an activity cannot be specified (as is also the case with entry definitions). However, activities may have zero service time so highly deterministic operations can be defined, by placing a single message with an activity of zero execution demand.

The second section in an activity definition specifies the connections between activities. Activities are treated as nodes in a graph which are chained together or are connected through *and* and *or* forks and joins. Cycles in the activity graph are not permitted; repetition is specified with the *repeat_list* rule.

The starting activity for a particular graph is defined by the *'A'* rule for the entry definition; this activity is executed first when a message arrives at the corresponding entry. Starting activities may not be shared between entries, however, graphs originating from different entries within a task may join (see §7.4.3).

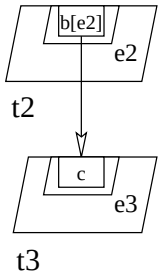
Entries that accept rendezvous type messages must generate a reply during the processing of the message. Replies are generated when an activity defined by the *reply_activity* rule completes. Entries that accept rendezvous messages cannot accept send-no-reply type messages, nor can they accept open arrivals. Replies can be *forwarded* to other tasks; forwarding is specified at the entry definition.

7.4 Examples

This section contains examples of the input specification language for a variety of interactions. It begins with some of the interaction patterns described in Section 7.1 and concludes with more complicated systems.

7.4.1 Asynchronous Send

Figure 7.15 shows three different ways of specifying the asynchronous send from Figure 7.5 in Section §7.1.3. Figure 7.16(b) shows the interaction using the existing input grammar. Figures 7.15(c) and 7.15(d) show the input specification using the activity extensions for *t2* and *t3* respectively.



(a) Activity Diagram

```
E 2
   s e2     1.0 -1
   s e3     0.5 -1
   z e2 e3 1.0 -1
-1
```

(b) Original syntax

```
E 2
   A e2     b1
   s e3     0.5 -1
-1

A t2
   s b1     1.0
   z b1 e3 1.0
-1
```

(c) E1 specificed using activities

```
E 2
   s e2     1.0 -1
   z e2 e3 1.0 -1
   A e3 c1
-1

A t3
   s c1     0.5
-1
```

(d) E2 specificed using activities

Figure 7.15: Input specification for the Asynchronous Send from Section 7.1.3. The general information, processor specification and task specification fields have been omitted.

7.4.2 Remote Procedure Call with Second Phase

Figure 7.16 shows three different ways of specifying the remote procedure call to a two-phase server from Figure 7.6 in Section §7.1.3. The client, *t1*, and server, *t2*, each have two activities, which are encoded using the existing input grammar as phases (shown in Figure 7.16(b)). Figure 7.16(c) shows the client specified using the activity extensions. Since there are two activities that execute one after the other, it is necessary to show their relationship. Figure 7.16(d) shows the input specification for the server. Since the server receives a remote procedure call, it is necessary for it to generate a reply. This action, done implicitly in Figures 7.16(b) and 7.16(c), must now be done explicitly.

7.4.3 Synchronization Server

The *"synchronization server"* is a special task used by the Method of Layers [113, §4] to perform barrier synchronization. Figure 7.17 shows how it is specified using the extended input grammar.

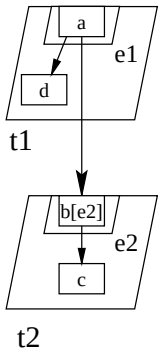
7.4.4 Intra-task Fork-Join

Figure 7.18 shows a sample input specification for specifying the intra-task fork-join pattern shown in Figure 7.10.

7.4.5 Asynchronous Remote Procedure Call

Asynchronous remote procedure calls can be specified either using an intra-task fork-join and a remote procedure call, or by two asynchronous messages and a join. The input specifications demonstrating the two methods, based on the activity diagrams shown earlier in Figure 7.11, are shown in Figure 7.19.

The specification based on the remote procedure call (Figure 7.19(a)) is preferable for two reasons. First, it retains RPC semantics (note that there is no cycle in the activity



(a) Activity Diagram

```
E 2
   s e1     0.75 0.25 -1
   y e1 e2 1.0  -1
   s e2     0.6  0.4  -1
-1
```

(b) Entry to entry syntax.

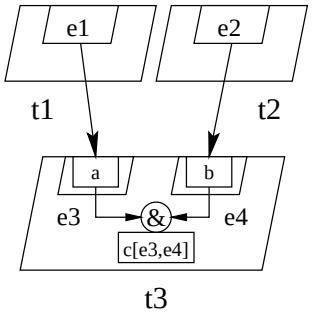
```
E 2
   A e1     a1
   s e2     0.6  0.4 -1
-1

A t1
   s a1     0.75
   y a1 e2 1.0
   s d1     0.25
:
   a1 -> d1
-1
```

(c) E1 specified using activities.

```
E 2
   s e1     0.75  0.25 -1
   y e1 e2 1.0   -1
   A e2 b1
-1

A t2
   s b1     0.6
   s c1     0.4
:
   b1[e2] -> c1
-1
```

(d) E2 specified using activities

Figure 7.16: Input specification for the Remote Procedure Call with a two-phase server from Section 7.1.3. The General information, processor specification and task specification fields have been omitted.



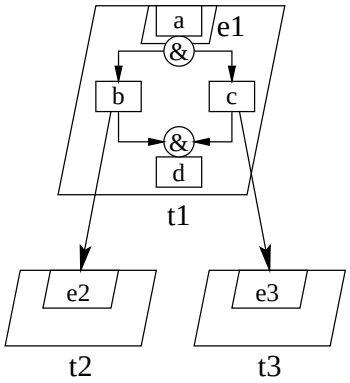
(a) Activity Diagram

```
E 4
   s  e1     1.5 -1
   s  e2     0.5 -1
   y  e1 e3  1.0 -1
   y  e2 e4  1.0 -1
   A  e3 a1
   A  e4 b1
-1
```

```
A t3
   s  a1  1.
   s  b1  1.
   s  c1  1.
:
   a1 & b1 -> c1;
   c1 [e3,e4]
-1
```

(b) Activity Specification

Figure 7.17: Input specification for the Barrier Synchronization example from Section 7.1.4. The general, processor and task information fields have been omitted.



(a) Activity Diagram

```
E 3
   A  e1 a1
   s  e2 1.5  -1
   s  e3 2.25 -1
-1
```

```
A t1
   s  a1     0.75
   s  b1     1.25
   s  c1     1.0
   s  d1     0.25
   y  b1 e2  1.
   y  c1 e3  1.
:
   a1 -> b1 & c1;
   b1 & c1 -> d1
-1
```

(b) Activity Specification

Figure 7.18: Input specification for the Intra-task Fork-Join system from Section 7.1.5. The general, processor and task specification sections have been omitted.

diagram). Second, the number of requests and the phase type of the activity initiating the asynchronous RPC is not constrained, unlike the specification based on asynchronous messages. (The latter must send exactly one request through the server back to the client.)

```
G "async-rpc-a" 1e-06 50 5 0.9 -1

P 2
   p  p1 f
   p  p2 f
-1

T 2
   t  t1 r  e1   -1  p1
   t  t2 n  e2   -1  p2
-1

E 2
   A  e1 a1
   s  e2    1.5  -1
-1

A t1
   s  a1    0.75
   s  b1    0.0
   f  b1    1
   s  c1    1.0
   s  d1    0.25
   y  b1 e2 1.0
:
   a1 -> b1 & c1;
   b1 & c1 -> d1
-1
```

(a) Fork-Join with a remote procedure call.

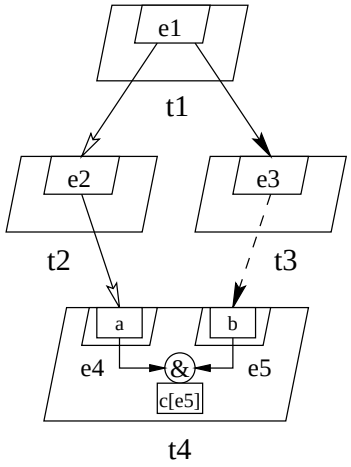
```
G "async-rpc-b" 1e-06 50 5 0.9 -1

P 3
   p  p0 f
   p  p1 f
   p  p2 f
-1

T 3
   t  t0 r  e0       -1 p0
   t  t1 n  e1 e3    -1 p1
   t  t2 n  e2       -1 p2
-1

E 4
   s  e0     0.0  -1
   y  e0 e1  1.0  -1
   A  e1 a1
   A  e3 b2
   s  e2     1.5  -1
   f  e2     1    -1
   z  e2 e3  1.0  -1
-1

A t1
   s  a1     0.75
   s  b1     0.0
   f  b1     1
   z  b1 e2  1.0
   s  b2     0.0
   s  c1     1.0
   s  d1     0.25
:
   a1 -> b1;
   b1 -> c1;
   c1 & b2 -> d1;
   d1[e1]
-1
```

(b) Asynchronous messages with a join.

Figure 7.19: Input specification for the Asynchronous Remote Procedure Call activity diagrams shown in Figure 7.11.

7.4.6 Inter-task Fork-Join

Figure 7.20 shows the input specification for the inter-task fork-join example shown earlier in Figure 7.12. The path through the intermediate server *t2* is by way of asynchronous messages while the path through *t3* consists of a forwarded remote procedure call. This structure complicates the input specification somewhat. First, the remote procedure call must be done last as it will block the client *t1*. To guarantee that the remote procedure call is done after the asynchronous send, the request is placed in a separate following phase. Second, there must be exactly one message generated along each path per cycle of the client. Consequently, all phases that generate messages must be marked as being deterministic (using the `f` record for an entry).



(a) Activity Diagram

```
E 5
   s  e1     0.5   0.5 -1
   f  e1     1     1   -1
   z  e1 e2  1.0   0.0 -1
   y  e1 e3  0.0   1.0 -1
   s  e2     1.5       -1
   f  e2     1         -1
   z  e2 e4  1.0       -1
   s  e3     2.25      -1
   F  e3 e5  1.0       -1
   A  e4 a1
   A  e5 b1
-1
```

```
A t4
   s  a1  0.75
   s  b1  1.25
   s  c1  1.0
:
   a1 & b1 -> c1;
   c1[e5]
-1
```

(b) Activity Specification

Figure 7.20: Input specification for the Inter-task Fork and Join from Section 7.1.5. The general, processor and task specifications have been omitted.

7.4.7 Chu, Sit and Leung Example

Chu et. al. [21] described a hypothetical distributed system consisting of eight "modules" running on two processors. The modules execute independently from each other except that four of the eight share a common resource. Results are passed from one module to the next through messages. The module interactions are shown in Figure 7.21; the parameters for the model are shown in Table 7.2. The solved the model by a combination of queueing models and a complex fork-join delay approximation (See Section 2.4.3).

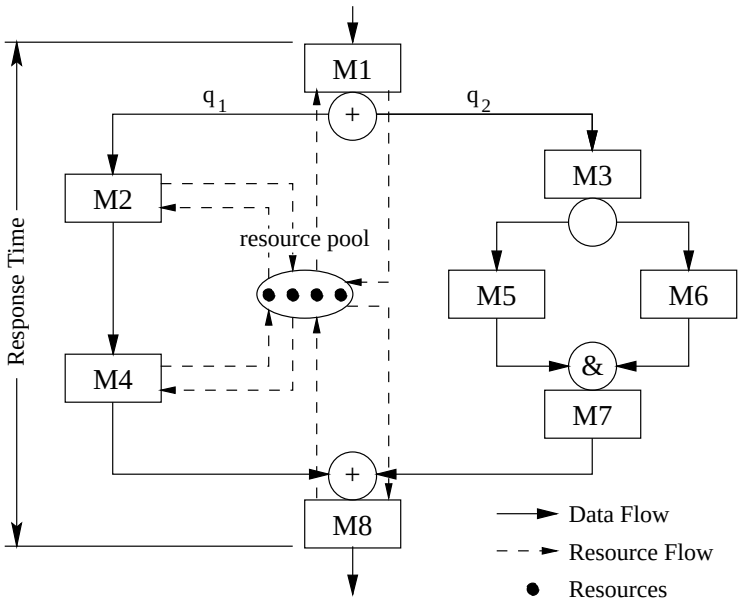


Figure 7.21: A task control-flow graph of the Chu, Sit and Leung model (from [21]).

Figure 7.22 shows the layered queueing network model for the Chu et. al. design. The model is divided up into three sections outlined with boxes in the figure these are:

- the *traffic source*,
- the *original system* and
- the *resource pool*.

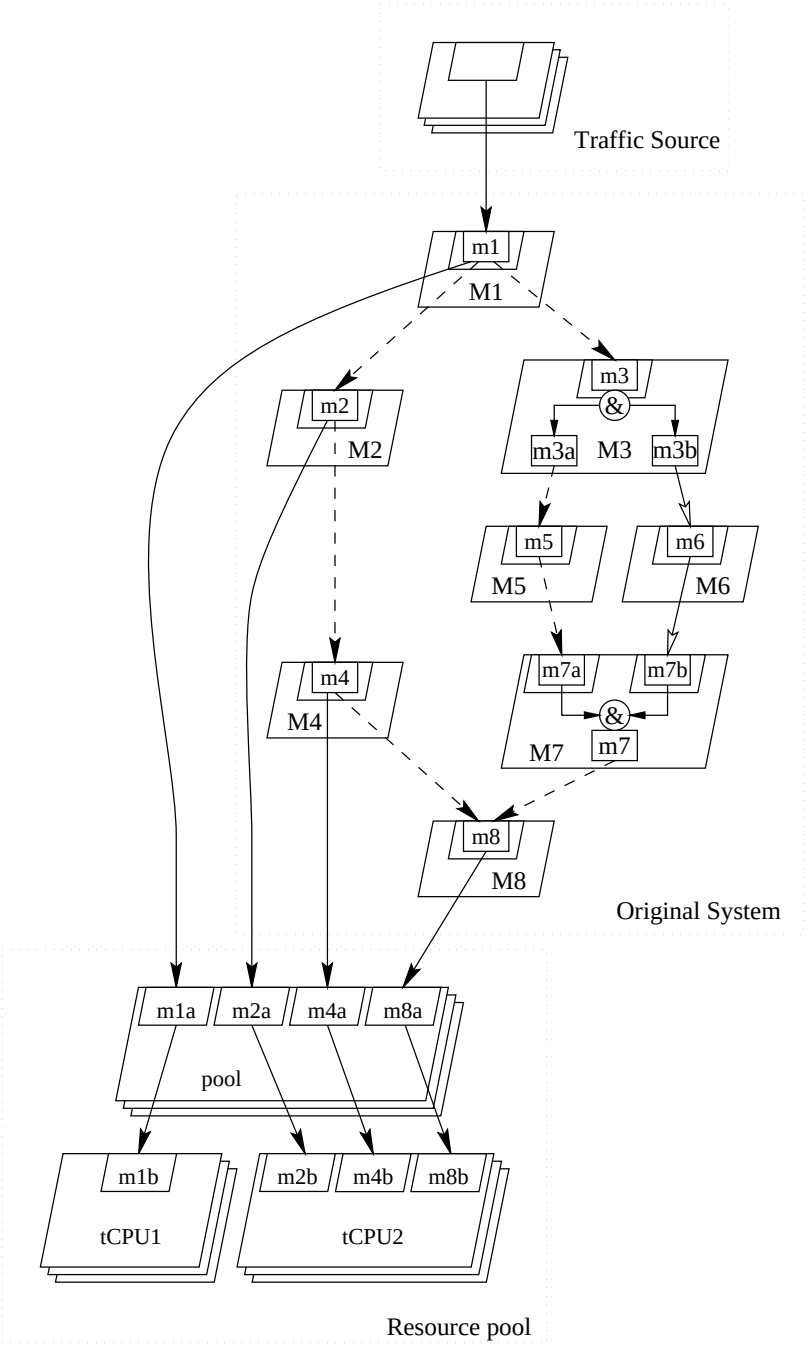


Figure 7.22: Layered queueing network for Chu, Sit and Leung model.

Module	Execution Time	Processor Assignment
m1	5.0	CPU1
m2	1.0	CPU2
m3	1.0	CPU2
m4	5.0	CPU2
m5	5.0	CPU2
m6	1.0	CPU1
m7	5.0	CPU1
m8	1.0	CPU2

Table 7.2: Model parameters for module assignment 1. The branching probabilities are: $q_1 = 0.6$ and $q_2 = 0.2$.

The "traffic source" box consists of a task pool and generates RPC requests to the original model. It converts the open model of Chu et. al. into a closed model.

The "original system" box consist of the modules from the original model. To mimic the original open system design, RPC requests are forwarded from *M1* to *M8* by way of either *M2* or *M3*. No special features from this chapter are needed to specify task *M1* because the paths are selected based on the forwarding probabilities for the arcs from *m1* to *m2* and *m3*. However, a minor difficulty does arise because of the forking that takes place at module *M3*. Forwarding can only follow one branch from the fork point. To solve this problem, the other branch is made up of asynchronous messages that are spawned each time *m3* is invoked. These messages are "joined" with the forwarded requests from the other path at *m7*. Finally, *m8* is called by either *m4* or *m7* and issues the reply back to the traffic source.

The last component of the model is the resource pool; it models the tokens in the resource pool ellipse in Figure 7.21. The resource itself is modelled using the transformation technique by Woodside [150]. The shared resource proper is modelled using a multi-server task; the number of instances of the multiserver corresponds to the number of tokens in the resource pool. Acquisition of resources is modelled by making rendezvous requests to the resource multi-server; blocking at the resource will only arise once all instances are acquired.

The service time parameters for the users of the resource pool are modified by moving the demand from the user to the corresponding entry on the resource pool multi-server; the service time at the original task is then set to zero. However, the user will continue to see the delay for this demand because of the blocking nature of the remote procedure call.

Additional auxiliary tasks are needed to model the resource pool because the tasks that share the common resource do not all execute on the same processor. The multi-server task labeled *pool* itself pushes the user's demand down to lower level servers, labeled *tCPU1* and *tCPU2* in Figure 7.22. This pair of servers models the demand at the CPU of the corresponding module in the original model. Further, as the *pool* task models the resource limit, these tasks are simply infinite servers in the layered queueing network model. Table 7.3 shows the parameters of the final model.

"Original Model" part			"Resource Pool" part		
Module (Entry)	Execution Time	Processor Assignment	Module (Entry)	Execution Time	Processor Assignment
m1	0.0	—	m1a	0.0	—
m2	0.0	—	m2a	0.0	—
m3	1.0	CPU2	m4a	0.0	—
m4	0.0	—	m8a	0.0	—
m5	5.0	CPU2	m1b	5.0	CPU1
m6	1.0	CPU2	m2b	1.0	CPU2
m7	5.0	CPU1	m4b	5.0	CPU2
m8	0.0	—	m8b	1.0	CPU2

Table 7.3: Parameters for the layered queueing network model. Tasks with zero service time do not need a processor to execute upon.

The input model definition, complete with extensions for fork-join behaviour, is shown in Figure 7.23. Of all the tasks in this model, the only one that needs the extended notation is *M7* because it is synchronizing two streams. The fork operation, performed by *M3* does not need an explicit fork because the route by way of *M5* is invoked using asynchronous messages (see 7.1.5). Also, tasks that reference the shared resource pool, *M1*, *M2*, *M6*

and *M8*, and the resource pool task itself all run on an "infinite processor" called *NIL*. As these tasks do not consume CPU time, they can be assigned to any processor at all, the *NIL* processor is added simply to highlight this fact. Finally, the number of tokens in the resource pool is defined by the number of instances of the task *pool*.

