Resource Allocation Algorithms in AON Network

by

May Ku

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

February 1994

Signature of Author ....
Department of Electrical Engineering and Computer Science
January 14, 1994

Certified by ....
Dr. Steven G. Finn
Lecturer, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Prof. Frederic R. Morgenthaler
Chair, Committee on Graduate Students

Resource Allocation Algorithms in AON Network

by

May Ku

Submitted to the Department of Electrical Engineering and Computer Science
on January 14, 1994, in partial fulfillment of the
requirements for the degree of

Master of Science

in Electrical Engineering and Computer Science

Abstract

This thesis presents a study of scheduling algorithms for allocating system resources in the lowest level of a wideband All Optical Network (AON) proposed by a consortium of AT&T, DEC and MIT. Three scheduling algorithms are considered and applied to uniform traffic, multiclass traffic, and client/server traffic for both blocking and queueing systems. We present mathematical approximations and bounds for several queueing and blocking systems. Simulations using OPNET software were run for these scheduling algorithms and compared to the mathematical approximations and bounds. From our study we conclude that a Random Assignment Scheduling Algorithm seems to be a very promising scheduling approach for the lowest level of the proposed AON network.

Thesis Supervisor: Dr. Steven G. Finn
Title: Lecturer, Department of Electrical Engineering and Computer Science

Acknowledgments

This thesis is dedicated to my family.

I would especially like to thank my thesis advisor, Dr. Steven Finn, for guiding and encouraging me through this endeavor.

I would very much like to thank Nate Osgood for finding six SPARC 10 machines to run my simulations, for keeping crazy hours to ensure the runs would be kicked off promptly during the final rush to get the work complete. Your time and efforts are greatly appreciated.

The work was performed at the Laboratory for Information and Decision Systems (LIDS) at MIT. I would like to thank Prof. Robert Kennedy, Prof. Robert Gallager, Prof. Mitch Trott and Sheila Hegarty of LIDS for their various help.

Many thanks to my friend Hua Xu, whom I can always count on for emotional support. Thanks to my former colleagues at IBM, Darlene Shamsid-Deen, Tom Jung, Jim Kelly, Bob Lannin who were always helpful and encouraging.

This research was partially supported by the Advanced Research Project Administration Grant #MDA972-92-J-1038.

Contents

List of Figures

Chapter 1 Introduction

Recent advances in optical fiber technology makes it the preferred transmission medium for long-distance, point-to-point communications links. In U.S. alone, more than two million miles of fiber has been installed by long distance phone companies[1]. However, they are mostly operated at a capacity much lower than their terahertz potential. One current research topic is how to build an optical fiber communication network that will use the fiber bandwidth more effectively. Various ideas have been proposed. A popular approach is to employ *wavelength-division multiplexing* (WDM) which divides the optical spectrum into many different wavelengths, each corresponding to a different communications channel.

WDM networks can be further categorized. One could use a bus or star topology. The transmitter and receiver can be fixed or dynamically tuned to available wavelength channels. Various media access (MAC) protocols from fixed assignment to random access can be used. There are single-hop or multihop networks, where in a single-hop network, any two nodes can talk directly to each other via a wavelength channel. In multihop networks some node pairs may need to route through intermediary node(s) since they don't share the same wavelength channels. (For a review of various proposed WDM network, see [1,2,3].)

A consortium of AT&T, DEC, and MIT has proposed a WDM based wideband All-Optical Network (AON)[4]. It uses tree-of-stars topology at its lowest level and has a hierarchical structure. Each node has a tunable transmitter and receiver, thus all nodes can talk directly to each other (i.e. single-hop network). A demand assigned "scheduled TDM" MAC protocol is proposed for local communications, where a central agent(s) is responsible for allocating the time slots to requesting terminals.

This thesis will study scheduling algorithms for allocating the system resource in the lowest level of the AON network. We will conclude from our study that a *Random Assignment Scheduling Algorithm* developed in this thesis seems to be a very promising scheduling approach for the AON level 0 subnetwork. It appears to work across a wide variety of traffic requirements including uniform traffic, multiclass traffic, and client/server traffic.

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of the AON network's architecture, service, and network operations. Chapter 3 establishes the network model and defines the problem. Chapter 4 derives the mathematical approximations or bounds for systems studied. Chapter 5 describes the OPNET simulation package that we will be using, specifies the network and node models used for the simulation. It also discuss the three scheduling algorithms used. Chapter 6 discuss the simulation results and compare them to mathematical bounds. Chapter 7 discusses our conclusions. Appendix A includes the C programs used to calculate the mathematical formulas, and in Appendix B are the reports from OPNET.