7.5 Activity Aggregation

An activity graph may contain a combination of the various subgraphs types shown in Figures 7.24 – 7.27. In the solver, each of the subgraphs is aggregated to a single activity whereupon aggregation once again takes place. Aggregation continues until only one or two activities remain which represent the average behaviour of the original graph. The aggregated activities will then be used as *phases* in the layered queueing network model to solve for contention delays. The sections that follow describe the standard formulations to aggregate the subgraphs.

7.5.1 Sequential Activities

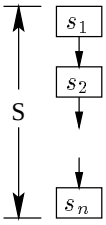
Activities that execute sequentially, shown in Figure 7.24, are aggregated to a single activity by summing the service times of each individual activity. The service times of each of the activities are assumed to be independent random variables, so the variance of the aggregate is the sum of the variances of each of the activities contained therein.

7.5.2 OR Fork-Join

The aggregate service time for an OR fork-join activity graph is the sum of the weighted services times for each activity, shown in Figure 7.25. The aggregate variance is found using (7.4) [131, p 178]. In this expression, I is an index set consisting of all unique pairings for the set $\{1, 2, ..., n\}$.

```
G "" 0.000010 100 10 0.9 -1

P 0
   p clients  f  i
   p CPU1     f  0.5
   p CPU2     f  0.5
   p NIL      f  i
-1

T 0
   t clients  r  clients -1 clients 0 m 10
   t M1       n  m1 -1 NIL 0
   t M2       n  m2 -1 NIL 0
   t M3       n  m3 -1 CPU2 0
   t M4       n  m4 -1 NIL 0
   t M5       n  m5 -1 CPU2 0
   t M6       n  m6 -1 CPU1 0
   t M7       n  m7a m7b -1 CPU1 0
   t M8       n  m8 -1 NIL 0
   t pool     n  m1a m2a m4a m8a -1 NIL 0 m 1
   t tCPU1 n  m1b -1 CPU1 0 i
   t tCPU2 n  m2b m4b m8b -1 CPU2 0 i
-1

E 0
   s   clients      0.0 16.0 -1
   y   clients m1   0.0 1.0  -1
   s  m1b           5.0 0.0  -1
   s  m2b           1.0 0.0  -1
   s  m4b           5.0 0.0  -1
   s  m8b           5.0 0.0  -1
   s  m1            0.0 0.0  -1
   y  m1   m1a      1.0 0.0  -1
   F  m1   m2       0.6 -1
   F  m1   m3       0.4 -1
   s  m2            0.0 0.0  -1
   y  m2   m2a      1.0 0.0  -1
   F  m2   m4       1.0 -1
   s   m3          1.0 0.0 -1
   F   m3   m5     1.0 -1
   z   m3   m6     1.0 0.0 -1
   f   m3          1   0   -1
   s   m4          0.0 0.0 -1
   y   m4   m4a    1.0 0.0 -1
   F   m4   m8     1.0 -1
   s   m5          5.0 0.0 -1
   F   m5   m7a    1.0 -1
   s   m6          1.0 0.0 -1
   z   m6   m7b    1.0 -1
   f   m6          1   0   -1
   A   m7a m7a
   F   m7a m8     1.0 -1
   A   m7b m7b
   s   m8          0.0 0.0 -1
   y   m8   m8a    1.0 0.0 -1
   s   m1a         0.0 0.0 -1
   y   m1a m1b     1.0 0.0 -1
   s   m2a         0.0 0.0 -1
   y   m2a m2b     1.0 0.0 -1
   s   m4a         0.0 0.0 -1
   y   m4a m4b     1.0 0.0 -1
   s   m8a         0.0 0.0 -1
   y   m8a m8b     1.0 0.0 -1
-1

A M7
   s   m7a         0.0
   s   m7b         0.0
   s   m7          5.0
:
      m7a & m7b -> m7;
      m7[m7a]
-1
```

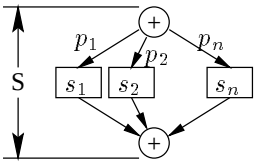
Figure 7.23: Input specification for Chu, Sit and Leung model.



$$S = \sum_{i}^{n} s_i \tag{7.1}$$

$$\sigma^2 = \sum_{i}^{n} \sigma_i^2 \tag{7.2}$$

Figure 7.24: Sequential activity aggregation.



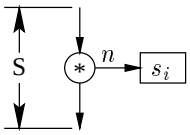
$$S = \sum_{i}^{n} p_i s_i \tag{7.3}$$

$$\sigma^2 = \sum_{i}^{n} p_i \sigma_i^2 + \sum_{i,j \in I} p_i p_j (s_i - s_j)^2 \tag{7.4}$$

Figure 7.25: OR fork-join activity aggregation.

7.5.3 Repetition

The mean service time for an activity which is repeated n times on average is the product of the average activity time multiplied by the average number of times the activity is executed. The aggregate variance is found using (7.6) [131, p 178]. σ_n^2 is the variance of the random



$$S = n s_i \tag{7.5}$$

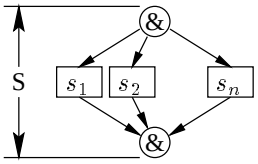
$$\sigma^2 = n \sigma_i^2 + s_i^2 \sigma_n^2 \tag{7.6}$$

Figure 7.26: Repeated activity aggregation.

variable n.

7.5.4 AND Fork-Join

AND fork-join subgraphs, shown in Figure 7.27, consist of activities that are invoked by a common parent, synchronize on a common child, and execute concurrently with one and another.



$$S = \max(s_1, s_2, ..., s_n) \tag{7.7}$$

Figure 7.27: AND fork-join activity graph aggregation.

The aggregate mean service time for an AND fork-join subgraph is $\max(s_1, s_n, ...s_n)$ where s_i is the mean service time for activity i. If the distribution functions $F(s_i)$ are known, then then

$$F_{\max}(S) = \prod_{i=1}^{n} F_i(s_i) \tag{7.8}$$

The moments can then be found from $F_{\max}(S)$.

The distribution functions for activities is not known, so (7.8) cannot be applied directly. An approximation, using the mean and variance, is described in Section 8.2.2.

7.5.5 "Non-Regular" Graphs

Figure 7.28 shows a graph which cannot be aggregated using the rules above. More complex graph analysis will be required.

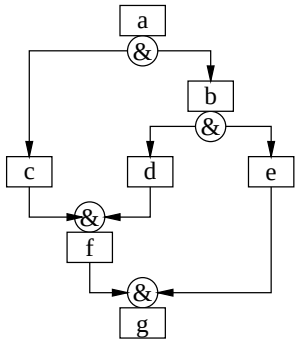


Figure 7.28: Graph which is difficult to aggregate.

7.6 Conclusions

This chapter describes a new grammar for incorporating activities into the layered modelling framework. *Activities* are components in the performance model that represent the lowest level of detail necessary and can be connected together with forks and joins.

A simulator has been written which executes the full semantics of the activity graph notation described here. The analytic solver LQNS can solve systems with hierarchical intra-task fork joins; described in the next chapter. Further research is needed to capture accurately the correlation of arrivals at joins for systems with inter-task fork-joins.

Chapter 8

Queueing Networks with Fork-Join Interactions

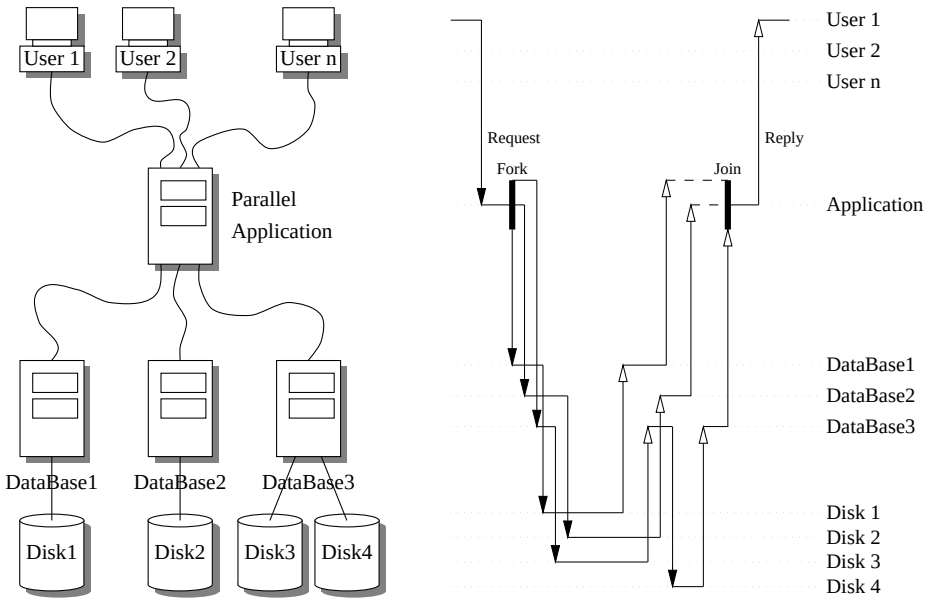
The preceding chapter introduced the concept of activities to layered queueing networks. This chapter describes approximations for solving systems with intra-task fork-joins. A "compensated complementary delay" approximation is described which exploits layered queueing approximations for layered resources which occur in client-server architectures, based on synchronization delay estimates and adjusted levels of contention. The new approximation uses the overlap of parallel branches and a new fast calculation of join delays. It gives acceptable errors (averaging about two percent), and has an enormously lower computational cost compared to the competing approach based on decomposition.

The material in this chapter was mostly published in [42].

8.1 Performance Implications of Parallel Operations

Processes in a distributed system use services over a network, and may have serious delays when blocked waiting for a service to complete. To reduce the impact of blocking, or to simply speed up a set of operations, a process that needs several operations could request

them in parallel, by forking parallel threads that then make the service requests. Figure 8.1 illustrates a database application with this kind of internal parallelism, sending requests to three separate databases and then combining the results. The right hand side shows how a user request is broken into three requests at the fork and sent to the three databases. At the join the application waits for all the replies. These *server systems with parallelism* may occur in business data processing, world wide web servers [55], and in geographic information systems, among others. Notice that in a multi-level system there can be parallel steps in servers at any level. For example, there might also be a parallel operation within one of the databases, issuing parallel requests to disks or file servers (such an example will be described later).



(a) System Architecture (b) Interaction Timeline

Figure 8.1: An example representing a business client-server system. The client workstations each make requests to a server running a parallel application. The application, in turn, makes requests to the three lower-level database servers. Figure 8.1(b) shows a sample interaction between User 1 and the parallel application.

Other performance optimizations are logically equivalent to parallel threads even if the threads are hidden from the programmer. For instance for prefetching a file, a file access thread is forked at the prefetch and joined when the data is used.

Since parallelism is a performance optimization, a way to estimate the performance benefit would be helpful. We would like to employ analytic modelling for speed, but the models are complex to create because they combine parallelism effects with resource contention and (in many cases) with simultaneous resource possession that arises when a server or thread blocks to wait for a reply. Furthermore, practical systems may be quite large, so the method must scale up well. The approach taken here is to add the parallelism effects to an existing modelling framework, the layered queueing model or "rendezvous network" [152, 110], which is scalable and which assists the model-builder in representing the resource contention and the simultaneous resources. Layered models have been validated against simulations on hundreds of cases and in a variety of practical and laboratory applications [126, 148, 30].

The common characteristics of service systems with parallelism are:

- random unsynchronized requests to a server from multiple clients,
- heterogeneous operations on the parallel paths, including perhaps requests to other servers,
- multiple layers of servers having resource constraints at all layers, such that a request may hold resources at higher layers while executing a low level service,
- a substantial element of uncertainty or randomness in the duration of the parallel paths,
- no system-wide synchronization between workload components. Rather, parts of the workload are due to separate users who compete for resources. This is different from massively parallel computing in which the parallel paths all execute the same instructions and there may be frequent global synchronizations.

8.1.1 Previous Work

Several authors have developed analytic models for parallel service systems. Heidelberger and Trivedi [51] considered a simple computational loop executed by each user, alternating between a sequential section and a parallel section. They did consider contention between the parallel subtasks, but only one layer of resources. They compared two modelling approaches, called "decomposition" and "complementary delays" (CD). Decomposition requires a separate performance model for each possible state of parallelism, as the number of independent users increases the algorithm suffers combinatorial explosion. It does not scale up to large systems. CD determines an equivalent delay for the synchronization at the join point and scales up well.

In later papers the decomposition approach was extended. Thomasian and Bay [138] developed a way to apply it to larger task graphs with a general structure of parallel and sequential parts. Petriu et. al. [99] applied it to layered systems with any number of layers of servers (that is, with complex simultaneous resources). In both cases the cost is prohibitive if the number of independent users is even moderately large.

The CD approach has also been extended. Chu, Sit and Leung [21] modelled the join delay more carefully by considering the distribution of delay on each parallel path, and included one additional upper-layer resource (equivalent to a middle layer of service). Mak and Lundstrom [84] improved the accuracy by accounting for the overlap in time of synchronized parallel sub-paths. However their development was limited to one set of forked jobs at a time, and also did not include simultaneous resources. Both these papers assumed that the branches of a parallel operation have an exponentially-distributed delay, to calculate join delay based only on mean value.

Another class of models for these systems is based on Markov Chains derived from Petri nets or Process algebras. Models in this class represent all the states of parallelism in a single model, together with the contention in each state. However they suffer even worse

from state explosion than the decompositions above, so we must regard them as unscalable in their present form.

If decomposition and state-based approaches do not scale up, then we must see if the delay-based approach can be made more accurate. To do so, this chapter extends the calculation of the overlap in contention given in [84], to make it apply to layered systems with multi-threaded servers. It also exploits the variance of the delay on the parallel paths, by using a "three-point" approximation for the join delay. This gives a "Compensated Complementary Delay" (CCD) approach described in section 5.2.

The CCD analysis is applied in Section 8.3 to several examples, showing that:

- the approach scales up to systems with many servers having parallel sections and to many independent users,
- approximation errors are limited to a few percent,
- from the 228 examples studied in [51], the CCD errors are about three times as large as errors from decomposition (CD errors are about ten times as large).

It is concluded that the delay approximation combines usable accuracy with scalable complexity.

8.1.2 The Example System as a Layered Model

Figure 8.2 shows a simplified version of the system of Figure 8.1 (i.e., without the disks) as a layered queueing model, showing parallel requests from the Application Server in the middle to the Database servers at the bottom. The parallelism is defined by the little *activity graph* nested inside the ParApp service in the Application Server. An activity graph defines precedence among its activities.

In a layered model, each task is a resource; a multi-threaded task is a set of M resources with a single queue. At the top of Figure 8.2 is a set of N Client tasks which cycle,

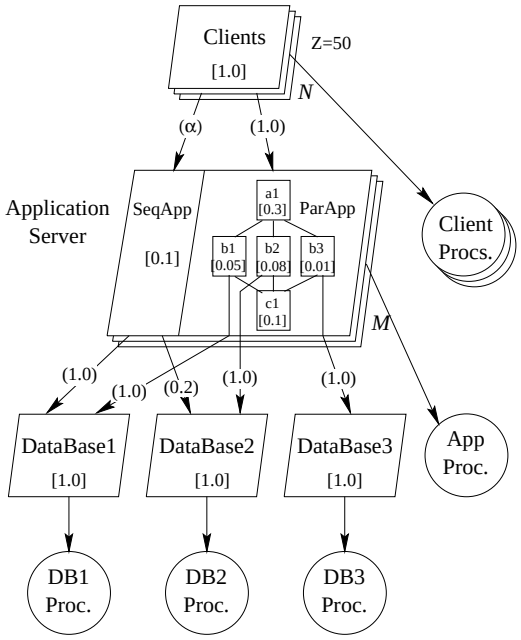


Figure 8.2: Layered Queueing Network for the business client-server system shown in Figure 8.1. The Application task has two different service types, labelled *SeqApp* and *ParApp*. The ParApp service type consists of a single thread of control that forks into three independent threads, then joins again. The notation developed in the previous chapter is simplified (the AND forks and joins are implied) to save space.

executing a unit of work in each cycle. In each cycle there is 1.0 seconds of CPU demand on the client's local processor, α requests for the first service, labelled *SeqApp*, and one request for the second service, labelled *ParApp*, which has internal parallelism. Each request for service is blocking, so the Client waits for it to be completed. The SeqApp service of the Application Server is a typical sequential program. It demands 0.1 seconds of CPU time on its processor, and makes an average of 1.0 and 0.2 requests to Databases 1 and 2, respectively. The number of requests is geometrically distributed.

The service ParApp of the Application Server makes requests in parallel to the three Database servers, and this is represented in the model as follows. When a request arrives at the Application Server for ParApp, it triggers the first activity at the top of the activity graph (indicated by a rectangle named a1 in Figure 8.2 and labelled with a CPU demand of 0.3 units). It then forks three parallel threads, indicated by the arcs coming out of the bottom of the first rectangle and the rectangles labelled b1, b2 and b3 with CPU demands of 0.05, 0.08, and 0.01 respectively; these threads make requests to the three Database servers, indicated by the arrows going from the boxes to the servers. When the parallel parts are completed they join and execute the last activity, labelled c1 with 0.1 unit of CPU demand.

The Application Server is also multi-threaded in the sense that it can initiate additional services for other clients, while its busy threads are waiting for responses from Databases, up to a limit of M concurrent threads. These M main threads all run on the single App-Proc processor, and execute whichever service (SeqApp or ParApp) is requested next. The Database servers are assumed here to be single threaded servers on a single processor each, which is simplistic but focuses attention on the parallelism issue; the model can instead accommodate any configuration of multithreaded, parallel and multiprocessor database servers as well, as will be shown in Section 8.3.

The performance is governed by a complex combination of resource contention and parallelism. When the Application Server is multithreaded its threads may contend for access to the Databases; threads executing SeqApp make a different pattern of Database

accesses, than those of ParApp. Conversely, when it is single threaded, the Application Server is held as a simultaneous resource during a Database server access. A larger value of α increases the contention for Databases 1 and 2, so the duration of parallel branches b1 and b2 increases also through contention. This delays the synchronization, and keeps the Application Server threads busy longer, waiting.

8.2 Solutions for Parallelism

Parallelism affects the solution of the layered queueing model in two ways. First, if a server has parallel branches, its service time includes the delay from the fork to the join, denoted t_{FJ}. Second, in the layers below the parallel server, the parallel part creates an additional customer entity for each active branch. The response times of these entities provides the branch delays used to calculate the join synchronization delay. If we just solved these layered models using standard queueing algorithms, we would have the CD approach of [51]; this technique is called CD$'$ when it includes the three-point delay approximation.

There is an *overlap* effect on contention that was not considered in the CD method of [51], which has motivated the "Compensation" aspect of Compensated Complementary Delays (CCD) developed here. The sibling parallel sub-threads of a single server thread are all launched at once at a fork operation, and this increases their contention at the beginning of the operation. Then as they complete one by one the contention level drops until the last one has no contention from its siblings. The contention will be adjusted for the amount by which the sibling sub-threads overlap each other in time.

The probability of having to contend with one of these sibling sub-threads will be multiplied by a compensation factor reflecting the probability that sub-threads are active at the same time. Since it is only the sibling sub-threads which are synchronized to begin with, there is no compensation for overlaps coming from other parallel sections or from other threads executing the same service. The impact on accuracy will be probably be greatest

where a single parallel section in a single-threaded server dominates a system.

8.2.1 Contention Generated by Parallel Activities

Figure 8.3 shows the activities and their interaction timeline of the ParApp service from Figure 8.2 along with their temporal relationship to the chains and active threads used by the MVA solution.

When an application service forks a set of parallel branches, each branch is represented by a separate customer chain in the lower layer submodel. For example, Figure 8.3(c) shows chain 1 representing the main thread, which executes activities a1 and c1 of the ParApp service and also executes the SeqApp service when requested, together with chains 2, 3, and 4 representing the threads that fork and execute activities b1, b2 and b3. The delay time (think time) of these chains is the mean time from the join to the next fork, for each main thread instance. This is a complementary delay. The chain populations are the same as the population of chain 1, which is the number M of main threads in the server.

The mean-value equations of contention by members of chains 2, 3 and 4 at queues in a lower level of servers must be adjusted to account for their synchronization at the fork. Mak and Lundstrom did this in [84] by finding the overlap between the active phases of chains 2, 3, and 4. The *overlap fraction* θ_{kj} is the probability that, when chain k is active, *sibling* chain j is active also. This fraction is

$$\theta_{kj} = d_{kj}/W_j \tag{8.1}$$

where W_j is the mean active lifetime of sub-path j and d_{kj} is the mean overlap time from the fork point, given by

$$d_{kj} = \frac{1}{W_j^{-1} + W_k^{-1}}. \tag{8.2}$$

When there are two related fork-joins, one nested in the other, then a branch of one also

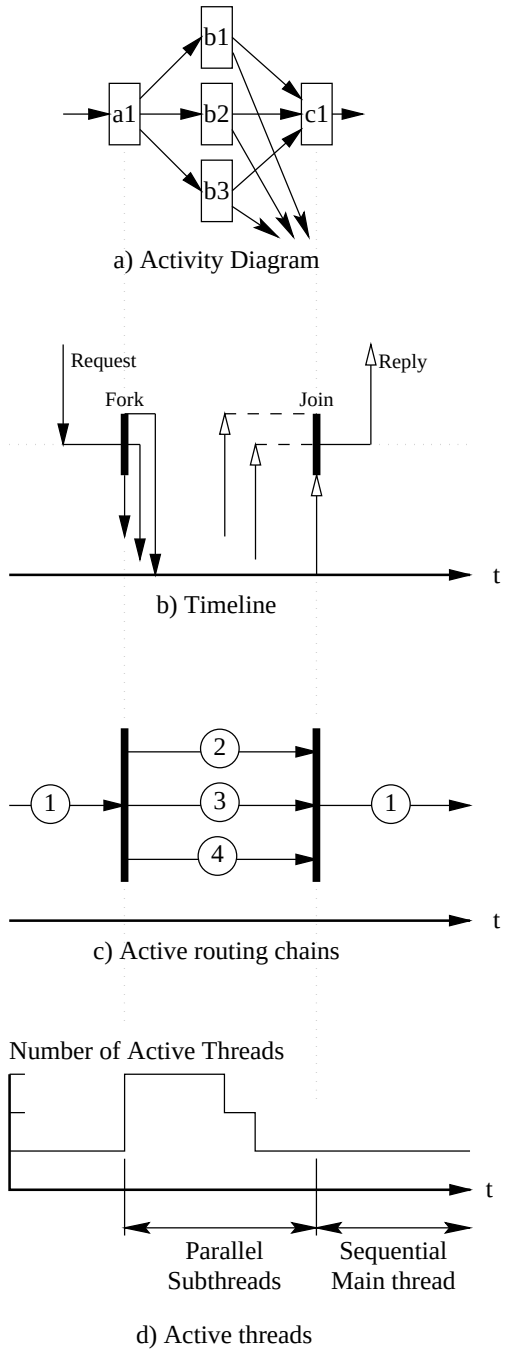


Figure 8.3: Activity graph - customer chain relationship for the parallel application from Figure 8.2. Note that even though there are four distinct chains, at most three are active at any time.

has an overlap with a branch of the other. This is calculated by determining the probability p_{kj} of any overlap at all, given by [84]:

$$p_{kj} = 1 - \Pr(E_j < S_k) - \Pr(E_k < S_j). \tag{8.3}$$

This calculation is based on the delay from the common synchronization point at the outermost fork; S_j denotes the time when j can start executing and E_j denotes the time when j finishes. Then

$$\theta_{kj} = \frac{p_{kj} d_{kj}}{W_j}. \tag{8.4}$$

Mak and Lundstrom considered only a queueing network with one customer in each chain. Since this work has servers with multi-threaded servers, there can be additional members of chain j which are not siblings of the member of chain k. For these others, the overlap is different. For example, a particular active member of chain 2 from Figure 8.3 is synchronized only with one member of chain 3 and one of chain 4, its own siblings that were forked with it from the same chain-1 parent. Other chain-1 customers fork at unrelated moments and spawn members of chains 1, 2 and 3 which have no special temporal relationship to this particular member of chain 2. Assuming the forking times of non-sibling chains j and k are independent, the *unsynchronized overlap fraction* for members of chain j and k, which we can call θ^*_{kj}, is just the fraction of time chain j is active:

$$\theta^*_{kj} = \lambda_k W_k \tag{8.5}$$

where λ_k is the throughput for chain j.

Over the N_j members of chain j, the average overlap fraction is found from 8.4 and 8.5.

$$\overline{\theta_{kj}} = \frac{\theta_{kj} + (N_j - 1)\theta^*_{kj}}{N_j}. \tag{8.6}$$

In MVA [107] a mean number L_{mj} of chain j customers is found in queue m at an arrival instant of a chain k customer. MVA assumes no temporal relationships between thread activations, and therefore corresponds to an unsynchronized overlap probability of θ^*_{kj}. If the average overlap is $\overline{\theta_{kj}}$ instead, we assume that the mean number of customers increases in proportion to $\overline{\theta_{kj}}$; the mean number L_{mj} then is replaced by $(\overline{\theta_{kj}}/\theta^*_{kj})L_{mj}$. With multiple chains $j = 1..K$ we obtain

$$W_{mk}(\mathbf{N}) = D_{mk}\left[1 + \sum_{j=1}^{K} \frac{\overline{\theta_{kj}}}{\theta^*_{kj}} L_{mj}(\mathbf{N} - \mathbf{e}_k)\right] \tag{8.7}$$

where $W_{mk}(\mathbf{N})$ is the total residence time per response cycle of chain k at station m, in a system with a chain population vector $\mathbf{N}$, D_{mk} is the total service demand of chain k at m, and $L_{mj}(\mathbf{N} - \mathbf{e}_k)$ is the mean number of chain-j customers found at an arrival instant (as computed by the usual MVA method [107]). L_{mj} is further approximated in the solver in the Linearizer algorithm.

Finally, there are pairs of customers that have zero overlap such as the parent customer in chain 1 with any of its children in chains 2, 3 and 4. For these pairs of customers $p_{kj} = 0$, and thus $\theta_{kj} = 0$.

8.2.2 Approximation for Join delays

The difficulty with finding join delays is that an accurate calculation requires the full probability distribution for the branch lifetimes, which is not available from a mean value analysis (MVA). The usual approximation, made for instance by Mak and Lundstrom [84] and by Heidelberger and Trivedi [51], is that branch delays are exponential, in which case there is a simple analytic formula for the join delay which depends only on mean values. However this may give quite large errors. To eliminate these, Chu, Sit and Leung [21] demonstrated a gamma-distribution approximation which requires fitting several distributions and then integrating numerically over the final distribution to obtain the mean and variance. This

works well but is computationally expensive.

This work used a new "three-point" approximation due to Jiang [62] to compute the mean and variance of the join delays, based on the mean and variance of the branch delays. The distribution for each branch delay is approximated by three point probabilities which match the mean and variance, and the rest of the analysis is exact. The algorithm for determining the locations t_{ij} of the three points and the values $a(t_{ij})$ for the probabilities (from the mean and variance of a delay t_i) is given in [62]. Figure 2.4 illustrates the point probabilities $a_i(t)$ for one branch and the approximation $A_i(t)$ for the probability distribution function for that branch. For the overall delay it is $A(t)$, the product of the $A_i(t)$. The probabilities $a(t_k)$ for the fork-join delay are found for the points t_k where $A(t)$ jumps, and the mean and variance follow. The discrete approximation makes the calculation fast.

The three-point approximation was tested in [62] with hundreds of randomly generated test cases. When the squared coefficient of variation of each of paths involved in the join was less than or equal to one, the mean absolute error in overall delay was less than one percent, and when the squared coefficient of variation was between one and nine, the mean absolute error was less than four percent. In larger task graphs the errors tended to be smaller. This accuracy is adequate for practical predictions.

8.2.3 Complexity

The layered model algorithms are described as "scalable" because their computational demands and storage go up as low-order polynomials in the size parameters of the model, such as the number of tasks, services and request arcs [152]. Each of the layer submodels is solved using the Linearizer approximate MVA algorithm, therefore the dominating complexity parameter is cubic in the number of chains [12] in a given submodel.

With parallel services there is an additional parameter B, the largest number of parallel branches in any one server. B enters the time complexity through the overlap and the

branch and join delay calculations, (which are linear in B) and through the increase in the number of chains in some submodels by a number proportional to B. Similar arguments apply to the space requirements. Thus the layered solution algorithm incorporating the fork-join approximations remains scalable.

8.3 Results

A useful robust approximation must be sufficiently accurate (a few percent error), and also be scalable to large systems. This section considers three examples, first to illustrate accuracy using a small example, second an example with greater system complexity and third to compare the accuracy to [51]. Each of the systems were compared to detailed simulations which modeled the semantics of the layered queueing networks.

8.3.1 Application Server Example

The example given in Figure 8.2 was analyzed with one thread and four threads in the Application Server, and with variations in the relative frequency of requests to the sequential service SeqApp and the parallel service ParApp, as determined by the SeqApp request rate α. The results in Tables 8.1(a) and 8.1(b) show the client throughput values by simulation, with confidence intervals, and the predicted throughput values and errors by CD$'$ and CCD methods. The prediction error is measured by the percentage relative error in the predicted client throughput, $\epsilon = \frac{\text{sim.}-\text{approx.}}{\text{sim.}} \times 100$.

With only one thread in the server, CCD is roughly four to eight times better. The advantage becomes large as the traffic levels increase (α increases) and the Application Server becomes more of a bottleneck. However with four threads, CCD is only a little better and only for small α. The difference is that with one thread all the sub-paths in the Application Server are synchronized and have overlap factors which are different from the default, while with four threads the unrelated and unsynchronized sub-paths dominate.

Also, as α increases, a bottleneck appears at Database1 and this dominates the system solution.

	Simulation		Analytic Approximations			
			CD′		CCD	
α	λ	$\pm 95\%$	λ	$\epsilon(\lambda)$	λ	$\epsilon(\lambda)$
1	0.48	0.0020	0.43	-10.06	0.50	2.76
2	0.55	0.0034	0.49	-11.48	0.56	1.82
4	0.62	0.0016	0.56	-9.97	0.63	1.52
7	0.67	0.0051	0.62	-7.95	0.68	0.96
10	0.69	0.0067	0.65	-6.12	0.70	1.10

(a) $M = 1$ thread

	Simulation		Analytic Approximations			
			CD′		CCD	
α	λ	$\pm 95\%$	λ	$\epsilon(\lambda)$	λ	$\epsilon(\lambda)$
1	0.90	0.0030	0.87	-4.38	0.88	-2.96
2	0.96	0.0047	0.93	-3.02	0.93	-2.58
4	0.98	0.0036	0.97	-0.76	0.97	-0.74
7	0.99	0.0019	0.99	0.37	0.99	0.37
10	0.99	0.0025	1.00	0.46	1.00	0.46

(b) $M = 4$ threads

Table 8.1: Results for the Application Server with M threads (throughput, predicted throughput and its percentage error).

8.3.2 Extended Client-Server Example

To show how the solution behaves in a larger client-server model, parallelism was incorporated into the lower level server, Database 3 in the previous example, as shown in Figure 8.4. It now has parallel paths within it, and additional disks are accessed in parallel. This system was then analyzed by simulation and by the CD′ and CCD solvers over a range of client population N from 10 to 100, and a range of threads in Application Server from 10

to 30. The computing time for each solution was about 20 seconds each for CD′ and for CCD (independent of the number of customers), and ranged from 20 to 140 minutes for simulation

The CCD results in Figure 8.5 show the Client throughput and response time rising as the number of clients is increased. The additional Application Server threads (beyond 10) make no difference until the clients approach 100. At this point, ten threads give a bottleneck at the Application Server, while 30 threads push the bottleneck down to its processor.

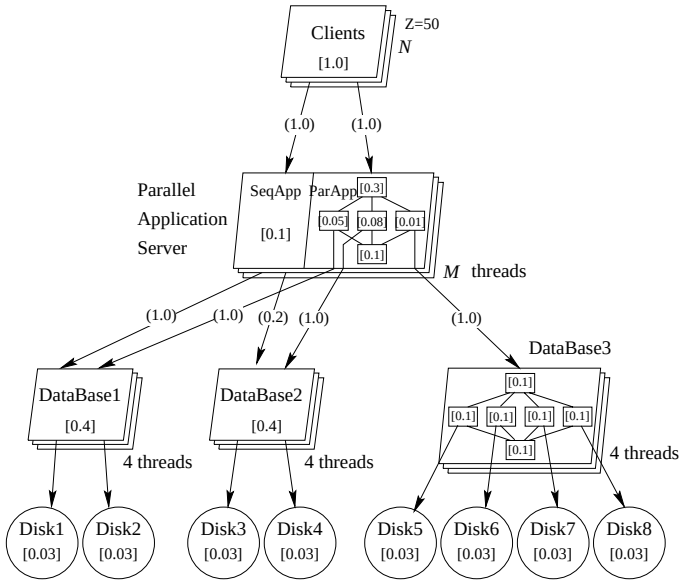
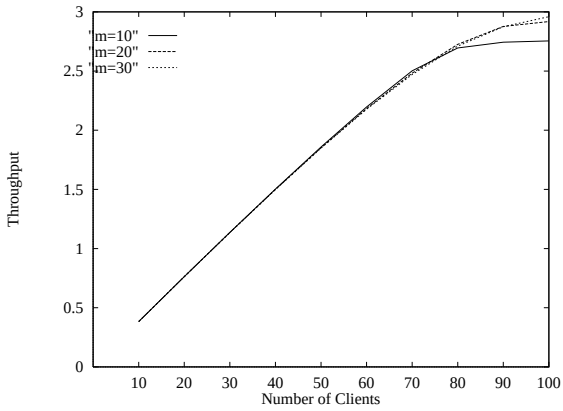
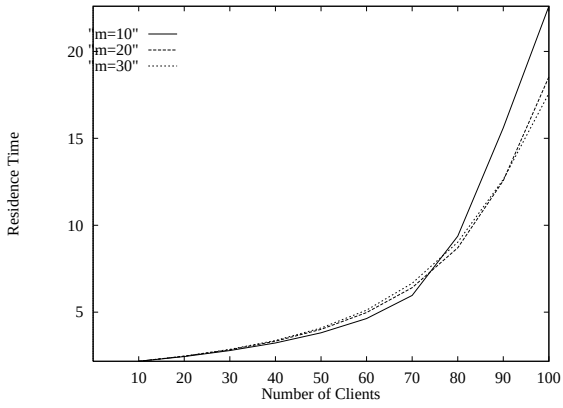


Figure 8.4: Layered Queueing Network for an extended business client-server system with parallelism in one of the databases.

When we compare the CD′ and CCD prediction errors in Tables 8.2 through 8.4 we find that they are much the same. In this system with so many threads the effect of overlap is buried by the large number of unsynchronized chains due to the other threads. The errors are largest at the bottom of Table 8.2, due to errors in the fork-join delay approximation, which clearly still needs to be improved. The mean error over all the runs, -0.79% is within



(a) Throughput



(b) Waiting Time.

Figure 8.5: Results for the extended example, with varying threads in Application Server.

the 95% confidence intervals of ±2% of the "true" simulation results.

N cust	Simulation		Analytic Approximations			
			CD′		CCD	
	λ	±95%	λ	$\epsilon(\lambda)$	λ	$\epsilon(\lambda)$
10	0.19	0.0007	0.19	-0.07	0.19	-0.07
20	0.38	0.0025	0.38	-0.74	0.38	-0.75
30	0.57	0.0028	0.57	-0.73	0.59	-0.73
40	0.76	0.0037	0.75	-0.93	0.75	-0.94
50	0.94	0.0012	0.93	-1.45	0.93	-1.45
60	1.12	0.0030	1.10	-1.78	1.10	-1.76
70	1.28	0.0054	1.25	-2.40	1.25	-2.35
80	1.42	0.0032	1.34	-5.12	1.35	-4.99
90	1.52	0.0065	1.37	-9.75	1.37	-9.59
100	1.55	0.0055	1.37	-11.64	1.37	-11.54

Table 8.2: Throughput and mean relative error results for the extended example, with 10 threads in Application Server. Simulation results had 95% confidence intervals of ±2%.

The value of CCD is now clear. Compared to CD′, CCD gives much better accuracy in certain cases, and in all cases, it contributes a negligible extra cost. It gives the extra accuracy when there is a small number of threads that can fork in parallel in the same server, and when this server is limiting the performance of the system.

8.3.3 Comparison to Decomposition

The 228 test cases described in Heidelberger and Trivedi [51] were recast as layered models to compare the new algorithm to the decomposition method of [51]. The test cases have one to five client tasks which cycle. Each client task has a single parent activity which forks either two or three child sub-paths and waits for the children to complete, then cycles again. All the sub-paths make requests to a set of disks. For the two-child test, 108 different cases were analyzed with one or two CPUs, 1, 3, or 5 clients, and different visit and service time values. For the three-child test, 120 cases were analyzed with one or three CPUs, 1, 3 or 5 clients, and different values of other parameters, as described in [51].

N cust	Simulation		Analytic Approximations			
			CD′		CCD	
	λ	$\pm 95\%$	λ	$\epsilon(\lambda)$	λ	$\epsilon(\lambda)$
10	0.19	0.0007	0.19	-0.07	0.19	-0.07
20	0.38	0.0030	0.38	-0.41	0.38	-0.41
30	0.57	0.0027	0.57	-0.49	0.57	-0.49
40	0.76	0.0033	0.75	-1.16	0.75	-1.17
50	0.94	0.0037	0.93	-1.61	0.93	-1.60
60	1.12	0.0022	1.09	-2.53	1.09	-2.52
70	1.28	0.0025	1.24	-3.10	1.24	-3.07
80	1.42	0.0039	1.36	-4.01	1.36	-3.96
90	1.51	0.0040	1.44	-4.81	1.44	-4.73
100	1.55	0.0044	1.46	-6.25	1.46	-6.15

Table 8.3: Throughput and mean relative error results for the extended example, with 20 threads in Application Server. Simulation results had 95% confidence intervals of $\pm 2\%$.