An All-Optical Network (AON) has been architected, and a test bed based on this design will be built to investigate the utilization of terahertz bandwidth capacity of optical networks by a ARPA sponsored consortium of AT&T, DEC, and MIT[4].

In the AON network, optical signals flow across the network without being converted to electrical signals. The network is designed to be scalable in the dimensions of geographic span, the number of users, and data rate. It employs wavelength division multiplexing (WDM) and time division multiplexing (over each wavelength) techniques to access the fiber bandwidth. Frequency reuse is utilized to enable network expansion over multiple geographical areas.

2.1 Network Architecture

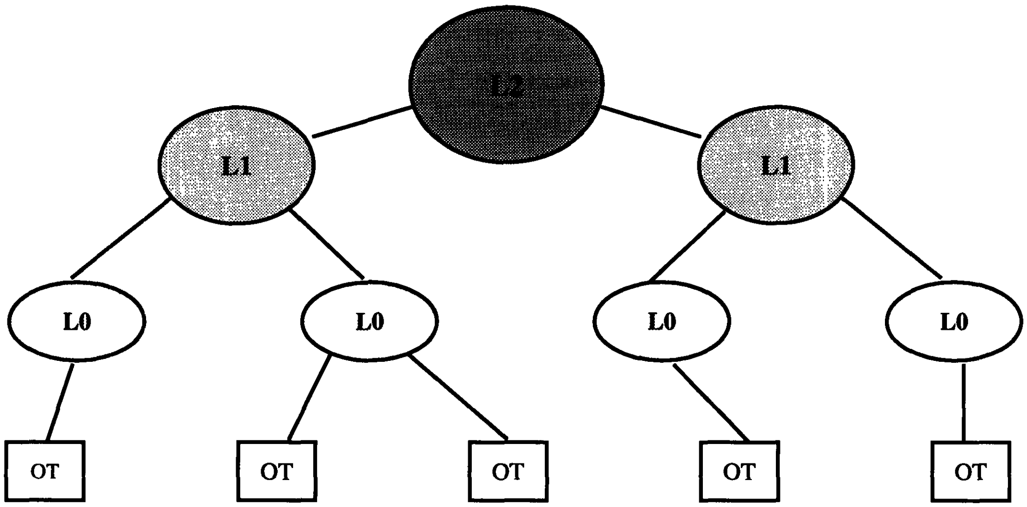


Figure 2.1.1 AON Network

The AON is a hierarchical network with three levels (L0, L1, L2) of sub-networks as shown in Figure 2.1.1. It is designed to scale gracefully to hundreds of thousands of all optical end nodes. One can consider L2 as the backbone of a national or worldwide network, L1 as a Metropolitan Area Network (MAN), and L0 as a Local Area Network (LAN). The lowest level L0 is a "local" broadcast star network. Optical Terminals (OTs) are attached to the AON via an L0 subnet. Within the L0 subnet, optical wavelengths are divided into three sets.

- L0 wavelengths: this wavelength set is used for local traffic between OTs within the same L0 subnet. L0 wavelengths are blocked from entering the L1 level by a frequency selective local bypass element located at the exit link of L0 to L1 subnet. These wavelengths may be reused in L2 and L1 subnets, as well as other L0 sub-networks.

• L1 wavelengths: this wavelength set is used for communication between OTs in different L0 subnets, which requires transmission through an L1 subnet.

• Control wavelength: this wavelength is dedicated for control, scheduling, network management, and datagram services.

The Media Access Control (MAC) protocol for the control wavelength channel is designed not to require a central resource or central timing since this channel is used for power-on configuration of the network. An Ethernet protocol based upon IEEE standard 10Broad36 will be used in the test-bed.

The L0 and L1 wavelength channels are allocated by a central scheduling agent located within the respective L0 and L1 subnet. Depending on the incoming request, a wavelength channel may be allocated as a whole, or as subunits by using "scheduled" time division multiplexing (TDM) techniques. The L0 subnet is a broadcast star network and doesn't support wavelength routing. We will discuss "scheduled TDM" in more detail in Chapter 3.

Multiple L0's may be connected to a L1, which is connected to a L2. There is a single L2 subnet in the network acting as the backbone. In each L0, L1 and L2 subnet, there is a dedicated control wavelength in addition to data wavelengths. In each subnet, there is a scheduling agent responsible for allocating the data wavelength channels as requested. Both L1 and L2 subnets support wavelength routing.

2.2 Network Services

Three basic services are provided by the AON network.

• Type-A "switched - physical circuit" services provide point-to-point or point-to-multipoint high speed circuit switched photonic sessions. It uses the entire bandwidth of a wavelength channel. The scheduling agent will allocate an entire wavelength channel to Type-A session.

• Type-B "scheduled TDM" services provide time division multiplexed (TDM) circuit-switched sessions in the range of a few Mbps to the full optical channel rate. It uses a portion of a wavelength channel. When we have Type-B session requests coming in, the scheduling agent will divide the wavelength channel(s) into slots using "scheduled TDM", and allocate slot(s) as requested.

• Type-C "unscheduled datagram" services use a dedicated "well-known" wavelength (i.e. control wavelength) for control, scheduling, network management, and datagram services. No scheduling is necessary for Type-C communication packets.

2.3 Network Operations

Optical Terminals send Type-A and Type-B session requests to the L0 scheduler via Type-

C packets. Upon receiving the request, the scheduler determines if adequate resource is available. If so, the scheduler allocates the resource and informs the destination(s) of the new session request. If the destination subsequently accepts the connection request, the requesting OT is reliably informed and the session begins. All sessions are unidirectional.

Scheduling of Type-A session is relatively straightforward. The scheduler needs to know that both the source and the destination have a free transmitter and receiver respectively, and a free wavelength channel is available. The scheduler informs the source and the destination of the wavelength channel to use, so they can tune their respective transmitter and receiver to the wavelength to start the session. For type-B sessions, since wavelength channels are time divided into slots, the scheduler needs to find enough slots to satisfy the session throughput request. It also has to make certain that both the source's transmitter and the destination's receiver are free during these slot intervals.

The frequency reuse property of the network gives the scheduler full control of its own resource. Therefore a session between OTs in the same L0 (intra-L0 session) can be established by the local L0 scheduler. Sessions between OTs in different L0's require L1 and possibly L2 resource, and cannot be scheduled by the local L0 scheduler alone. We will limit our study to the first case, and leave L0/L1/L2 scheduler cooperation to future investigation. From now on, network or system means a L0 subnet and available wavelength channels are the L0 wavelength channels for use of intra-L0, point-to-point Type-B sessions.

In this thesis, we will study the scheduling algorithm used for resource allocation for intra-L0, point-to-point Type-B sessions.

In this chapter we discuss the aspects of the AON network that are relevant to our study of intra-L0, point-to-point, Type-B session scheduling, and define the problem we will study.

3.1 System Resource Allocation

To access the system resource, wavelength division multiplexing (WDM) and time division multiplexing (over each wavelength) techniques are used. The available fiber bandwidth is divided into *W* wavelength channels of equal bandwidth. Transmission over each wavelength channel is organized in frames of equal size *T*. All frames over different wavelength channels are aligned.

A frame is said to have size *T* if it is divided into *T* slots of equal time duration. For a system with *W* wavelength channels and frame size *T*, the total number of available slots is *WT*. They are demand assigned to sessions by the scheduler. Each slot is referred to by its wavelength number and the position in the frame, (w, t), where $w \in (\lambda_1, \ldots, \lambda_w)$ and $t \in (1, \ldots, T)$. Slot A in Figure 3.1.1 is referred to as $(\lambda_{3,} 2)$, or simply as (3,2). A *row* of slots means all slots with the same wavelength number and different slot number.A *column* of slots means all slots with the same slot number and different wavelength channels. So *row* $i = \{(\lambda_i, j) : j = 1, \ldots, T\}$, and *column* $j = \{(\lambda_i, j) : i = 1, \ldots, W\}$.

When an Optical Terminal(OT) needs to establish a session, it sends the scheduler information on itself (source), the destination, and the throughput requirement *L* in terms of the number of slots needed per frame. The scheduler is responsible for allocating the required resource. Once the slot(s) is allocated, the session will use the same slot(s) in all subsequent frames until it terminates. Since we are primarily concerned here with the efficiency of the scheduling algorithm, we ignore the processing time it takes to establish a session.