N cust	Simulation		Analytic Approximations			
			CD′		CCD	
α	λ	$\pm 95\%$	λ	$\epsilon(\lambda)$	λ	$\epsilon(\lambda)$
10	0.19	0.0007	0.19	-0.07	0.19	-0.07
20	0.38	0.0010	0.38	-0.36	0.38	-0.36
30	0.57	0.0029	0.57	-0.74	0.57	-0.74
40	0.76	0.0028	0.75	-1.28	0.75	-1.28
50	0.94	0.0039	0.92	-1.68	0.92	-1.68
60	1.12	0.0047	1.09	-2.56	1.09	-2.56
70	1.28	0.0012	1.24	-3.44	1.24	-3.43
80	1.42	0.0016	1.35	-4.48	1.35	-4.47
90	1.51	0.0019	1.44	-5.09	1.44	-5.04
100	1.55	0.0044	1.48	-4.88	1.48	-4.82

Table 8.4: Throughput and mean relative error results for the extended example, with 30 threads in Application Server. Simulation results had 95% confidence intervals of $\pm 2\%$.

Table 8.5 shows the mean absolute percentage error, the 90th percentile and the largest sample of $|\epsilon|$, denoted by Ave, 90% and Max respectively over the 108 cases in each of the two-child (Ch = 2) reports and the 120 cases in each of the three-child (Ch = 3) reports. The columns labelled "Dcmp" and "CD" list the results given in [51] for decomposition and for CD respectively. The results in the columns labelled "CD′" and "CCD" were found by comparing simulations (with confidence intervals of 0.5% at the 95% level) to predictions made with CCD and with CD′. The comparisons between results for Ave, 90% and the maximum observed $|\epsilon|$ are very similar so they will be discussed together.

The results for CD and CD′ differ slightly because the simulations were redone, and because CD′ used the three-point approximation to find the join delays. On these tests they are about the same, with some errors larger in one column, some larger in the other.

The last column describes the new CCD results. Compared to decomposition in column one, the errors are three to four times greater for CCD. While this is disappointing, CCD still has quite good accuracy for practical purposes (about 2%–3% in the Ave measure). Compared to CD′ (without the overlap compensation), CCD is two to three times better, and this improvement can be valuable. Thus, the compensation is a big gain.

The second table (8.5) gives more detail on the comparison between CD′ and CCD from the new results, by showing the mean and standard deviation of ϵ (without taking the absolute value). Here $\epsilon(\lambda)$ stands for errors in the throughputs calculated for all servers in the models. Now we can see that most of the throughput error is due to systematic bias which depends on the number of CPUs or parallel branches. The spread, indicated by σ, for CCD is much smaller.

8.4 Conclusions

The examples demonstrate the power of the layered modelling framework to provide a convenient platform for defining models with parallelism. They can have parallelism in

CPU	Ch Act	Meas	From [51] Dcmp	CD	New Approx CD′	CCD
1	2	Ave	0.5	5.8	5.83	2.74
		90%	1.0	10.0	10.01	5.08
		Max	2.7	12.7	17.63	11.5
2	2	Ave	0.6	4.9	3.98	2.10
		90%	1.2	8.7	7.92	3.84
		Max	2.2	10.3	11.64	11.5
1	3	Ave	0.7	5.6	5.19	2.86
		90%	1.4	7.4	8.58	4.37
		Max	4.2	15.3	11.39	7.07
3	3	Ave	1.0	2.5	4.07	2.15
		90%	2.0	8.6	7.53	3.53
		Max	6.8	10.7	9.12	8.34

Table 8.5: Magnitudes of percentage throughput errors in the 228 test cases studied by Heidelberger and Trivedi [51]. Ave is the mean of the absolute relative errors, stated as percentages; 90% is a 90th percentile and Max is the maximum error in a group of samples.

CPUs	Child Activites	Measure	$\epsilon(\lambda)$ CD′	CCD
1	2	Mean	-5.84	1.84
		σ	0.21	0.04
2	2	Mean	-3.65	1.97
		σ	0.19	0.05
1	3	Mean	-4.49	1.44
		σ	0.23	0.01
3	3	Mean	-2.47	1.51
		σ	0.28	0.01

Table 8.6: Bias and spread of the percentage errors in throughput from the same cases as Table 8.5.

different servers and in different parts of the system, and can represent layered contention effects and the effects of multi-threading.

Compared to decomposition, CCD has adequate (though worse) accuracy combined with a manageable cost which remains feasible for large numbers of threads and branches, while the cost of decomposition rises exponentially. We can describe this as a robust combination of accuracy and cost.

The CCD approximation has been found to give quite good accuracy for practical systems, typically with errors of a few percent. It was tested over many hundreds of examples with different parameters. CCD is intermediate in accuracy between decomposition and a CD approximation which is uncompensated for chain overlap. Unlike decomposition, it can be economically scaled up to large systems, in the sense that its time and storage demands are low-order polynomials in the size parameters of a model. CCD is robust in cost and accuracy, in that it avoids both the larger errors of complementary delay methods and the exponential cost escalation of decomposition.

Compensation in CCD was found to give an error reduction of up to a factor of two in the throughput prediction over CD$'$. The greatest advantage occured in those cases where the parallel server has few threads and is a bottleneck to the system.

In other cases complementary delays (CD$'$) is almost as good. Therefore it may also have a place as a technique giving useful accuracy in many practical cases.

The examples do not exhaust the possible applications of the CCD approximation. For instance they do not include asynchronous RPCs and prefetches, but these can be modelled by forking one thread to execute the remote request while the main computation goes on in a second parallel child thread, up to a join where the result is needed. Similarly, the models do not consider network latencies, but these delays can be added easily.

The evaluation has been restricted to intra-task fork-joins. Some experiments with inter-task fork-joins had unacceptably large errors, likely caused by the correlation of arrivals at the join.

Chapter 9

Case Study of the Linux 2.0 NFS Implementation

One of the most commercially successful and widely used available remote file system protocol is the Network File System (NFS) [135, 83, 121, 141], designed by Sun Microsystems. It was designed as a client-server application; the client imports file systems from server machines and makes remote procedure calls to perform operations such as `read()` and `write()`.

This chapter uses the layered queueing network model to study the performance of the Network File System as implemented in the Linux Version 2. kernel. This implementation of NFS uses the NFS-V2 protocol [135] over UDP.

First, the layered queueing network model of NFS is described and validated against a live system consisting of several 200 MHz Pentium II processors interconnected with a 100 Mb ethernet. Then, the model is perturbed to study the effects of a higher speed network, a different workload, and efficiency improvements to the implementation.

The material in this chapter was mostly published in [43].

9.1 The Network File System

The standard NFS server was designed to be stateless so that server operations could be simplified. The server need not keep track of what each client is doing and, in the event of a crash, no special recovery operations are needed. Because of this design each NFS RPC operation is idempotent. If a client detects a failure, it simply re-issues the request. For operations which modify the data on the server, the server's reply denotes success or failure. This behaviour implies that modifications to files are committed to stable storage before a reply is generated. Because disks are relatively slow devices, this behaviour has also led to performance problems. Solutions to this problem include *write gathering* [67], battery-backed disk RAM caches [87] and not performing synchronous writes at all.

9.1.1 Linux NFS operation

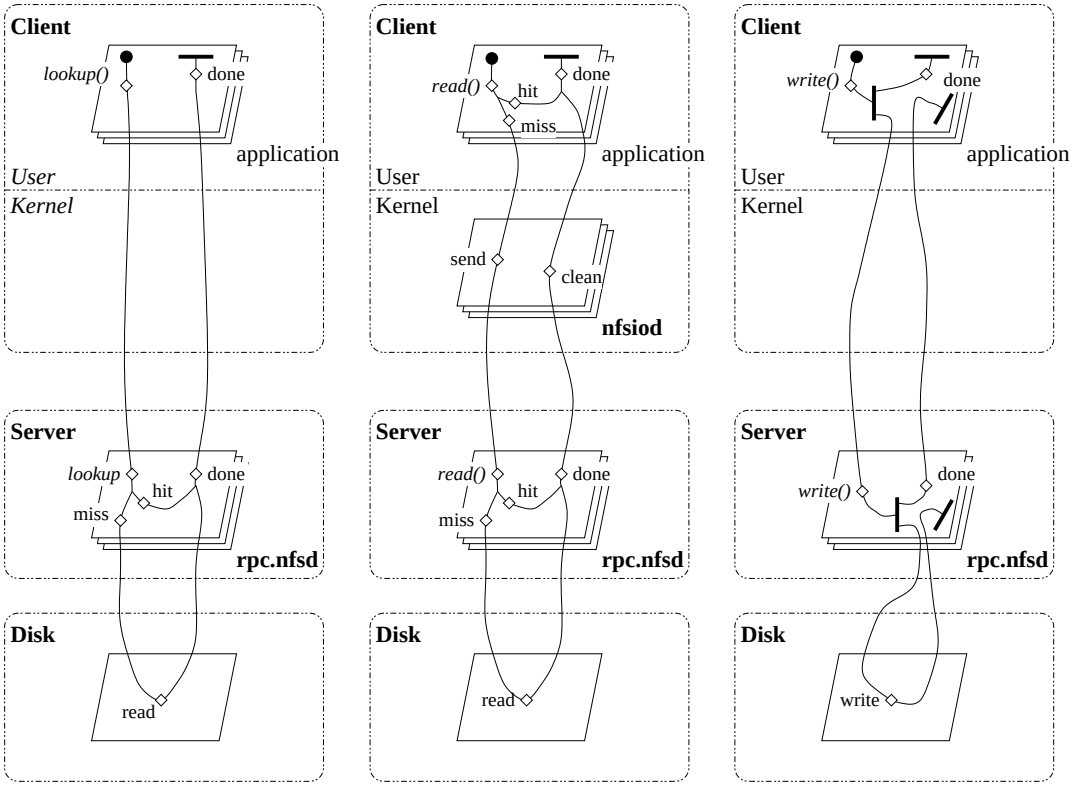
Figure 9.1 shows "use case maps"[1] [9] of the NFS operations used to derive the model in this paper. (The maps shown here are simplified in the sense that many of the operations may loop; only one occurrence is shown here.) These use cases are then parameterized and turned into a layered queueing network model.

Only three NFS operations were modelled: `lookup`, `read` and `write`. Lookup and read dominate the workload. Write, while not a large component of the workload, is also modelled because it has been a major source of performance problems. The sections that follow describe these operations more detail. Section 9.2 describes the actual layered queueing network model.

Lookup

The lookup operation, shown in Figure 9.1(a), is used to locate a file on the server and return a "file handle" to the client. The file handle is used by the client for subsequent file

[1] A use case map shows an execution trace of a program.



(a) Lookup (b) Read (c) Write

Figure 9.1: Use Case Map of the primary NFS operations. Processes and disks are shown as parallelograms. The filled circle represents the start of the operation and the filled rectangles the end. The arc between the start and end symbols represents the temporal ordering of the operations, shown as diamonds in the figure.

operations such as `read()` and `write()`.

On the server side of the lookup operation, the *rpc.nfsd* daemon will check its internal cache to determine whether a file handle exists for the requested file. If not, the daemon will attempt to create a new file handle by looking for the requested file in the server's file system. Searching for the file may result in several disk accesses as directories are read while searching for the file.

Read

Figure 9.1(b) shows the UCM for an NFS read operation. When the application issues a read, the client file system checks its cache to determine if the requested data is already present. If not, the file system requests blocks in 4K byte pages from the file server. Each read request blocks the client process while the server processes the request. Since each read request sent to the server blocks the requester, four *nfsiod* processes are used as proxies. If an *nfsiod* process is available, the client can issue the read request to the server by way of the proxy and continue to execute itself. Typically, *nfsiod*'s are invoked when the client issues a read request larger than the system's *page size* so that multiple read requests can be issued in parallel.

When a read request arrives at the server, the server's buffer cache is checked to see if the page is present. If so, the server immediately replies to the client with the data. If the page is not present, the server makes a blocking read request to the disk; once the data is read, the server can reply.

The server's file system also issues read-ahead requests asynchronously to the disk. A subsequent read request may then find the data in the buffer cache, in which case the server replies immediately, or the server may find the read request pending, in which case the server will block.

Write

Figure 9.1(c) shows the UCM for a write request. When the client application writes to an NFS file system, the request is immediately forwarded to the server, shown by the AND-Fork in the UCM. This write occurs asynchronously; the client does not wait for the server to reply. Rather, the client's file system waits for the reply and then frees the buffer[2].

At the server, write requests are also forwarded to the disk immediately. Again, the write occurs asynchronously so that the *rpc.nfsd* process can reply immediately to the client. Note that this approach, while having significant performance benefits as demonstrated later, does not obey NFS semantics.

9.1.2 Related Work

The performance of NFS has been studied extensively in the past, both empirically [95, 47, 46] and using performance models [144, 96]. These studies have shown that network delays (in particular, when UDP packets are lost resulting in RPC timeouts and retries) and synchronous writes at the server both impact performance. Improvements to the protocol [95], the use of TCP instead of UDP, performance optimizations at the server [67, 68], and special hardware [87] have all been used for significant performance improvements.

9.2 Layered Queueing Model of NFS

This section describes a Layered Queueing Network model of the Linux 2.0 NFS implementation, shown in Figure 9.2. The throughput predicted by the model is compared against a live system. The model is divided into four parts: the client, the server, the disk on the server, and the network. The model is solved both analytically using the layered queueing network solver, and with simulation. The analytic results are compared to simulation and

[2]The buffer is not freed until the reply arrives in case the client must re-issue the write request due to a server crash or a dropped UDP packet.

to the live system to show the accuracy of the model and of the analytic technique. The execution time of the benchmark, the analytic solver, and the simulator are also compared.

To speed file operations, Linux uses a buffer cache to store pages of recently accessed files. During read operations, the file system *reads ahead* so that read requests from the user access the data from the buffer cache without waiting for the disk. Write operations are processed asynchronously so that the user process does not have to wait for the disk to complete its operation.

To incorporate the asynchronous read and write effects, the LQN model incorporates *two-phase multi-servers* [41] to model the buffer cache. The number of copies of the multi-server denotes the number of buffers in the system. The utilization of the multi-server denotes the minimum number of buffers needed. If the multi-server is fully utilized, the queueing delay represents the time needed for the buffer cache to acquire a free buffer.

The asynchronous reads and writes can also be modelled as an open class in a mixed queueing network model. However, this method would not incorporate contention delays for buffers.

The sections that follow describe the workload and each of the four parts of the model in detail.

Workload

The workload for this performance study was generated by the nfsstones benchmark [127]. Other NFS benchmarks exist such as LADDIS [145] and SPECsfs [137]), but these benchmarks execute the NFS protocol directly, thus bypassing the the client's file system altogether. As the intent of this study is to include both the client's and the server's behaviour, benchmarks like LADDIS are not satisfactory.

Table 9.1 shows the workload measured by Sun Microsystems, the workload generated by the nfsstone and LADDIS benchmarks, and the workload used here generated by a modified nfsstones (the file size parameter was increased by a factor of ten). This change

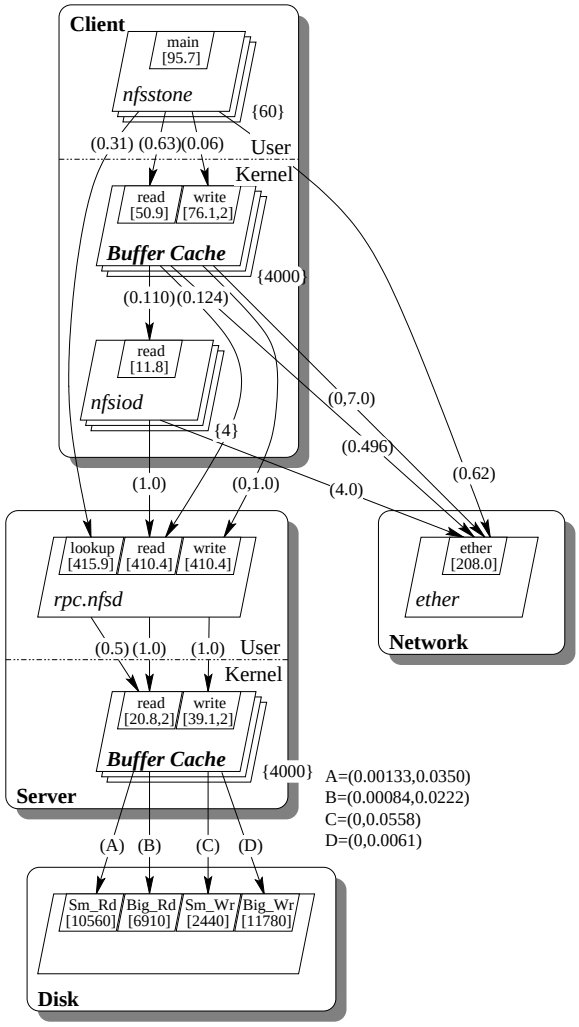


Figure 9.2: Layered Queueing Network of principle NFS operations. The large parallelograms represent tasks and resources. The smaller parallelograms within the tasks are entries and serve to differentiate service. The shaded boxes group the tasks by processor.

was necessary because main memory sizes are much larger today than they were in 1989; without the change the benchmark would only exercise the client's cache.

NFS Op.	Percentage of Total			
	Sun [120]	nfsstone [127]	LADDIS [145]	LQN Model
lookup	50	53.0	34	26.3
read	30	32.0	22	63.2
readlink	7	7.5	8	3.7
getattr	5	2.3	13	
write	3	3.2	15	6.4
create	1	1.4	2	
readdir			3	
statfs			1	
remove			1	
setattr			1	

Table 9.1: NFS Operation Mix.

Nfsstones works by forking 60 clients that perform write, then a set of reads. Operations on disk are not uniformly distributed, which may skew results.

Client Submodel

Table 9.2 shows the mean user and system times and their standard deviations of the *nfsstones* and *nfsiod* processes for twenty runs on the live system. The service time for *nfsstones* was found using the Linux time utility. Service times for *nfsiod* were found by examining the process information found in the `/proc` file system. The system times for *nfsstones* and *rpc.nfsd* was further refined by using the strace utility to trace the number of request and the fraction of total system time used by each system call. All of the system time for read and write requests were allocated to the *buffer cache* LQN tasks.

The request rates from *nfsstones* to the client's *buffer cache* and to the server's *rpc.nfsd* were set based on the work load found in Table 9.1. The request rate to the *ether* task

Process	Calls		Time			
	Function	N	$\overline{t}$ (S)	σ_t	% of Total	per call (μS)
nfsstones	*user*	95010	0.617	0.0557	100.00	6.494
	system		12.232	0.3436	100.00	
	access	25003	7.267		59.41	290.633
	read	60000	3.052		24.95	50.873
	readlink	3498	1.025		8.38	293.122
	write	6001	0.469		3.83	78.136
	open	508	0.196		1.61	386.486
	lseek	18048	0.108		0.88	5.966
	getpid	25003	0.080		0.65	3.188
	other	1344	0.035		0.28	25.673
nfsiod	*system*		0.0784	0.0036		

Table 9.2: Client Service Times.

representing the network is set to twice the value of the request rate to the lookup entry of *rpc.nfsd* because one ethernet packet is needed for both the request and the reply to the server.

The request rates from the client's *buffer cache* to its lower-level servers for read requests was set by finding the product of the buffer cache miss ratio, the fraction of requests handled by the *nfsiod* processes and the number of 4K pages read per 8K read request issued by the client. This information was found by instrumenting the kernel and later verified by comparing the strace data from the *nfsstones* and *rpc.nfsd* processes. Read requests to *rpc.nfsd* are issued by way of the ethernet. Each 4K read request made directly from the *buffer cache*, or indirectly from the *nfsiod* process, results in four packets on the ethernet; one for the request and three for the reply.

Each write request made by the client is passed directly to the server by way of the buffer cache. However, unlike reads, writes take place from phase 2 so that the client does not block waiting for the request to complete. Further, write requests occur in 8K chunks, thereby requiring seven packets on the ethernet.

Server Submodel

The server process *rpc.nfsd* is modelled as a single threaded, single phase, multiple entry server. Reads and writes are modelled using their own entries. All other functions are modelled by *lookup*.

The aggregate service time for the server was found by examining the process data in the `/proc` file system after each run; the mean and standard deviation are shown in Table 9.3. User time is assumed to be distributed equally among the requests (the bulk of this time is in the common RPC code). The system times was further partitioned from the service time information collected by strace. System times for reads and writes are pushed down to the *buffer cache* multi-server task. The remaining system time is assigned to the *rpc.nfsd* task.

Process	Calls		Time			
	Function	N	$\overline{t}$ (S)	σ_t	% of Total	per call (μS)
rpc.nfsd	*user*	44670.2	7.859	0.1113	100.00	175.934
	system		10.962	0.195	100.00	
	select	44547.1	8.0620		73.54	180.979
	lstat	42078.3	1.012		9.23	24.050
	sendto	48666.6	0.740		6.75	15.210
	recvfrom	44670.2	0.360		3.29	8.069
	read	13518.6	0.309		2.82	22.845
	write	6000.2	0.247		2.25	41.101
	time	24735.7	0.108		0.99	4.372
	readlink	3498	0.046		0.42	13.204
	lseek	19519.2	0.035		0.32	1.812
	truncate	492	0.028		0.26	57.348
	other	353	0.014		0.13	40.236

Table 9.3: Server Service Times.

Each read and write request results in one nested request to *buffer cache*. The *rpc.nfsd* task is blocked during this interval. Some of these requests will result in further nested requests to the disk. These requests are described in the next section. It is assumed that

requests to *lookup* reference the buffer cache 50% of the time.

Disk Submodel

The parameters for the disk submodel are based on:

1. the number of requests to the disk for each request made to the server's *buffer cache*,
2. the mean request size from the buffer cache, and
3. the mean service time for each request.

For reads, the file system tries to *read ahead*; in some cases more blocks are read than are actually required (i.e. during random read requests). In most instances, the read ahead operations are asynchronous; they are treated as phase-two requests. For writes, all operations are treated as phase two; the write system call returns as soon as the data is written to the buffer cache. The buffer itself is "busy" until it is written to disk.

Equations (9.1) through (9.3) are used to determine the request rates from the server's *buffer cache* to the disk. The variables are:

L Logical reads,

B Blocking reads,

R all Read aheads,

A Asynchronous read aheads,

S disk sectors read (1K Blocks), and

P Page size (4K).

$$\eta = 1 - B/L \tag{9.1}$$

$$\alpha = A/R$$

$$\beta = S - PB$$

$$\text{Ph1} = \frac{(1-\alpha)\beta + PB}{PL} \tag{9.2}$$

$$\text{Ph2} = \frac{\alpha\beta}{PL} \tag{9.3}$$

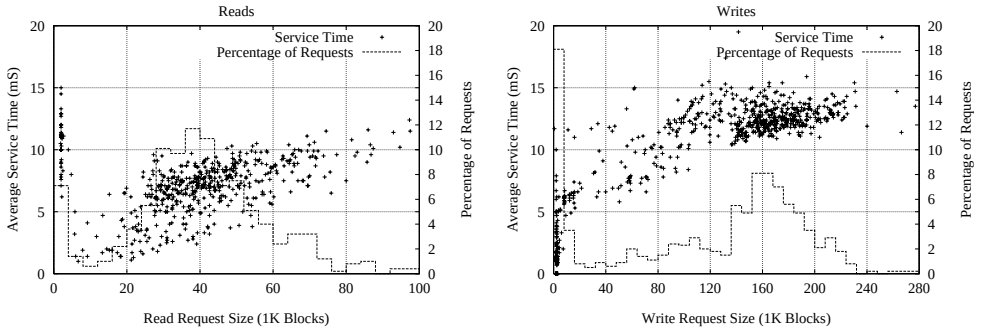
Equation (9.1) is the buffer cache hit ratio. Equations (9.2) and (9.3) are the number of requests made to the disk from phase 1 and phase 2 of the buffer cache respectively for each read request received. These results are further modified by the mean request size, described next. Table 9.4 shows the measured values and the results.

Reads		Read Ahead		Sectors	η	Rate	
L	B	R	A	S		Ph1	Ph2
14117.6	65.5	672.0	650.3	59179.9	0.9953	0.0383	1.0097

Table 9.4: Disk Request Rates.

Disk modelling is complex because of seek times, rotational latency and local caching [117]. For this study a simpler model in which the operation times were grouped in 'buckets' is used instead. The disk service time was measured by adding performance monitoring counters to the kernel and polling these counters while *nfsstones* was running. Figure 9.3 shows disk service times and the histograms of requests. The histograms in Figure 9.3 show that there are a number of requests of less than 4K bytes in size, and a second grouping of requests around 42K for reads and 140K for writes. The wide-range of request sizes arises from the I/O system merging adjacent read and write requests into one big request.

For this performance study, two buckets were used: one for requests less than 4K bytes in size and the other for all other requests. Table 9.5 shows the mean request sizes and service times found from the measurement data. The mean service time for each bucket is used as the entry service time in the performance model. The request rates are found by



(a) Read Service Times (b) Write Service Times

Figure 9.3: Disk Service Times Scatter Plots.

multiplying the phase 1 and 2 rates from Table 9.4 by the fraction of requests made to each bucket, then dividing by the mean bucket size.

Parameter	Service Time (mS)		Size (1K Blocks)		Requests (%)	Calls	
	$\overline{t}$	σ_t	$\overline{\text{size}}$	σ_{size}		Ph 1	Ph 2
Small Read	10.56	1.99	2.05	0.15	7.1	0.00133	0.0350
Big Read	6.91	2.15	42.33	16.07	92.9	0.00084	0.0222
Small Write	2.44	0.36	2.49	1.82	13.9		0.0558
Big Write	11.78	2.33	140.66	56.00	86.1		0.0061

Table 9.5: LQN Model Parameters.

Network Model

The network model used here is from [75, pg. 341], and modified slightly to account for the fact that an ethernet with large packets is about 83% efficient [142]. The following set of

equations are solved for $n = 2$:

$$X(n) = \left[1 - \frac{1}{n}\right]^{n-1} \tag{9.4}$$

$$C(n) = \sum_{i=1}^{\infty} iX(n)(1 - X(n))^i$$

$$= \frac{1 - X(n)}{X(n)} \tag{9.5}$$

$$E(n) = P/B + SC(n) \tag{9.6}$$

where B is the network bandwidth in bits per second (100×10^6), P is the packet size in bits (1518×8) and S is the slot duration and is assigned the value 0.0000512 (from [86]).

For this model, each lookup operation needs 2 packets, one for the request and one for the reply. Reads are 4K bytes in size, therefore each request requires 4 packets. Finally, writes are 8K bytes in size and require 8 packets.

9.3 Results

9.3.1 Model Validation

This section compares the results found by solving the model both analytically using LQNS [40], and from simulation. The predicted throughputs are compared against each other and against the results from the "live" system to compare the accuracy of the model, and to compare the accuracy of the approximation. The run times of the three techniques are also compared.

The live system consists of an isolated network of 200MHz Pentium processors running on a 100Mb switched ethernet. The analytic solutions and simulations were also performed on the same hardware.

The results from the base model shown in Figure 9.2 are presented first. Next, the effect of different service time distributions are described. Finally, the sensitivity of the ethernet

model and cache ratios is shown.

Sources of Error

Errors in the results may arise from a number of sources:

1. the service times may be incorrect from measurement error,
2. the visits from *rpc.nfsd* to the buffer cache and disk may be incorrect because it is not known how many disk requests are made for each lookup operation,
3. the ethernet model may not be correct, and
4. the distribution of requests from the client to the server is not random.

Base Model

The model in Figure 9.2 was solved analytically using LQNS and by simulation to find the throughput at the client. The results are compared to the average of twenty live runs in Table 9.6 for both accuracy and run time. The run time given for the live run is the sum of the twenty runs.

Method	NFSstones		Solution
	λ	ϵ_λ (%)	Run Time
Live	2820	N/A	11:15.00
LQNS	2984	5.82	11.98
Sim.	2883	2.23	2:02:56.80

Table 9.6: Base model results. The simulations were run with a 95% confidence interval of $\pm$ 5.0% for each of the entry's service times.

Table 9.7 shows the utilization of all of the four principle components in Figure 9.2. The performance model clearly shows that the ethernet is the bottleneck in the system.

Component	Utilization	
	Live	LQNS
client	0.36	0.40
server	0.59	0.67
disk		0.53
network		1.00

Table 9.7: Base model component utilizations.

Sensitivity to Non-exponential Service Times

The base model assumes that all of the service times are exponentially distributed. To study the effect on the variance of service times, the coefficient of variation, $\overline{t}/\sigma_t$, was calculated from the measured means and standard deviations found in Tables 9.2, 9.3 and 9.5. No data is available for the ethernet, so it is assumed that its service time is exponentially distributed. The results in Table 9.8 show that the change in predicted throughput is minimal. However, the run time of the simulation increased by a factor of four.

Method	NFSstones		Solution
	λ	ϵ_λ (%)	Run Time
Live	2820	N/A	11:15.00
LQNS	2994	6.17	22.68
Sim.	2910	3.19	7:59:13.75

Table 9.8: Results for near-deterministic service times. Simulations were run with a 95% confidence interval of ± 5.0% for each of the entry's service times.

Sensitivity to Disk Classes

Figure 9.3 shows the disk service time is highly variable. The base model assigns service time into two buckets to accommodate this variability. Two other cases are considered here:

1. a case with only one disk class, shown in Figure 9.4(a). The service time is simply

the mean service time for the read or write request.

2. a case with four disk classes, shown in Figure 9.4(b). The first bucket consists of all requests less than or equal to 4K bytes in size. The remaining three buckets are based on the mean and standard deviation of all of the requests larger than 4K bytes in size (i.e., the "big" read/write bucket in the base model). The buckets used are shown in Table 9.9.

The changes to the base model for both cases are shown in Figure 9.4.

Parameter	Service Time (mS)		Size (1K Blocks)	
	$\overline{t}$	σ_t	$\overline{\text{size}}$	σ_{size}
Read	7.17	2.34	39.47	18.63
Write	10.16	4.29	117.48	72.67

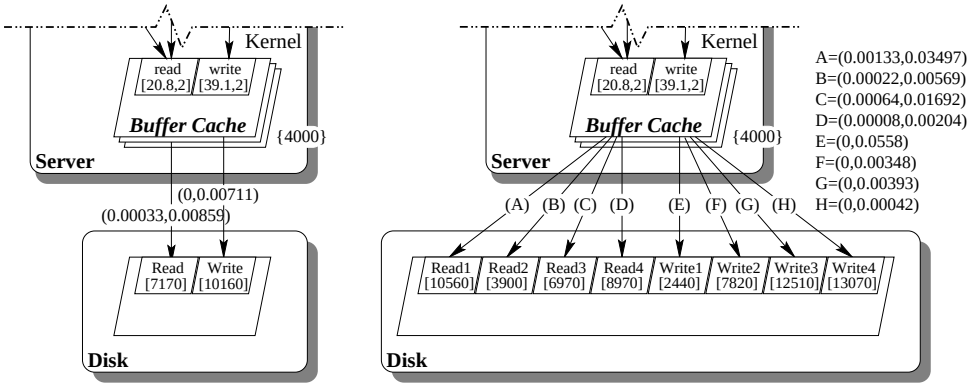
(a) Single Read/Write Class

Parameter			Service Time (mS)		Size (1K Blocks)		Requests (%)
	(1K Blocks)						
	Start	End	$\overline{t}$	σ_t	$\overline{\text{size}}$	σ_{size}	
Read1	0	4.0	10.56	1.99	2.05	0.15	7.1
Read2	4.0	26.26	3.99	2.16	18.79	6.46	11.1
Read3	26.26	58.40	6.97	8.34	40.33	8.34	67.6
Read4	58.40	∞	8.97	1.41	70.18	9.81	14.2
Write1	0	4.0	2.49	1.83	2.44	0.37	13.9
Write2	4.0	84.66	7.82	2.59	40.28	27.30	14.5
Write3	84.66	196.66	12.51	1.19	152.81	28.15	62.1
Write4	196.66	∞	13.07	0.86	214.82	15.83	9.5

(b) Four Read/Write Classes

Table 9.9: Measured Disk Parameters for one and four class disk models.

Table 9.10 shows the results from using 1, 2 and 4 classes to represent disk service times. Despite the odd distribution of time, a model with only one class is only marginally less



(a) Single Read/Write Class (b) Four Read/Write Classes

Figure 9.4: Vary the number of disk classes.

accurate than the more complex models. The more complex four-class model also takes significantly more time to simulate.

Method	c	nfsstones		Solution
		λ	ϵ_λ (%)	Run Time
Live	na	2820	N/A	11:15.00
LQNS	1	3002	6.45	11.98
Sim.		2927	3.79	2:02:56.80
LQNS	2	2985	5.85	10.28
Sim.		2883	2.23	2:00:27.01
LQNS	4	2984	5.82	9.92
Sim.		2880	2.13	4:30:50.42

Table 9.10: Results for c classes of disk service. The simulations were run with a 95% confidence interval of $\pm$ 5.0% for each of the entry's service times.

Sensitivity to Ethernet Service Time

Figure 9.7 shows that the bottleneck for NFS performance in the test system is the ethernet. The ethernet's service time was estimated using (9.6), and adjusted to account for a maximum efficiency of 83%. Figure 9.5 shows the effect on throughput when the ethernet service time is varied by ± 20%. This figure shows a difference of 45% between the highest and lowest throughput values; the model (and the live system) are very sensitive to ethernet performance.

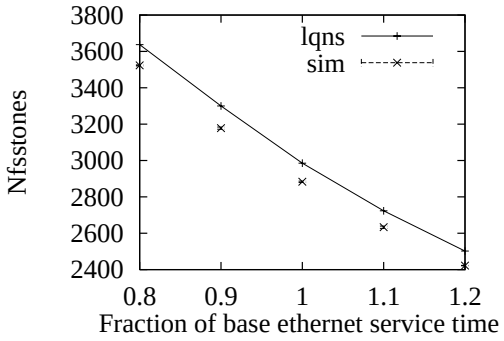


Figure 9.5: Effect on ethernet service time on Throughput.

Sensitivity to Client Cache Hit Ratio

The amount of traffic seen by the bottleneck in the base model is dependent on the client's cache miss ratio. Figure 9.6 shows the effect on throughput when the miss ratio is varied from 0.04 to 0.16 (a hit ratio of 0.96 to 0.84). This figure shows a difference in throughput of 21% as a result of a 14% difference in the hit ratio.

Server Cache Hit Ratio Sensitivity

Figure 9.7 shows the effect on throughput when the phase 1 visit ratio from the server's buffer cache to the disk is varied by ± 20%. The figure shows that blocking reads at the

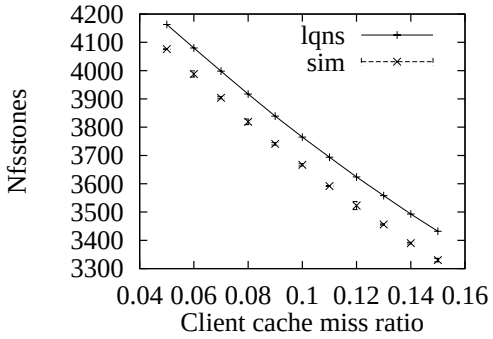


Figure 9.6: Effect of Client Cache miss ratio on Throughput.

server do not have a large effect on the client's throughput.

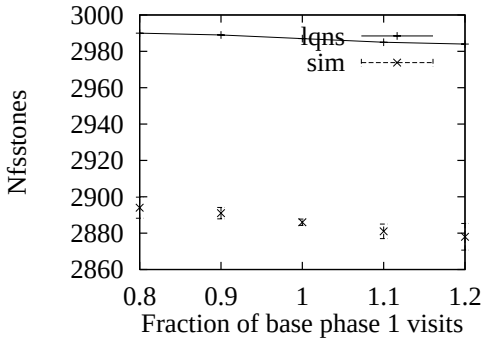
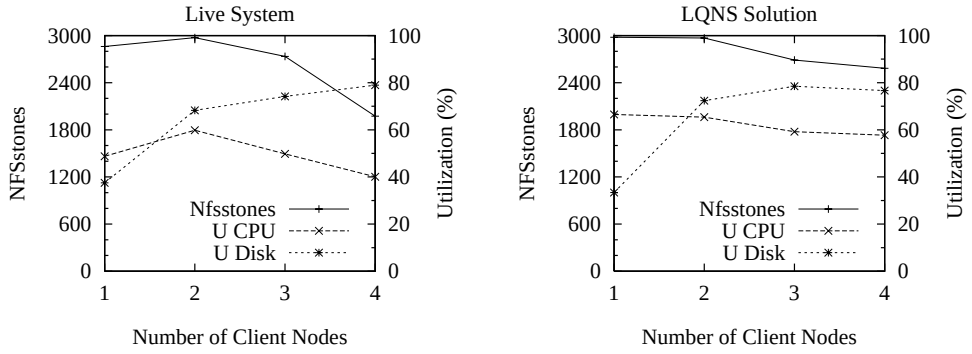


Figure 9.7: Effect of Server Cache miss ratio on Throughput.

Sensitivity to Multiple Clients

Figure 9.8 shows the effect of adding additional client nodes on the throughput of the NFS Server; Figure 9.8(a) is the results of running up four client nodes, each with 60 clients, on one server and Figure 9.8(b) shows the corresponding results from the analytic model.

The results in Figure 9.8 compare favourably for one to three clients. However, the live system shows a noticeable drop in throughput and CPU utilization when there are four



(a) Live System (b) Analytic Model

Figure 9.8: Varying the number of clients.

clients. Figure 9.9 shows a possible explanation. As the number of client nodes is increased, the number of collisions also increases.

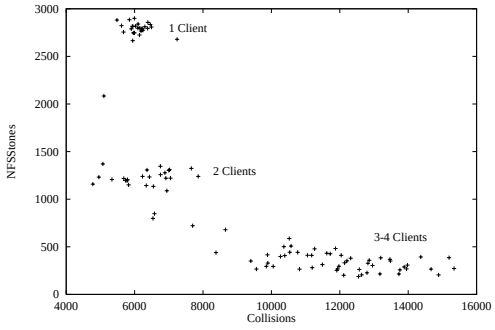


Figure 9.9: Ethernet collisions.