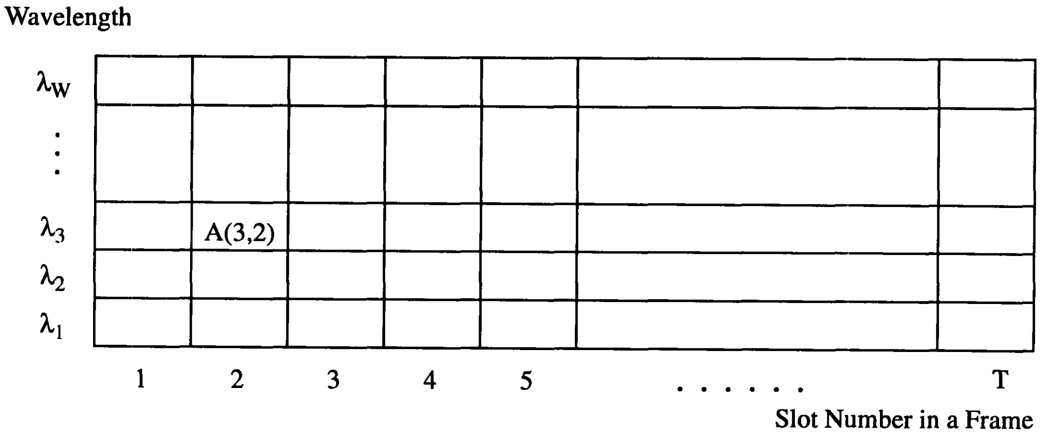


Figure 3.1.1 System Resources

3.2 Number of Transceivers

In the study, we assume each OT has only one tunable transmitter and one tunable receiver for Type-B sessions. Consequently, each OT can transmit or receive only on one wavelength at a time. Referring to Figure 3.1.1, a session which needs two slots can be assigned slot (3,2) and (1,4), but not (3,2) and (1,2). However an OT may be transmitting at (3,2) and receiving at (1,2). If an OT is transmitting over multiple slots, all slots must have different slot numbers. This constitutes the most basic constraint on our scheduling algorithm, and gives arise to the concept of "column conflict", "pre-column conflict", and "post-column conflict".

- *Column Conflict*

A session (*s,d*) is one with node *s* as source and node *d* as destination. When allocating resource for session (*s,d*), if either node *s* is transmitting or node *d* is receiving over column *j* involving some other session, we say there is a *column conflict* over *j* due to transmitter or receiver conflict, and none of the slots in column *j* can be assigned to session (*s,d*).

- *Pre-column Conflict*

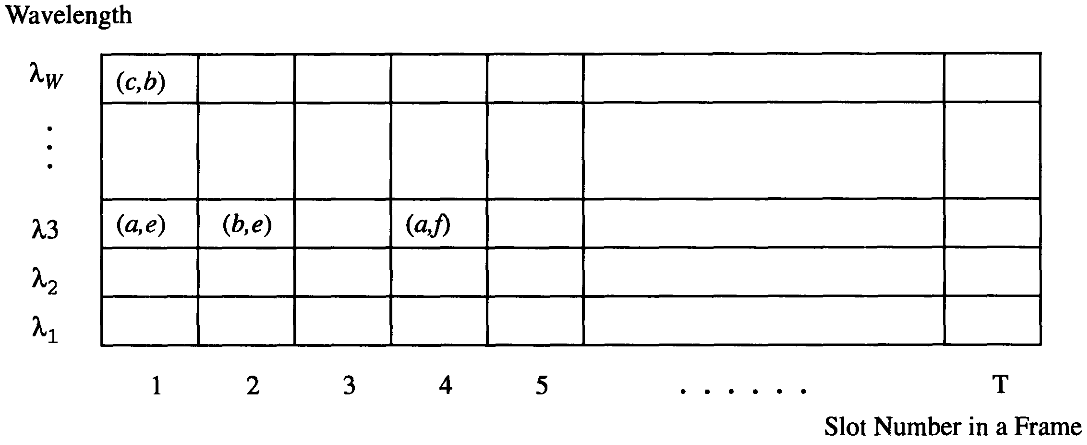
When allocating resource for session (*s,d*), if either node *s* is transmitting or node *d* is receiving over column *j -1* involving some other session(s) on wavelength λ_t or λ_r, we say there is a *pre-column conflict* over *j* due to transmitter or receiver conflict on wavelength λ_t or λ_r. It is possible that none of the slots in column *j* can be assigned to session (*s,d*) due to constraints on transmitter or receiver tuning times. Sometimes we will be able to resolve pre-column conflict as discussed in the next section.

- *Post-column Conflict*

When allocating resource for session (*s,d*), if either node *s* is transmitting or node *d* is receiving over column *j +1* involving some other session(s) on wavelength λ_t or λ_r, we say there is a *post-column conflict* over *j* due to transmitter or receiver conflict on wavelength λ_t or λ_r. It is possible that none of the slots in column *j* can be assigned to session (*s,d*) to constraints on transmitter or receiver tuning times. Sometimes we'll be able to resolve post-column conflict as discussed in the next section.

3.3 Tuning, Modulation, Turn On/Off Time

Since each transmitter and receiver can operate on any of the *W* wavelength channels, there is a finite tuning/modulation/turn on-off overhead time (will be referred to as tuning overhead time) required as it moves from wavelength channel to wavelength channel. This overhead varies as a function of the distance between the two wavelengths, and can be a source of inefficiency in the network. The following describes some possible methods that can be used to reduce the capacity lost to this tuning overhead (Refer to Figure 3.3.1 as we gradually build up our system from an empty one).