9.3.2 Performance Predictions

The base model described earlier in Section 9.3.1 was varied in a number of ways to study the effect of possible changes in the system. The changes were analyzed using the workload

described here (labelled "Base" in the tables follow), the Laddis workload ("Laddis"), and with the Laddis workload using a network which is ten times faster than the 100Mb one analyzed here ("Fast").

Synchronous Writes

The NFS protocol specification states that writes are to be committed to stable storage before the server replies to the client. The Linux NFS implementation simply waits for the write request to complete then replies; the disk block is actually written a short time afterwards. *Write gathering* [67] has a similar effect, though the latter follows the NFS semantics whereas the Linux NFS implementation does not.

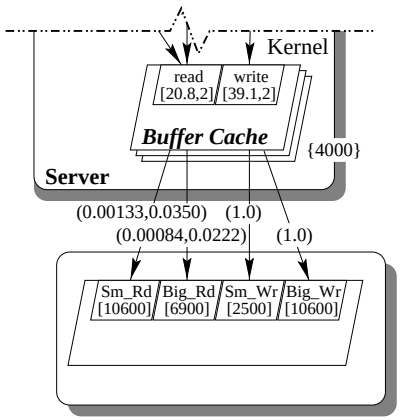


Figure 9.10: Synchronous Writes.

To properly implement NFS, *rpc.nfsd* was modified to perform synchronous writes on the live system. The LQN model was modified as follows and shown in Figure 9.10.

1. All writes from the server's *buffer cache* were moved to phase 1 from phase 2.

2. Each write is assumed to take the same time as a 4K Read. The extra time allows for seeking and rotational delays.

3. Each 8K write results in two write requests: one for the meta data in the inode, and one for the actual data. Both writes must complete synchronously.

The results, shown Table 9.11, show a significant degradation in performance. Even though writes make a small fraction of the workload, their behaviour has a significant effect on the performance of NFS.

Client Model	Method	NFSstones (λ)		Change (%)
		Base	Synch.	
Base	Live	2820	1072	−61.99
	LQNS	2984	538	−81.97
	Sim.	2883	848	−70.59
Laddis	LQNS	1947	254	−86.95
	Sim.	1906	406	−78.67
Fast	LQNS	2566	254	−90.10
	Sim.	2653	406	−84.66

Table 9.11: Comparison of Synchronous Writes to Base model.

Synchronous Write Gathering

Write gathering [67] is a technique used to improve NFS performance while retaining the synchronous write semantics. Writes to the disk, and replies to the client, are delayed by *rpc.nfsd* in the hope that additional write requests will arrive a short time later. The goal is to merge the requests into one large operation thus saving significant amounts of disk activity from meta data updates intermixed with data updates.

To model synchronous writes, the base model was modified by moving the write requests at the server's *buffer cache* from phase 2 to phase 1. The write performance from the base model is retained.

The results, shown in Table 9.12, show that write gathering has significant performance benefits. However, when the network performance is improved to the point where it is no longer the bottleneck, synchronous writes still impact performance.

Client Model	Method	NFSstones (λ)		Change (%)
		Base	Gather	
Base	LQNS	2990	2977	−0.44
	Sim.	2884	2787	−3.35
Laddis	LQNS	1947	1940	−0.36
	Sim.	1906	1895	−0.57
Fast	LQNS	2566	2214	−13.72
	Sim.	2653	2213	−16.55

Table 9.12: Comparison of Synchronous Writes with Gathering to Base model.

Big Reads

The Linux 2.0 NFS implementation performs reads in chunks no larger than 4K bytes, even if the `rsize` mount parameter is larger than this value. This behaviour is an artifact of the file system implementation because the virtual file system makes read requests on a 4K page basis. Conversely, the largest write request is limited only by the `wsize` mount parameter. Larger requests can improve performance by reducing network traffic.

To study the performance impact of RPC read requests which are twice as large, the model was modified as follows (shown in Figure 9.11):

1. The call rates from the client's buffer cache read to *nfsiod* and *rpc.nfsd* were halved to reflect the larger reads. The number of requests to the ethernet was changed to reflect the larger read size.

2. The call rates from the client buffer cache's read to *ether* was changed from four packets per request to seven packets per request.

3. The request rates from the server's buffer cache read to the *disk* were doubled because each read request is now twice as large.

Table 9.13 shows the estimated performance improvement from this change. The base model shows the biggest performance improvement because the ethernet is the bottleneck

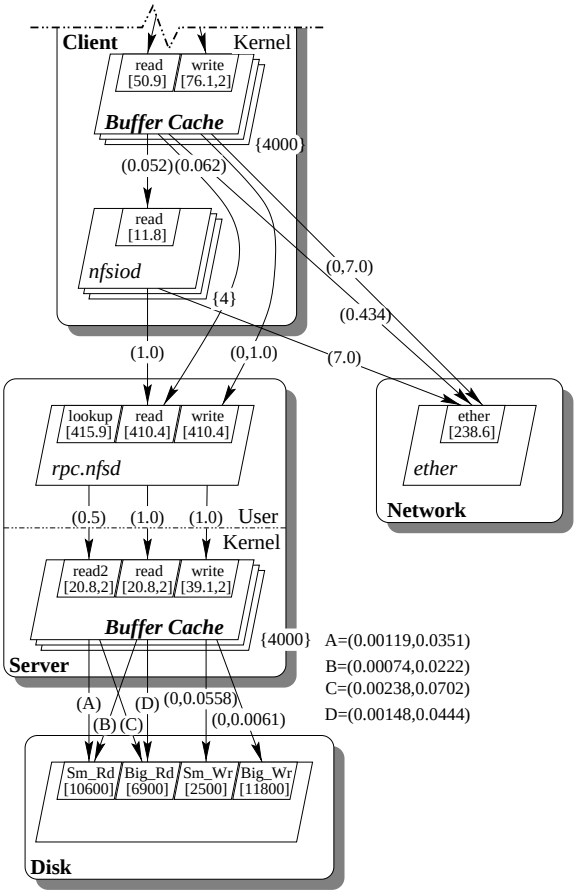


Figure 9.11: 8K Read.

and the workload is skewed towards reads. When the Laddis workload mix is used with the 100Mb ethernet, there is very little difference between the base and 8k-read throughput results because this workload has less read activity. However, with a faster ethernet, the larger read request size shows more improvements.

Client Model	Method	NFSstones (λ)		Δ (%)
		Base	8K Read	
Base	LQNS	2990	3173	6.10
	Sim.	2884	3054	5.89
Laddis	LQNS	1947	1973	1.34
	Sim.	1906	1930	1.25
Fast	LQNS	2565	2654	3.43
	Sim.	2653	2733	3.01

Table 9.13: Comparison of 8K Read Requests to Base model.

Implementing rpc.nfsd as a Kernel Process

The current NFS server is a user mode process that uses the existing kernel interface to interact with the server's file system. Performance improvements are possible by making the *rpc.nfsd* process a kernel-mode-only process because less context switching will occur, and because the process can access the file system data structures in a more efficient manner.

To study this change, the service time at the *rpc.nfsd* process was decreased by 30%. This change to the base model is shown in Figure 9.12.

Table 9.14 shows the results of the change. Since the ethernet is the bottleneck for the Base and Laddis configurations, no significant improvement in throughput is seen. However, when the network performance is improved (the fast case), moving the performance bottleneck to the *rpc.nfsd* process, NFS performance improves significantly.

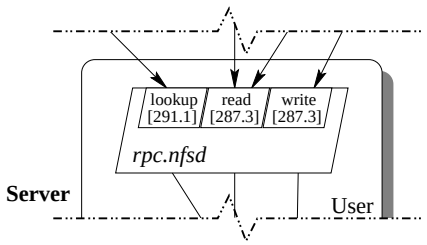


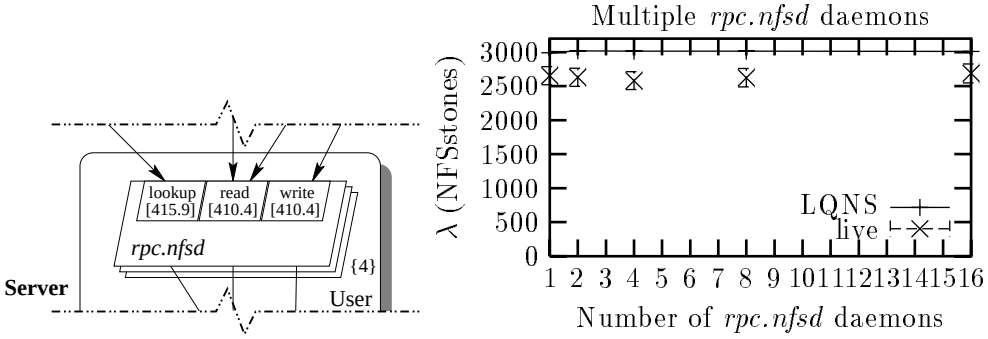
Figure 9.12: Kernel-based *rpc.nfsd*.

Client Model	Method	NFSstones (λ)		Δ (%)
		Base	knfsd	
Base	LQNS	2990	3008	0.61
	Sim.	2884	2887	0.14
Laddis	LQNS	1947	1950	0.15
	Sim.	1906	1907	0.06
Fast	LQNS	2566	3400	32.50
	Sim.	2653	3475	31.00

Table 9.14: Comparison of Kernel-Based nfsd to Base model.

Multiple rpc.nfsd Threads

For this configuration, *rpc.nfsd* was updated to version 2.2beta37 to allow multiple processes to handle NFS requests. Figure 9.13(a) shows the change to the base LQN model, and Figure 9.13(b) shows the results for 1 to 16 server processes.



(a) Model (b) Results

Figure 9.13: Multiple *rpc.nfsd* threads.

From Figure 9.13 multiple server threads do not improve performance. *Rpc.nfsd* only blocks on a physical read to the disk, which happens infrequently. Table 9.15 shows the results for the LADDIS workload and for the Fast network. Multiple threads appear to be beneficial only in the Fast network configuration.

9.4 Conclusions

This chapter has presented a layered queueing network model of the Linux Version 2. Network File System. The throughput error of the analytic model was about 6% when compared to the system under test. When the results of the analytic model are compared to the simlations, the results are typically within 3% of each other. The solution run time of the analytic model was roughly 50 times faster than the run time of the nfsstone benchmark

Client Model	Method	NFSstones (λ)		Δ (%)
		Base	4 Servers	
Base	Live	2656	2581	−2.84
	LQNS	2990	3016	0.88
	Sim.	2884	2927	1.52
Laddis	LQNS	1947	1904	0.15
	Sim.	1906	1908	0.08
Fast	LQNS	2566	2821	9.94
	Sim.	2653	2783	4.93

Table 9.15: Comparison of Multiserver *rpc.nfsd* to Base model. The standard deviation in throughput for the live run is about 130 for the base and 4 servers cases.

and about 600 times faster than the simulation.

The performance model described here shows that that the network is the bottleneck for performance. Performance can be improved by reducing the number of packets used for read requests by increasing the request size from 4096 to 8192 bytes. The model suggests that other proposed improvements, such as a kernel-based implementation of the NFS server, rpc.nfsd, will not yield significant improvements unless network performance is improved.

The Linux V2. implementation of NFS violates the semantics of NFS because writes are not committed to stable storage before the server replies to the client. However, using synchronous writes at the server severely degrades performance, even though writes are not significant portion of the workload. *Write-gathering*, where replies are deferred and batched, can ameliorate much of the overhead of synchronous writes while retaining full NFS semantics.

Finally, since the network is the bottleneck for NFS performance, more accurate analytic models are needed in the layered queueing network approximation. In particular, the current approximation techniques fail to accurately account for ethernet contention.

Chapter 10

Conclusions

The contribution of this thesis is to extend a modeling tool (layered queueing) in several important ways which are needed to meet the challenge of distributed systems. To do this the author has created new approximations, and integrated together known approximations that had not previously been used in combination. He has evaluated competing approaches to some of these approximations, and he has created a tool which allows a wide range of combinations to be selected by the user. Finally he has evaluated many of these combinations on many examples. The result is a new solver, called LQNS (Layered Queueing Network Solver).

The contributions will be summarized under the headings of accuracy improvements, modeling extensions, and case studies.

10.1 Accuracy Improvements

The accuracy of analytic solutions to layered queueing networks has been improved by changes to the interlocking and overtaking calculations.

10.1.1 Interlocking Calculation

The solution technique used to solve layered queueing networks in this work uses hierarchical decomposition to break the input model into a set of submodels. Parameters are exchanged between adjacent submodels during their solution. However, there are often traffic dependencies (referred to as "interlocking") between non-adjacent submodels which, if unaccounted for, can introduce errors of up to 50% or more in the solution. Accounting for interlock reduces the solution error to about 1 to 2% for the cases described here.

The algorithms incorporated into LQNS improve upon previous approaches by handling a broader range of situations where interlocking arises by generalizing the earlier algorithms and by searching more exhaustively through the model for interlocking relationships.

10.1.2 Overtaking Calculation

Previous analytic solutions for servers with two phases of service were overly simplified resulting in solutions with errors in throughput of over 60% for some cases. Two improvements have been described in this work which have significantly improved accuracy. First, a more detailed analysis of the overtaking probabilities takes place which improves the accuracy for cases where a two-phase client calls a two-phase server. Second, an improved expression for finding the waiting time at a two-phase server is included. This enhancement is particularly noticeable for models with multiple customers in the routing chains of the underlying MVA submodel. Both improvements together have reduced the solution error by over to no more than 10% in some of the worst cases described here.

10.2 Modelling Power Enhancements

The thesis introduces four enhancements to broaden the modelling power of the layered queueing networks, described in the sections below.

10.2.1 Forwarding

The first extension to the model is forwarding. Forwarding defers the reply to a client from an intermediate level server and forwards it to a lower level server. In distributed systems, forwarding improves performance by reducing network traffic by eliminating one or more replies. Forwarding can also be used to close open systems and to model systems built using asynchronous messaging with synchronous interactions.

10.2.2 Two-Phase Multiservers

The technique of early replies is used in distributed systems to improve performance by reducing the time a client spends blocked at a server (it is most effective when the server is not heavily utilized). This research extends two-phase service to multiservers. Two phase multiservers can be used to model distributed systems with multiservers that make early replies and they can be used to model resource pools such as file system buffers as was shown in Chapter 9.

10.2.3 Activities

Heterogeneous threads and other non-sequential patterns of execution were previously largely outside the capabilities of the model. To model them, new model constructs were introduced using *activities*. Activities are the lowest level of detail necessary in the performance model and can be connected together in a variety of patters. A simulator has been created which can solve model with any form of activity connection. Unfortunately, analytic techniques do not exist to handle the entire interconnection space.

10.2.4 Intra-task Fork-Join

One important subset of activity interconnection is fork and join within a task (intra-task fork-join). This type of interaction is found in transaction monitors for distributed databases

and in disk arrays (e.g. RAID). This research extends the work of Mak and Lundstrom to incorporate heterogeneous threads into single and multiservers. While this approach is not as accurate as approximations based on decompositions with flow-equivalent servers, it is sufficiently accurate for practical purposes. Further, it is computationally efficient and scales well to very large systems.

10.2.5 Summary

Extensions to the model can be used with simulation even if the analytic approximations are inadequate. The contribution of the thesis is the model extension itself. Features for which simulation is still the preferred method include priorities and inter-task fork-join.

10.3 Case Study

An analytic model of the Linux V2.0 NFS implementation was constructed and solved using LQNS. The performance model showed that performance was limited by the 100Mb ethernet connnection between the client and the server. Linux acheives its high performance by performing writes at both the client and the server asynchronously; this was modelled using a two-phase multiserver. The model was then used to explore changes in the system such as synchronous writes and large reads. In some cases, the test system was also changed to verify the predictions.

Compared to the test system, throughput error from the analytic solution were about 6%, and roughly twice that of a simulation of the same model. Run times for the model were 50 times faster than run times of the test system, and more than 600 times faster than the run times of the simulation.

10.4 Future Research

The activity notation introduced in LQNS can be used to solve analytically models with a broad variety of interactions. At present, only intra-task fork-join works well. Research is needed to capture accurately the correlation of arrivals at joins for systems with inter-task fork-joins.

The analytic solution of models with intra-task fork-joins is restricted to models that can be hierarchically decomposed. The algorithms that aggregate the threads that arise in these models needs to be extended to incorporate non-hierarchical (or spaghetti like) interactions.

The current solver uses a fixed point iteration scheme to solve the interconnected sub-models. This technique works well in the vast majority of cases solved to date, but sometimes either converges slowly or fails altogether when two or more servers saturate at approximately the same rate. Research is needed to identify the causes of convergence problems and to correct it when it occurs.

Bibliography

[1] ACM Sigmetrics. *Proceedings of the First International Workshop on Software and Performance (WOSP '98)*, Santa Fe, NM, oct 1998. Association for Computing Machinery.

[2] *The Programming Language Ada: Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1983.

[3] Yonathon Bard. Some extensions to multiclass queueing network analysis. In Matyas Arato, Alexandre Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*. North Holland, Amsterdam, 1979.

[4] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248 – 260, April 1975.

[5] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2nd edition, 1994.

[7] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service centers. *Performance Evaluation*, 4:241–260, 1984.

[8] Raymond M. Bryant, Anthony E. Krzesinski, M. Seetha Lakshmi, and K. Mani Chandy. The MVA priority approximation. *ACM Transactions on Computer Systems*, 2(4):335–359, November 1984.

[9] R. J. A. Buhr and R. S. Casselman. *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, Upper Saddle River, NJ, 1996.

[10] Jeffery P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, September 1973.

[11] K. Mani Chandy, John H. Howard, Jr, and Don Towsley. Product form and local balance in queueing networks. *Journal of the ACM*, 24(2):250–263, April 1977.

[12] K. Mani Chandy and Doug Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Communications of the ACM*, 25(2):126–134, February 1982.

[13] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high performance reliable secondary storage. *ACM Computing Surveys*, 36(3):145–185, August 1994.

[14] D. R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[15] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[16] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, February 1979.

[17] G. Chiola. A graphical Petri net tool for performance analysis. In Serge Fdida and Guy Pujolle, editors, *Modelling Techniques and Performance Evaluation*. Elsevier Science, Amsterdam, March 1987.

[18] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN-1.7 – graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1–2):47–68, November 1995. special issue on Performance Modeling Tools, S.S. Lavenberg and E.A. MacNair guest editors.

[19] Wesley W. Chu and Kin K. Leung. Task response time model and its applications for real-time distributed processing systems. In *Real-Time Systems Symposium*, Austin, TX, December 1984. IEEE Computer Society Press.

[20] Wesley W. Chu and Kin K. Leung. Module replication and assignment for real-time distributed systems. *Proceedings of the IEEE*, 75(5):547–562, May 1987.

[21] Wesley W. Chu, Chi-Man Sit, and Kin K. Leung. Task response time for real-time distributed systems with resource contention. *IEEE Transactions on Software Engineering*, 17(10):1076–1092, October 1991.

[22] Gianfranco Ciardo. Manual for the SPNP package version 1.0. Technical Report DUKE–TR–1988–27, Duke University, January 1, 1988.

[23] A. E. Conway and D. O'Brien. Validataion of an approximation technique for queueing network models with chain-dependent FCFS queues. *Computer Systems Science & Engineering*, 6(2):117–121, April 1991.

[24] Adrian E. Conway. Fast approximate solution of queueing networks with multi-server chain-dependent FCFS queues. In Ramon Puigjaner and Dominique Potier, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 385–396. Plenum, New York, 1989.

[25] Adrian E. Conway, Edmundo de Souza e Silva, and Stephen S. Lavenberg. Mean value analysis by chain of product form queueing networks. *IEEE Transactions on Computers*, 38(3):432–442, March 1989.

[26] John R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, New York, 1991.

[27] E. de Souza e Silva and E. E. Lavenberg. Calculating joint queue-length distributions in product-form queueing networks. *Journal of the ACM*, 26(1):194–207, January 1989.

[28] E. de Souza e Silva and Richard R. Muntz. A note on the computational cost of the linearizer algorithm. *IEEE Transactions on Computers*, 39(6):840–842, June 1990.

[29] Edmundo de Souza e Silva and Richard R. Muntz. Approximate solutions for a class of non-product form queueing network models. *Performance Evaluation*, 7:221–242, 1987.

[30] John Dilley, Rich Friedrich, Tai Jin, and Jerome Rolia. Measurement tools and modeling techniques for evaluating Web server performance. In Raymond Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino, editors, *Computer Performance Evaluation Modelling Techniques and Tools*, volume 1245 of *Lecture Notes in Computer Science*, pages 155–168. Springer-Verlag, St. Malo, France, June 1997.

[31] C. I. Dimmer. The Tandem Non-Stop system. In T. Anderson, editor, *Resilient computing systems*, pages 178–196. Collins, London, 1985.

[32] Bharat Doshi. Single server queues with vacations. In Hideaki Takagi, editor, *Stochastic Analysis of Computer and Communication Systems*, pages 217–265. North Holland, Amsterdam, 1990.

[33] Derek L. Eager and John N. Lipscomb. The AMVA priority approximation. *Performance Evaluation*, 8:173–193, 1988.

[34] George J. Febish and David E. Y. Sarna. Building three-tier client-server business solutions. White paper, Object Soft. Corp., Englewood, NJ, March 1995.

[35] Michael L. Fontenot. Software congestion, mobile servers, and the hyperbolic model. *IEEE Transactions on Software Engineering*, SE-15(8):947–962, August 1989.

[36] Open Software Foundation. *Introduction to OSF DCE*. Prentice Hall, first edition, 1992.

[37] B. Francis. Client/server: the model for the 1990s. *Datamation*, 36(4):34–36, 38, 40, February 1990.

[38] Greg Franks. Layered queueing network solver software design. file:/home/greg/srvn/linearizer/doc/lqns_1.html.

[39] Greg Franks. Traffic dependencies in client-server systems and their effect on performance prediction. In *IEEE International Computer Performance and Dependability Symposium*, pages 24–33, Erlangen, Germany, April 1995. IEEE Computer Society Press.

[40] Greg Franks, Alex Hubbard, Shikharesh Majumdar, Dorina Petriu, Jerome Rolia, and Murray Woodside. A toolset for performance engineering and software design of client-server systems. *Performance Evaluation*, 24(1–2):117–135, November 1995.

[41] Greg Franks and Murray Woodside. Multi-threaded software servers with asynchronous and deferred operations. Submitted for publication., July 1998.

[42] Greg Franks and Murray Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proceedings of the First International Workshop on Software and Performance (WOSP '98)* [1], pages 120–130.

[43] Greg Franks and Murray Woodside. A re-usable plug-in performance model of the Linux 2.0 Network File System. Submitted for publication., 1999.

[44] E. Gelenbe and Mitrani I. *Analysis and Synthesis of Computer Systems*. Computer Science and Applied Mathematics. Academic Press, Toronto, 1980.

[45] Geoffrey Gordon. *System Simulation*. Prentice Hall, Englewood Cliffs, N.J., 2 edition, 1978.

[46] James Hall, Roberto Sabatino, Simon Crosby, Ian Leslie, and Richard Black. Counting the cycles: a comparative study of NFS performance over high speed networks. In *Proceedings of the 22nd Conference on Local Computer Networks (LCN '97)*, pages 8–19, Minneapolis, MN, November 1997. IEEE Computer Society Press.

[47] James Hall, Roberto Sabatino, Simon Crosby, Ian Leslie, and Richard Black. A comparative study of high speed networks. In *INFOCOM '98*, volume 2, pages 774–782, San Francisco, CA, March 1998. IEEE Computer Society Press.

[48] Paul Harmon. *Objects In Action: Commercial Applications of Object-Oriented Technologies*. Addison-Wesley, January 1993.

[49] Øystein Haugen. MSC methodology. Technical Report L-1313-7, SISU II, Oslo, Norway, December 1994.

[50] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 94–105, Ashville, NC, USA, December 1993. Published in ACM Operating Systems Review Vol.27, No.5, Dec. 1993.

[51] Philip Heidelberger and Kishor S. Trivedi. Analytic queueing models for programs with internal concurrency. *IEEE Transactions on Computers*, 32(1):73–82, January 1983.

[52] John Holm, Antonio Lain, and Prithviraj Banerjee. Compilation of scientific programs into multithreaded and message driven computation. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 518–525, 1994.

[53] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 transputer. *IEEE Micro*, 7(5):10–26, October 1987.

[54] C. Hrischuk, J. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 399–409. IEEE Computer Society Press, 1995.

[55] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high performance web servers over high speed networks. In *INFOCOM '98*, San Francisco, CA, March 1998. IEEE Computer Society Press.

[56] Chien-Yuan Huang, Shi-Chung Chang, and Chern-Lin Chen. Performance evaluation of a cache-coherent multiprocessor by iterative mean-value analysis. *Performance Evaluation*, 23(1):31–52, July 1995.

[57] J.R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131 – 142, October 1963.

[58] Patricia A. Jacobson and Edward D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Communications of the ACM*, 25(2):142–151, February 1982.

[59] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons, 1991.

[60] Bao Chyuan Jenq, Walter H. Kohler, and Don Towsley. A queueing network model for a distributed database testbed system. *IEEE Transactions on Software Engineering*, 14(7):908–921, July 1988.

[61] Minwen Ji, Edward W. Felten, and Li Kai. Performance measurements for multithreaded programs. In *Proceedings of the ACM SIGMETRICS '98/PERFORMANCE '98 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 161–170. ACM SIGMETRICS and IFIP Working Group 7.3, June 1998. Also as *Performance Evaluation Review* **26**(1).

[62] Xianghong Jiang. Evaluation of approximation for response time of parallel task graph model. Master's thesis, Department of Systems and Computer Engineering, Carleton University, Canada, April 1996.

[63] H. Jonkers. Probabilistic performance modelling of parallel numerical applications. In G. R. Joubert, D. Trystram, F. J. Peters, and D. J. Evans, editors, *Parallel Computing; Trends and Applications*, volume 9 of *Advances in Parallel Computing*, pages 707–711. North Holland, Amsterdam, 1994.

[64] H. Jonkers, A. J. C. van Gemund, and G. L. Reijns. A probabilistic approach to parallel system performance modelling. In Edward A. Stohr, editor, *Proceedings of the Twenty-Eigth Annual Hawaii International Conference on System Sciences*, volume II (Software Technology), pages 412–421, Wailea, Hawaii, January 1995. IEEE Computer Society Press.

[65] Henk Jonkers. Queueing models of parallel applications: The Glamis methodology. In Günter Haring and Gabriele Kotsis, editors, *Computer Performance Evaluation*,

volume 794 of *Lecture Notes in Computer Science*, pages 123–138. Springer-Verlag, Berlin, May 1994.

[66] Henk Jonkers and Gerard L. Reijns. Predicting the performance of general task graphs with underlying queueing model. In *Proceedings of the 1st Annual Conference of the Advanced School for Computing and Imaging*, pages 293–302, May 1995.

[67] Chet Juszczak. Improving the write performance of an NFS server. In *Proceedings of the Winter 1994 USENIX Conference*, pages 247–259, San Francisco, CA, January 1994. USENIX Asscociation.

[68] Olaf Kirch. Linux NFS performance and security. http://www.lunetix.de/kongress/abstracts/okir, February 1996. Talk from Internationale Linux Kongreß, May 1996, Berlin, Germany.

[69] R. L. Klevans, , and W. J. Stewart. From queuing-networks to Markov-chains - the XMARCA interface. *Performance Evaluation*, 24(1–2):23 – 45, November 1995. special issue on Performance Modeling Tools, S.S. Lavenberg and E.A. MacNair guest editors.

[70] A. Krzesinski and J. Greyling. Improved linearizer methods for queueing networks with queue dependent service centers. In *Proceedings of Performance '84 and 1984 ACM SIGMETRICS on Measurement and Modeling of Computer Systems*, Cambridge, MA, August 1984. ACM SIGMETRICS.

[71] Toshiyasu Kurasugi and Issei Kino. Approximation methods for two-layer queueing models. *Performance Evaluation*, 36–37:55–70, August 1999. Performance '99.

[72] Stephen S. Lavenberg, editor. *Computer Performance Modeling Handbook*, volume 4 of *Notes and Reports in Computer Science and Applied Mathematics.* Academic Press, Toronto, ON, 1982.

[73] Stephen S. Lavenberg and Charles H. Sauer. Analytical results for queueing models. In Stephen S. Lavenberg, editor, *Computer Performance Modeling Handbook*, number 4 in Notes and Reports in Computer Science and Applied Mathematics, pages 56–172. Academic Press, 1983.

[74] R. Greg Lavender and Douglas C. Schmidt. Active object – An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 27. Addison-Wesley, 1996.

[75] Edward D. Lazowska, John Zhorjan, Scott G. Graham, and Kenneth C. Sevcik. *Quantitative System Performance; Computer System Analysis Using Queueing Network Models*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[76] Louis-Marie Le Ny and C. Murray Woodside. Performance modelling of queues with rendezvous service. Technical Report 941, Institut National de Recherche en Informatique et en Automatique (INRIA), Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex, France, December 1988.

[77] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison-Wesley, 1997.

[78] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 98–109, Santa Clara, CA, May 1993. ACM SIGMETRICS. Also as *Performance Evaluation Review* **21**(1).

[79] J.D.C. Little. A proof for the queueing formula: $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[80] Jay Littman. Applying threads. In *Proceedings of the Winter 1992 USENIX Technical Conference and Exhibition*, pages 209–221, San Francisco, CA, USA, January 1992. USENIX Asscociation.

[81] Y. C. Liu and H. G. Perros. Approximate analysis of a closed fork/join model. *European Journal of Operational Research*, 53(3):382–392, August 1991.

[82] Y. C. Liu and H. G. Perros. A decomposition procedure for the analysis of a closed fork/join queueing system. *IEEE Transactions on Computers*, 40(3):365–370, March 1991.

[83] B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, D. Walsh, and P. Weiss. Overview of the sun network file system. Technical report, Sun Microsystems, Inc., January 1985.

[84] Victor W. Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[85] Victor W. K. Mak. Queueing network models for parallel processing of task systems: An operational approach. Technical Report CSL-TR-86-306, Stanford University, September 1986.

[86] Daniel A. Menascé, Virgilio A. F. Almeida, and Larry W. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*, chapter 7: Performance of Client-Server Architectures, pages 205–233. Prentice Hall, Englewood Cliffs, NJ, 1994.

[87] J. Moran, R. Sandberg, D. Coleman, J. Kepecs, and B. Lyon. Breaking through the NFS performance barrier. In *Proceedings of the 1990 Spring European UNIX Users Group*, pages 199–206, Munich, Germany, April 1990.

[88] Sape Mullender, Guido von Rossum, Andrew Tanenbaum, Robbert von Renesse, and Hans von Staveren. Amoeba: a distributed operating system for the 1990's. *Computer*, 23(5), May 1990.

[89] John E. Neilson, C. Murray Woodside, Dorina C. Petriu, and Shikharesh Majumdar. Software bottlenecking in client-server systems and rendezvous networks. *IEEE Transactions on Software Engineering*, 21(9):776–782, September 1995.

[90] Randolph Nelson, Don Towsley, and Asser N. Tantawi. Performance analysis of parallel processing sytems. *IEEE Transactions on Software Engineering*, 14(4):532–539, April 1988.

[91] Randolph D. Nelson. The mathematics of product form queuing networks. *ACM Computing Surveys*, 25(3):339–369, September 1993.

[92] E. Neron and C. M. Woodside. A performance model for rendezvous based systems with processor-shared service. Technical report, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, September 1986.

[93] Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, 1998.

[94] Amy M. Pan. Solving stochastic rendezvous networks of large client-server systems with symmetric replication. Master's thesis, Department of Systems and Computer Engineering, Carleton University, September 1996. OCIEE-96-06.

[95] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. NFS version 3: Design and implementation. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference*, pages 137–151, Boston, MA, June 1994. USENIX Asscociation.

[96] Odysseas I. Pentakalos, Daniel A. Menascé, Milt Halem, and Yelena Yesha. An approximate performance model of a Unitree Mass Storage System. In *IEEE Symposium on Mass Storage Systems (MSS '95)*, pages 210–224, Monterey CA, September 1995. IEEE Computer Society Press.

[97] Dorina C. Petriu. *Approximate Solution for Stochastic Rendezvous Networks by Markov Chain Task-Directed Aggregation.* PhD thesis, Carleton University, Ottawa, Ontario, Canada, 1991.

[98] Dorina C. Petriu. Approximate mean value analysis of client–server systems with multi-class requests. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.*, pages 77–86, Nashville, TN, U.S.A., May 1994. A.C.M. SIGMETRICS.

[99] Dorina C. Petriu, Shikharesh Majumdar, Jing-Ping Lin, and Curtis Hrischuk. Analytic performance estimation of client-server systems with multi-threaded clients. In V. Madisetti, E. Gelenbe, and J. Walrand, editors, *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'94)*, pages 96–100, Durham, NC, January 1994. IEEE Computer Society Press.

[100] Dorina C. Petriu and C. Murray Woodside. Approximate MVA for software client/server models by markov chain task-directed aggregation. In *The Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1991. I.E.E.E.

[101] Dorina C. Petriu and C. Murray Woodside. A new mean value analysis of client–server software by task-directed aggregation of markov models. Technical Report SCE-93-29, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, October 1993.

[102] D. Potier, editor. *Modelling Techniques and Tools for Performance Analysis*. North Holland, Amsterdam, 1984.

[103] S. Ramesh and H. G. Perros. A multi-layer client-server queueing network model with synchronous and asynchronous messages. In *Proceedings of the First International Workshop on Software and Performance (WOSP '98)* [1], pages 107–119.

[104] M. Reiser. Mean value analysis of queuing networks, a new look at an old problem. In Matyas Arato, Alexandre Butrimenko, and E. Gelenbe, editors, *Performance of Computer Systems*, pages 63–77. North-Holland, Amsterdam, 1979.

[105] M. Reiser and H. Kobayashi. On the convolution algorithm for separable queuing networks. In P. P. S. Chen and Mark Franklin, editors, *International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 109–117, Cambridge, Massachusetts, March 1976. ACM-Sigmetrics. IFIP Working Group 7.3 on Computer System Modelling., Association for Computing Machinery.

[106] M. Reiser and S.S. Lavenburg. Mean value analysis of closed multichain queueing networks. Technical Report RC 7023, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, March 1979.

[107] M. Reiser and S.S. Lavenburg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, 27(2):313–322, April 1980.

[108] Martin Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Transactions on Communications*, COM-27(8):1199 – 1209, August 1979.

[109] Martin Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Performance Evaluation*, 1(1):7–18, 1981.

[110] J. A. Rolia and K. A. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995.

[111] Jerome A. Rolia and Kenneth C. Sevcik. Fast performance estimates for a class of generalized stochastic Petri nets. In *Computer Performance Evaluation '92: Modelling Techniques and Tools*, pages 21–33. Edinburgh University Press, August 1993.

[112] Jerome Alexander Rolia. Performance estimates for systems with software servers: The lazy boss method. In Ignacio Casas, editor, *VIII SCCC International Conference On Computer Science*, pages 25–43, Santiago, Chile, July 1988. Chilean Computer Science Society.

[113] Jerome Alexander Rolia. *Predicting the Performance of Software Systems*. PhD thesis, Univerisity of Toronto, Toronto, Ontario, Canada. M5S 1A1, January 1992.

[114] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M Guillemont, F. Hermann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the chorus distributed operating systems. Technical Report CS/TR-90-25, Chorus Systèmes, February 1991.

[115] Ekkart Rudolph, Jens Grabowski, and Peter Graubmann. Tutorial on message sequence charts (MSC'96). In *Tutorial of the FORTE/PSTV'96*, Kaiserslautern, Germany, October 1996.

[116] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. Tutorial on message sequence charts. ftp://ftp.win.tue.nl/pub/techreports/sjouke/msctutorial.ps.Z, January 1996.

[117] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[118] James Rumbaugh, Blaha Michael, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[119] A. Rüth. Entwicklung, Implementierung und Validierung neuer Approximationsverfahren für die Mittelwertanalyse (MWA) zur Leistungsberechnung von Rechnersystemen. Diplomarbeit am IMMD der Friedrich-Alexander-Universität Erlangen-Nürnberg, 1987.

[120] Russel Sandberg. The sun network file system: Design, implementation and experience. Technical report, Sun Microsystems Inc., Palo Alto, California, 1985.

[121] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the 1985 Summer USENIX Conference*, pages 119–130, Portland OR, June 1985. USENIX Asscociation.

[122] W. H. Sanders, W. D. Obal, M. A. Qureshi, and F. K. Widjanarko. The UltraSAN modeling environment. *Performance Evaluation*, 24(1–2):89–115, November 1995.

[123] Charles H. Sauer and Edward A. MacNair. The evolution of the research queueing package. In Potier [102], pages 5–24.

[124] Rainer Schmidt. An approximate MVA algorithm for exponential, class-dependent multiple servers. *Performance Evaluation*, 29:245–254, 1997.

[125] P. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proceedings of the International Conference on Stochastic Control and Optimization*, Amsterdam, 1979.

[126] Fahim Sheikh and Murray Woodside. Layered analytic performance modeling of a distributed database system. In *17th International Conference of Distributed Computing*

Systems (ICDCS '97), pages 482–490, Baltimore, ME, May 1997. IEEE Computer Society Press.

[127] Barry Shein, Mike Callahan, and Paul Woodbury. NFSSTONE: A network file server performance benchmark. In *USENIX Summer 1989 Technical Conference Proceedings*, pages 269–275, Baltimore, MD, June 1989. USENIX Asscociation.

[128] M. Sherman. Open distributed transaction processing with encina. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems (PDIS-93)*, San Diego, CA, January 1993.

[129] Christiane Shousha, Dorina Petriu, Anant Jalnapurkar, and Kennedy Ngo. Applying performance modelling to a telecommunication system. In *Proceedings of the First International Workshop on Software and Performance (WOSP-98)* [1], pages 1–6.