Wavelength	1	2	3	4	5	......	T
λ_W	(c,b)						
⋮							
$\lambda 3$	(a,e)	(b,e)		(a,f)			
λ_2							
λ_1							

Figure 3.3.1 Session Assignment

- *Avoid Tuning Overhead*

Consider an empty system and a request of one slot for session (a,e)[1]. Assume slot (3,1) is assigned to this session. A new request of one slot for session (a,f) comes in, and assume this is the only session involving node *f*'s receiver. If we assign session $(a.f)$ to any of the empty slots in row 3, no tuning overhead will occur since node *a*'s transmitter is already at wavelength 3 and node *f*'s receiver can be set to this wavelength. Any other assignment will result in tuning overhead for node *a*'s transmitter. Let's assign the session to slot (3,4) as in Figure 3.3.1.

- *Resolve Pre-column Conflict / Post-column Conflict*

If request for a one slot session (b,e) comes in, there is *Pre-column Conflict* over column 2 due to node *e*'s receiver conflict on wavelength λ_3. In this case we can still assign the session to slot (3,2) since node *e*'s receiver is already tuned to wavelength 3, so no additional tuning is needed. We say the *Pre-column Conflict* over column 2 is resolved. However, if slot (3,2) is already assigned to some other session, the *Pre-column Conflict* over column 2 is unresolvable, and none of the slots in column 2 can be used. The same applies to *Post-column Conflict.* The most restrictive situation is when *Pre-column Conflict* due to transmitter and receiver as well as *Post-column Conflict* due to transmitter and receiver occur. There are essentially three independent sessions between the same source and destination nodes, two already assigned and one needs to be assigned. The two assigned sessions are one slot number apart. The conflicts are resolvable only if the two assigned sessions have the same wavelength channel, and the slot between them with that wavelength channel is free to be assigned to the new session. We will end up with three sessions assigned to consecutive slots.

[1] session (a,e) is the notation for a communications channel from source node '*a*' to destination node '*e*'.

- *Off-line Tuning*

If the tuning can be done off-line, and assuming the modulation and turn on-off time is negligible, we can substantially reduce tuning overhead conflicts. A request for a one slot session (*c,e*) comes in, and can be assigned to slot (3,3) using off-line tuning technique. As shown in Figure 3.3.1, even though node *c*'s transmitter is tuned to wavelength *W* for session (*c,b*) during slot 1 time, it can be tuned to wavelength 3 during its idle period of slot 2, and be ready for session (*c,e*) assigned slot (3,3).

- *Combine Tuning into Data Slot*

If the tuning time and the requested slots can be a fractional number, we can combine them into a slot. To illustrate the point, let's assume the tuning overhead uses 0.3 of a slot, and the session requires 1.6 slots. Instead of assigning one slot for tuning and another two slots for data for a total of three slots, we can assign two slots. The session will use the first 0.3 of the slot for tuning, followed by data immediately.

For our study, we will assume all tuning overhead and sessions require an integer number of slots. More specifically we will assume one slot for tuning overhead should it ever be needed.

3.4 Session Distribution between OTs

The L0 subnet is basically considered a campus-wide Local Area Network. The traditional LAN is built upon the client/server model. A server could be a file server, printer, gateway, time-sharing system, etc. One study of LAN traffic has shown that for one network measured, these identifiable servers sent about 69% and received about 73% of the packets over one typical day[5].

For either a client/server or distributed model, the L0 subnet may be highly compartmentalized. The users in the same department are more likely to talk to each other then to users outside their department. The study previously referred to reported that 72% of traffic measured was intranet or intradepartmental packets[5].

In our study, we will initially concentrate on a uniformly distributed traffic model. After we get an understanding of the uniform system, we will introduce a client/server traffic model into the subnet and see how it influences the system characteristics.

3.5 Queueing / Blocking System

In the queueing system model, the scheduler has a queue(s) to hold requests that cannot be immediately satisfied. It will try repeatedly until resource is found. In the blocking system, the scheduler does not have any queue. Any unsatisfied request is discarded, and needs to be regenerated by the terminal.

3.6 Scheduling Approach

- Contiguous-Slot (*CS*) and Random-Slot (*RS*) Assignment:

A session requesting multiple slots $\{(\lambda_i, j)\}$, tuning overhead included, is said to have *CS* assignment if, when arranging *j* in increasing order, the increment is always one. A scheduling algorithm is implementing *CS* if all sessions have *CS* assignment, otherwise it is implementing *RS*.

- Single-Wavelength (*SW*) and Multiple-Wavelength (*MW*) Assignment:

A session requesting multiple slots $\{(\lambda_i, j)\}$, tuning overhead included, is said to have *SW* assignment if all λ_i's are identical. A scheduling algorithm is implementing *SW* if all sessions have *SW* assignment, otherwise it is implementing *MW*.

By combining variations on wavelength and slots, a scheduling algorithm can implement one of the four assignments, *SW-CS*, *SW-RS*, *MW-CS*, and *MW-RS*. In this work, we will be focusing on the most constraint *SW-CS* and the least constraint *MW-RS* assignment.

3.7 Scheduling Algorithms Considered

The following is the outline of the three different scheduling approaches to tuning overhead that we will consider.

- *Contiguous L Assignment Algorithm (CL Assignment)*

This scheduling algorithm allows no tuning overhead time on the common channel, and is used in conjunction with *SW-CS* assignment. When a session requests *L* slots per frame, it gets *L* slots. The tuning overhead is avoided by using the off-line tuning technique or by resolving *Pre-column* and *Post-column Conflicts* if possible. Otherwise the session request is either rejected or queued depending on the system. The advantage of this approach is algorithmic simplicity. There is no fragmentation to worry about, and a single class (*L* slots per session per frame) system with *W* wavelength channels and frame size *T* can be treated as a single class of one slot per session per frame system with *W* wavelength channels and frame size $\lfloor T/L \rfloor$ due to the *SW-CS* approach used. The disadvantage of this approach is that in a system with a small number of users, this algorithm can have low channel utilization because of rejections due to unresolved *Pre-column* or *Post-column Conflicts*. So that even if a block of *L* slots is free, it goes unused and thus reduces the channel utilization. The larger the *L*, the more pronounce the effect becomes. However when the number of users in the system is large, the algorithm should perform well since minimal tuning conflict is expected.

- *Contiguous L+1 Assignment Algorithm (L+1 Assignment)*

In this algorithm we always allocate one additional slot for tuning overhead, therefore there is no *Pre-column* or *Post-column Conflicts* to worry about. It is used in conjunction with *SW-CS* algorithm, so all sessions are assigned a block of $L+1$ slots. As in the *Contiguous L Assignment Algorithm*, a single class (L slots per session per frame) system with W wavelength channels and frame size T can be treated as a single class of one slot per session per frame system with W wavelength channels and frame size $\lfloor T/(L+1) \rfloor$. Compared to *Contiguous L Assignment Algorithm*, since an overhead of one slot per L slots is introduced, the channel utilization will degrade in a system with large number of users. In a system with small population the cost in overhead can be traded off against the loss due to rejection resulting from unresolved *Pre-column* and *Post-column Conflicts*. The channel utilization can actually improve compared to *Contiguous L Assignment Algorithm.* The exact nature of the improved efficiency also depends on the value of L.

- *Random L Assignment Algorithm (RL Assignment)*

This algorithm uses *MW-RS* assignment and off-line tuning technique. The slot used for one session's off-line tuning can be used for another session's on-line data. This is made possible by the use of *RS* assignment. Off-line tuning also means that all sessions are assigned L slots as requested. This approach introduces fragmentation and additional algorithm complexity, but we hope it will utilize the available slots more fully and give added performance.

3.8 System Characteristics and Traffic Models

In this section, we define system characteristics and traffic models for a system with W number of wavelength channels and frame size T. The number WT describes the total number of slots per frame available for use. Both W and T are deterministic. We assume that each wavelength channel has fixed capacity, so the system capacity is linearly proportional to the number of wavelength channels.

A/B/C/D/E System Characteristics

- The first parameter A indicates the session arrival process. It is G for a general distribution of interarrival times, M for memoryless, specifically the Poisson process; and D for deterministic interarrival time.

- The second parameter B indicates the distribution of session service time (session hold time). It will be M, G, and D for exponential, general, and deterministic probability distribution, respectively.

We assume that successive interarrival times and service times are statistically independent of each other.

- The third parameter C indicates the distribution of the number of slots required per session. It will be M, G, and D for exponential, general, and deterministic probability distribution, respectively. When the number of slots required per session per frame is one of s predetermined values $\{ L_1, \dots , Ls \}$, we say the system is s-class, and is denoted by numeral *s*. So a number 1 means single class, and 2 means two class system.

- The fourth parameter *D* indicates if it is a blocking(*B*) or a queueing(*Q*) system.

- The fifth parameter *E* gives the scheduling algorithms described in Section 3.7. It will be *CL* for "Contiguous L Assignment Algorithm", *L+1* for "Contiguous L+1 Assignment Algorithm", and *RL* for "Random L Assignment Algorithm".