[130] David Simpson. Cut costs with client/server computing? Here's how! *Datamation*, 41(18):38, October 1 1995.

[131] Connie U. Smith. *Performance Engineering of Software Systems*. The SEI Series in Software Engineering. Addison-Wesley, 1990.

[132] Connie U. Smith and Lloyd G. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Transactions on Software Engineering*, 19(12):720–741, July 1993.

[133] William J. Stewart. MARCA: Markov chain analyzer - a software package for Markov chain modelling. In William J. Stewart, editor, *Numerical Solutions to Markov Chains*, pages 37–82. Marcel Dekker, New York, 1991.

[134] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2 edition, 1991.

[135] Sun Microsystems, Inc. RFC 1094: NFS: Network File System Protocol specification, March 1989. See also RFC1813.

[136] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–64, December 1990.

[137] The Standard Performance Evalulation Corporation (SPEC). SPEC SFS97 benchmarks. http://www.spec.org/osg/sfs97/, 1997.

[138] Alexander Thomasian and Paul Bay. Queueing network models for parallel processing of task systems. In H. J. Siegel and Leah Siegel, editors, *Proceedings of the 1983 International Conference on Parallel Processing*, pages 421–428, Columbus, OH, August 1983. IEEE Computer Society Press.

[139] Alexander Thomasian and Paul F. Bay. Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers*, 35(12):1045–1054, December 1986.

[140] Elizabeth Varki. Mean value technique for closed fork-join queues. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 103–112, Atlanta, GA, May 1999. ACM SIGMETRICS. Also as *Performance Evaluation Review* **27**(1).

[141] Dan Walsh, Bob Lyon, Gary Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the Sun network file system. In *Proceedings of the Winter 1985 USENIX Conference*, pages 117–124, Dallas, TX, January 1985. USENIX Asscociation.

[142] Jia Wang and Srinivasan Keshav. Efficient and accurate ethernet simulation. Technical Report TR99-1749, Computer Network Research Group, Cornell University, Ithaca, NY, May 1999.

[143] James E. White. A high-level framework for network-based resource sharing. In *National Computer Conference*, New York, NY, June 1976. AFIPS Press. See also: RFC 707.

[144] Dirk Wisse. Measuring and modeling a distributed filesystem. http://cardit.et.tudelft.nl/DNPAP/Research/Dirk/asci/asci.html, May 1995.

[145] Mark Wittle and Bruce E. Keith. LADDIS: The next generation in NFS file server benchmarking. In *USENIX Summer 1993 Technical Conference Proceedings*, pages 111–128, Cincinnati, OH, June 1993. USENIX Asscociation.

[146] C. M. Woodside, J. E. Neilson, J. W. Miernik, D. C. Petriu, and Constantin R. Performance of concurrent rendezvous systems with complex pipeline structures. In Ramon Puigjaner and Dominique Potier, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 307–322, New York, September 1988. Plenum Press. International Conference on Modeling Techniques and Tools for Computer Performance Evaluation (4th : 1988 : Palma, Spain).

[147] C. M. Woodside, E. Neron, E. D.-S. Ho, and B. Mondoux. An "active server" model for the performance of parallel programs written using rendezvous. *Journal of Systems and Software*, pages 844–848, 1986.

[148] C. M. Woodside and C. Schramm. Scalability and performance experiments using synthetic distributed server systems. *Distributed Systems Engineering*, 3:2–8, 1996.

[149] C. Murray Woodside. Throughput calculation for basic stochastic rendezvous networks. *Performance Evaluation*, 9:143–160, 1989.

[150] C. Murray Woodside, Xianghong Jiang, and Alex Hubbard. A fast approximation for mean fork-join delays in parallel programs. DRAFT PAPER, October 1997.

[151] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous multi-tasking distributed software. Technical Report SCE-89-8, Department of Systems and Computer Engineering, Carleton University, April 1991. Originally released March 6, 1989 as "The Rendezvous Network Model for Performance Synchronous Multi-tasking Distributed Software.".

[152] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(8):20–34, August 1995.

[153] ITU-T Recommendation X.901. Information technology - open distributed processing - reference model: Overview. Technical Report X.901, International Telecommunications Union, August 1998.

[154] Recommendation Z.120. *Message Sequence Chart (MSC)*. International Telecommunication Union, Geneva, mar 1993.

[155] John Zahorjan, Derek L. Eager, and Hisham M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Performance Evaluation*, 8:255–270, 1988.

Appendix A

Input File Grammar

This appendix gives the formal description of SRVN input file grammar in BNF form. For the nonterminals the notation $\langle nonterminal\_id \rangle$ is used, while the terminals are written without brackets as they appear in the input text.

The notation

$$\{\cdots\}_n^m, \text{ where } n \leq m$$

means that the part inside the curly brackets is repeated at least n times and at most m times. If $n = 0$, then the part may be missing in the input text.

A.1 General Information

$\langle SRVN\_input\_file \rangle \rightarrow \langle general\_info \rangle \ \langle processor\_info \rangle \ \langle task\_info \rangle \ \langle entry\_info \rangle$
$\{\langle activity\_info \rangle\}_0^*$

$\langle general\_info \rangle \rightarrow$ `G` $\langle comment \rangle \ \langle conv\_val \rangle \ \langle it\_limit \rangle \ \langle print\_int \rangle_{\text{opt}}$
$\langle underrelax\_coeff \rangle_{\text{opt}} \ \langle end\_list \rangle$

$\langle comment \rangle \rightarrow \langle string \rangle$ /*comment on the model*/

$\langle conv\_val \rangle \rightarrow \langle real \rangle$ /*convergence value*/

⟨*it_limit*⟩ → ⟨*integer*⟩ /*maximum number of iterations*/

⟨*print_int*⟩ → ⟨*integer*⟩ /*intermediate result print interval*/

⟨*underrelax_coeff*⟩ → ⟨*real*⟩ /*under_relaxation coefficient*/

⟨*end_list*⟩ → `-1` /*end_of_list mark*/

⟨*string*⟩ → `"` ⟨*text*⟩ `"`

A.2 Processor Information

⟨*processor_info*⟩ → `P` ⟨*np*⟩ ⟨*p_decl_list*⟩

⟨*np*⟩ → ⟨*integer*⟩ /*total number of processors*/

⟨*p_decl_list*⟩ → $\{\langle p\_decl\rangle\}_0^{np}$ ⟨*end_list*⟩

⟨*p_decl*⟩ → `p` ⟨*proc_id*⟩ ⟨*scheduling_flag*⟩ $\langle quantum\rangle_{\text{opt}}$
$\langle multi\_server\_flag\rangle_{\text{opt}}$ $\langle replication\_flag\rangle_{\text{opt}}$ $\langle proc\_rate\rangle_{\text{opt}}$

⟨*proc_id*⟩ → ⟨*integer*⟩ | ⟨*identifier*⟩
/*processor identifier*/

⟨*scheduling_flag*⟩ → `f` /*First come, first served*/
| `p` /*Priority, premeptive*/
| `h` /*Head Of Line*/
| `r` /*Random*/
| `s` /*Processor sharing*/

⟨*multi_server_flag*⟩ → `m` ⟨*copies*⟩ /*number of duplicates*/
| `i` /*Infinite server*/

⟨*replication_flag*⟩ → `m` ⟨*copies*⟩ /*number of replicas*/

⟨*quantum*⟩ → ⟨*real*⟩ /*CPU quantum for RR sched.*/

⟨*proc_rate*⟩ → `R` ⟨*ratio*⟩ /*Relative proc. speed*/

⟨*copies*⟩ → ⟨*integer*⟩

⟨*ratio*⟩ → ⟨*real*⟩

A.3 Task Information

$\langle task\_info \rangle \rightarrow$ `T` $\langle nt \rangle$ $\langle t\_decl\_list \rangle$

$\langle nt \rangle \rightarrow \langle integer \rangle$ /*total number of tasks*/

$\langle t\_decl\_list \rangle \rightarrow \{\langle t\_decl \rangle\}_0^{nt}$ $\langle end\_list \rangle$

$\langle t\_decl \rangle \rightarrow$ `t` $\langle task\_id \rangle$ $\langle ref\_task\_flag \rangle$ $\langle entry\_list \rangle$ $\langle proc\_id \rangle$
$\langle task\_pri \rangle_{\text{opt}}$ $\langle multi\_server\_flag \rangle_{\text{opt}}$ $\langle replication\_flag \rangle_{\text{opt}}$

$\langle task\_id \rangle \rightarrow \langle integer \rangle \mid \langle identifier \rangle$
/*task identifier*/

$\langle ref\_task\_flag \rangle \rightarrow$ `r` /*reference task*/
| `n` /*non-reference task*/
| `f` /*non-reference task*/ /* FITO scheduling */
| `h` /*non-reference task*/ /* HOL scheduling */
| `p` /*non-reference task*/ /* PPR scheduling */

$\langle entry\_list \rangle \rightarrow \{\langle entry\_id \rangle\}_1^{ne_t}$ $\langle end\_list \rangle$
/*task t has ne_t entries*/

$\langle entry\_id \rangle \rightarrow \langle integer \rangle \mid \langle identifier \rangle$
/*entry identifier*/

$\langle task\_pri \rangle \rightarrow \langle integer \rangle$ /*task priority, optional*/

A.4 Entry Information

$\langle entry\_info \rangle \rightarrow$ `E` $\langle ne \rangle$ $\langle entry\_decl\_list \rangle$

$\langle ne \rangle \rightarrow \langle integer \rangle$ /*total number of entries*/

$\langle entry\_decl\_list \rangle \rightarrow \{\langle entry\_decl \rangle\}_1^{ne \times 9}$ $\langle end\_list \rangle$
/*k = maximum nb of phases*/

$\langle entry\_decl \rangle \rightarrow$ `a` $\langle entry\_id \rangle$ $\langle arrival\_rate \rangle$

$\langle entry\_decl \rangle \rightarrow$ `A` $\langle entry\_id \rangle$ $\langle activity\_id \rangle$ /* Initial activitiy */

		`c` ⟨*entry_id*⟩ $\{\langle coeff\_of\_variation\rangle\}_1^k$ ⟨*end_list*⟩
		`f` ⟨*entry_id*⟩ $\{\langle ph\_type\_flag\rangle\}_1^k$ ⟨*end_list*⟩
		`F` ⟨*from_entry*⟩ ⟨*to_entry*⟩ ⟨*p_forward*⟩ ⟨*end_list*⟩
		`i` ⟨*from_entry*⟩ ⟨*to_entry*⟩ $\{\langle fanin\_flag\rangle\}_1^k$ ⟨*end_list*⟩
		`o` ⟨*from_entry*⟩ ⟨*to_entry*⟩ $\{\langle fanou\_flag\rangle\}_1^k$ ⟨*end_list*⟩
		`s` ⟨*entry_id*⟩ $\{\langle ph\_serv\_time\rangle\}_1^k$ ⟨*end_list*⟩
		`y` ⟨*from_entry*⟩ ⟨*to_entry*⟩ $\{\langle ph\_RNV\_nb\rangle\}_1^k$ ⟨*end_list*⟩
		`z` ⟨*from_entry*⟩ ⟨*to_entry*⟩ $\{\langle ph\_SNR\_nb\rangle\}_1^k$ ⟨*end_list*⟩
		`Z` ⟨*from_entry*⟩ ⟨*to_entry*⟩ $\{\langle think\_time\rangle\}_1^k$ ⟨*end_list*⟩
⟨*arrival_rate*⟩	→	⟨*real*⟩ /* *open arrival rate to entry* */
⟨*ph_serv_time*⟩	→	⟨*real*⟩ /* *mean phase service time* */
⟨*ph_type_flag*⟩	→	⟨*integer*⟩ /* *0 – stochastic phase* */ /* *1 – deterministic phase* */
⟨*fanin_flag*⟩	→	⟨*integer*⟩ /* *Fanin for replication* */
⟨*fanout_flag*⟩	→	⟨*integer*⟩ /* *Fanout for replication* */
⟨*coeff_of_variation*⟩	→	⟨*real*⟩ /* *serv.time coeff. of variation* */
⟨*ph_RNV_nb*⟩	→	⟨*real*⟩ /* *mean number of RNVs/ph* */
⟨*ph_SNR_nb*⟩	→	⟨*real*⟩ /* *mean nb.of non-blck.sends/ph* */
⟨*p_forward*⟩	→	⟨*real*⟩ /* *probability of forwarding* */
⟨*think_time*⟩	→	⟨*real*⟩ /* *entry think time* */
⟨*from_entry*⟩	→	⟨*entry_id*⟩ /* *Source of a message* */
⟨*to_entry*⟩	→	⟨*entry_id*⟩ /* *Destination of a message* */

A.5 Activity Information

⟨*activity_info*⟩	→	`A` ⟨*task_id*⟩ ⟨*activity_defn_list*⟩ `:` ⟨*activity_conn_list*⟩ ⟨*end_list*⟩

/* Activity definition. */

⟨activity_defn_list⟩ → {⟨activity_defn⟩}$_1^{na}$

⟨activity_defn⟩ → `s` ⟨activity_id⟩ ⟨ph_serv_time⟩ /* Service time */
| `c` ⟨activity_id⟩ ⟨coeff_of_variation⟩ /* Sqr. Coef. of Var. */
| `f` ⟨activity_id⟩ ⟨ph_type_flag⟩ /* Phase type */
| `y` ⟨activity_id⟩ ⟨to_entry⟩ ⟨ph_RNV_nb⟩ /* Rendezvous */
| `z` ⟨activity_id⟩ ⟨to_entry⟩ ⟨ph_SNR_nb⟩ /* Send-no-reply */
| `Z` ⟨activity_id⟩ ⟨think_time⟩ /* Think time */

/* Activity Connections. */

⟨activity_conn_list⟩ → ⟨activity_conn⟩ {; ⟨activity_conn⟩}$_1^{na}$

⟨activity_conn⟩ → ⟨join_list⟩
| ⟨join_list⟩ `->` ⟨split_list⟩
| ⟨repeat_list⟩ `->` ⟨split_list⟩

⟨join_list⟩ → ⟨reply_activity⟩
| ⟨and_join_list⟩
| ⟨or_join_list⟩

⟨split_list⟩ → ⟨activity_id⟩
| ⟨and_split_list⟩
| ⟨or_split_list⟩

⟨and_join_list⟩ → ⟨reply_activity⟩ {`&` ⟨reply_activity⟩}$_1^{na}$

⟨or_join_list⟩ → ⟨reply_activity⟩ {`+` ⟨reply_activity⟩}$_1^{na}$

⟨and_split_list⟩ → ⟨activity_id⟩ {`&` ⟨activity_id⟩}$_1^{na}$

⟨or_split_list⟩ → ⟨prob_activity⟩ {`+` ⟨prob_activity⟩}$_1^{na}$

⟨repeat_list⟩ → ⟨real⟩ `*` ⟨activity_id⟩ ⟨next_activity⟩$_{\text{opt}}$

⟨prob_activity⟩ → `(` ⟨real⟩ `)` ⟨activity_id⟩

⟨reply_activity⟩ → ⟨activity_id⟩ ⟨reply_list⟩$_{\text{opt}}$

⟨next_activity⟩ → `,` ⟨activity_id⟩

$\langle reply\_list \rangle \quad \rightarrow \quad$ `[` $\langle entry\_id \rangle$ $\{$ `,` $\langle entry\_id \rangle$ $\}_0^{ne}$ `]`

Appendix B

Marginal Probabilities

The following equations give the marginal probabilities of finding i customers (of any class) at a station j for a customer population $\mathbf{N}$. They are found recursively from an initial condition of zero customers in the network.

$$\begin{aligned} P_m(i,\mathbf{N}) &= \frac{1}{i}\sum_{k=1}^{K} U_{mk}(\mathbf{N})P_m(i-1,\mathbf{N}-\mathbf{e}_k), \qquad 0 < i < J_m \\ PB_m(\mathbf{N}) &= \frac{1}{J_m}\sum_{k=1}^{K} U_{mk}(\mathbf{N})[PB_m(\mathbf{N}-\mathbf{e}_k) + P_m(J_m-1,\mathbf{N}-\mathbf{e}_k)] \\ P_m(0,\mathbf{N}) &= 1 - \sum_{i=1}^{J_m-1} P_m(i,\mathbf{N}) - PB_m(\mathbf{N}) \end{aligned} \tag{B.1}$$

The following equations give the marginal probabilities of finding $\mathbf{n}$ customers at a station j for a customer population $\mathbf{N}$. They are found recursively from an initial condition of zero customers in the network.

$$\begin{aligned} P_m(\mathbf{n},\mathbf{N}) &= \sum_{k=1}^{K} \frac{U_{mk}(\mathbf{N})}{\alpha_{mk}(\mathbf{n})} P_m(\mathbf{n}-\mathbf{e}_k,\mathbf{N}-\mathbf{e}_k) \qquad \forall \mathbf{n}, \mathbf{0} < \mathbf{n} \leq \mathbf{N} \\ P_m(\mathbf{0},\mathbf{N}) &= 1 - \sum_{0<\mathbf{n}\leq\mathbf{N}} P_m(\mathbf{n},\mathbf{N}) \end{aligned} \tag{B.2}$$

When using approximate MVA, the marginal probabilities at a customer population $\mathbf{N} - \mathbf{e}_k$ are found by assigning $P_m(i, \mathbf{N} - \mathbf{e}_k) = P_m(i, \mathbf{N})$. This assignment often leads to infeasible probabilities if (B.1) or (B.2) is used.

Krzesinski and Greyling [70] developed the following set of equations to find the marginal probabilities of finding i customers at a station when using Linearizer approximate MVA. $P_m(0, \mathbf{N})$ is set to 1, then used in (B.1) to find pseudo-probabilities $P_m(i, \mathbf{N})$ and $PB_m(\mathbf{N})$. The sum of these expressions is then used to re-normalize the probabilities, yielding:

$$\begin{aligned} P_m(0, \mathbf{N}) &= \left(1 + \sum_{i=1}^{J_m - 1} P_m(i, \mathbf{N}) + PB_m(\mathbf{N})\right)^{-1} \\ P_m(i, \mathbf{N}) &= \frac{U_m^i(\mathbf{N})}{\prod_{l=1}^{i} \alpha_m(l)}, \qquad 0 < i < J_m \\ PB_m(\mathbf{N}) &= \frac{U_m^{J_m}(\mathbf{N})}{\prod_{l=1}^{J_m} \alpha_m(l)} \cdot \frac{U_m(\mathbf{N})}{J_m - U_m(\mathbf{N})} \end{aligned} \tag{B.3}$$

Schmidt also found that (B.2) frequently needed renormalization [124]. He assumed that customers were distributed binomially within a queue at a multiclass FCFS multiserver:

$$P_m(\mathbf{n}, \mathbf{N}) = \prod_{k=1}^{K} \binom{N_k}{n_k} \left(\frac{L_{mk}(\mathbf{n})}{N_k}\right)^{n_k} \left(1 - \frac{L_{mk}(\mathbf{n})}{N_k}\right)^{N_k - n_k} \qquad \forall \mathbf{n}, \mathbf{0} \le \mathbf{n} \le \mathbf{N} \tag{B.4}$$

This equation is perfectly suited for approximate MVA because the marginal probabilities with one customer from chain k removed are not needed at all.