In this thesis, mathematical estimations/bounds are derived and OPNET simulations are run for the following system traffic models. The models are chosen since they represent some typical aspects of the system. The simplest model of single class and uniform traffic is first analyzed to give us some understanding of the efficiency of the different algorithms. To get a more accurately approximation of the real system traffic, we analyzed the two class and uniform traffic system. This system is simple enough to study yet it represents the multiclass SONET traffic that AON may carry. Finally to approximate the client/server situation, we analyze the single class and client/server traffic and compare that to single class and uniform traffic. From the study of these three systems, we want to observe how well the three scheduling algorithms perform, and when it is important to take tuning overhead into consideration.

System Traffic

1. Single Class and Uniform Traffic System

We will analyze the blocking and queueing systems respectively for the single class and uniform traffic system. For blocking system, *M/M/1/B/CL, M/M/1/B/L+1*, and *M/M/1/B/RL* characteristics are used. Sessions arrive according to a Poisson process with rate λ, exponential session hold time with mean $1/\mu$, and all sessions require *L* slots per frame. We also assume all sessions are uniformly distributed among optical terminals in the network. We will obtain a mathematical approximation and simulation result for *M/M/1/B/L+1* system. For *M/M/1/B/L+1* and *M/M/1/B/RL* systems, mathematical bounds and simulation results in terms of channel utilization and blocking probability are obtained. The queueing system is the same as the blocking one except the unsatisfied session requests are queued. Again, mathematical bounds and simulation results for queueing delay and the average queue size are obtained for *M/M/1/Q/CL, M/M/1/Q/L+1*, and *M/M/1/Q/RL* systems.

This study will allow us to check our simulation and mathematical results against each other, and show how effectively the three scheduling algorithms deal with tuning overhead.

2. Two Class and Uniform Traffic System

We will study *M/M/2/B/CL, M/M/2/B/L+1*, and *M/M/2/B/RL* for blocking system and *M/M/2/Q/RL* for queueing system. Sessions arrive according to a Poisson process with rate λ_1 for those requiring L_1 slots per frame and λ_2 for those requiring L_2 slots per frame.

Session hold time is exponential with mean $1/\mu$ for both types of sessions. All sessions are uniformly distributed among optical terminals in the network. In the *Contiguous L* and *L+1 Assignment Algorithms*, the system is divided into two subsystems according to the relative traffic load of each type of sessions. Each subsystem serves only one type of sessions using *Contiguous L* or *L+1 Assignment Algorithm.* The advantage of this approach is that each type of sessions has its fair share of the system resource. The disadvantage is that accurate estimate of the relative traffic is needed in order to divided up the system resource. This problem can be solved by using *Random L Assignment Algorithm* which has additional complexity and may not be as fair. We will compare the three scheduling algorithms for the blocking system, and show that the *Random L Assignment Algorithm* has higher efficiency. For the queueing system, only *M/M/2/Q/RL* will be simulated.

3. Single Class and Client/Server Traffic System

This system is essentially the same as system 1. Only we will introduce server nodes into the system (N_s server nodes among a total of N nodes including servers). So in addition to uniform traffic among regular nodes, we have client/server traffic between the server nodes and the regular nodes (i.e. client nodes). The traffic break up is such that each server generates *st* percentage of sessions to and another *st* percentage from clients. The rest of the $1\text{-}N_s{*}2{*}st$ percentage of the traffic are among clients (this is the uniform part). Notice that since each transmitter or receiver can only use one out of W wavelength channels at a given time, *st* has to be less than or equal to *1/W* before server's transmitter and receiver become system's bottleneck. We want to see how robust the result is from system 1.

3.9 Problem Summary

In this thesis, we will study scheduling algorithms for intra-L0, point-to-point, Type-B sessions in the AON network under the following assumptions:

- Zero propagation delay in the network.
- Zero processing delay in session scheduling.
- One tunable transmitter and tunable receiver per OT.
- The total system capacity is proportional to the number of wavelength channels.
- All tuning overhead and sessions require an integer number of slots.
- Poisson arrival process and exponentially distributed session service time.
- Successive interarrival time and session service time are statistically independent.

The approach we take is by studying the three "typical" system traffic models specified in Section 3.8, the single class uniform traffic blocking and queueing system, the two class uniform traffic blocking and queueing system, and the single class client/server traffic blocking and queueing system. The scheduling algorithms employed in studying these models are specified in Section 3.7, the *Contiguous L Assignment Algorithm*, *Contiguous L+1 Assignment Algorithm*, and *Random L Assignment Algorithm.*

What we will conclude from our study is that when the number of users in the system is relatively large compared to the number of wavelength channels, the effect of tuning overhead is minimal, and the efficiency closed to the bound can be obtained by *Random L Assignment Algorithm* which uses off-line tuning and *MW-RS* assignment approach.

The simulations will be run for system with frame size 128 as in the testbed built by the AON consortium. The number of wavelength channels will be 2, 4, and 8. The number of nodes in the system will be 8 and 40. So the ratio of nodes to wavelength channels takes on the value of 1, 2, 4, 5, 10, and 20. From this range of ratios we will be able to conclude how large the number of nodes in the system has to be, compared to the number of wavelength channels, for the effect of tuning overhead to be negligible. Finally the session throughput requirement will take on the value of 1, 3, and 12 in single class system to help us understand the effect of throughput requirement on the efficiency of scheduling algorithms. For two class system, the session throughput requirement will be 3 and 12. We vary the relative traffic load of each type of the session to understand its effect on the efficiency of scheduling algorithms. For single class client/server system, we fixed the total number of nodes to 40, and wavelength channels to 8, while the number of servers takes on 1 and 3, and server traffic percentage *st* takes on the value of 0.05 and 0.1 (the maximum *st* is 1/8 before server's transmitter and receiver become system bottleneck). We will compare the results to single class uniform traffic situations.

Chapter 4 Mathematical Approximations and Bounds

In this chapter we derive mathematical approximations or bounds for single class uniform traffic and two class uniform traffic systems.

4.1 Offered Load

Single Class

Assuming sessions arrive with Poisson rate λ, service time (i.e. session holding time) is exponentially distributed with mean $1/\mu$, and the session throughput requirement is L slots per frame.

Let the capacity of each wavelength channel be C_s bps, so the total system capacity is WC_s bps. Let the number of bits transmitted by each session be exponentially distributed with mean of K bits per session. If a session can use an entire wavelength channel, it takes on average $1/\mu_0 = K/C_s$ second to transmit. If it acquires only L slots per frame, it takes on average $1/\mu = KT/C_sL$ second to transmit. Therefore $1/\mu = T/\mu_0L$. The offered load is the percentage of system capacity used, $\rho = \lambda$(sessions/sec)$\cdot K$(bits/session) / WC_s(bps) = $\lambda L/\mu WT$.

In our simulations, we normalize $1/\mu_0$ to one, meaning the mean session holding time is one second had it been using an entire wavelength channel. When sessions only use L of the T slots, the mean session holding time is $1/\mu = T/L$.

$$\rho = \frac{\lambda}{\mu} \cdot \frac{L}{WT} \qquad \textit{and} \qquad \frac{1}{\mu} = \frac{1}{\mu_0} \cdot \frac{T}{L} \tag{4.1.0.1}$$

Two Class

Assuming sessions with throughput requirement L_1 slots per frame arrive with Poisson rate $\lambda_1 = \alpha\lambda$, and those with throughput requirement L_2 slots per frame arrive with Poisson rate $\lambda_2 = (1-\alpha)\lambda$. Where λ is the overall session arrival rate, and α is the percentage of arrivals that are L_1 type sessions. Assume the service time (i.e. session holding time) is exponentially distributed with mean $1/\mu$ for all sessions.

Let the capacity of each wavelength channel be C_s bps, so the total system capacity is WC_s bps. Let the number of bits transmitted by type L_1 and L_2 sessions be exponentially distributed with mean of K_1 bits and K_2 bits per session respectively. Since type L_2 sessions have throughput requirement L_2/L_1 times that of type L_1 sessions and the session holding time is the same for both, we must have $K_2 / K_1 = L_2 / L_1$. When a type L_1 session acquires only L_1 slots per frame, it will take $1/\mu = K_1T/C_sL_1$ second to transmit. Similarly $1/\mu = K_2T/C_sL_2$, The offered load is the percentage of system capacity used, $\rho =$

{ λ_1(sessions/sec)$\cdot K_1$(bits/session) + λ_2(sessions/sec)$\cdot K_2$(bits/session) } / WC_s(bps) = $\lambda\{\alpha L_1+(1\text{-}\alpha)L_2)\}$ / μWT.

If we normalize the service time against type L_1 sessions (i.e. assuming that each type L_1 session can use an entire wavelength channel), the mean session holding time is $1/\mu_0 = K_1/C_s$. Since in reality each type L_1 session uses only L_1 of T slots, the mean session holding time is $1/\mu = T/\mu_0 L_1$. Similarly we will have $1/\mu = T/\mu_0 L_2$ and $1/\mu_0 = K_2/C_s$ if normalizing the service time against L2 type sessions. In either case we will set $1/\mu_0$ to one when running simulations.

$$\rho = \frac{\lambda}{\mu} \cdot \frac{\alpha L_1 + (1-\alpha) L_2}{WT} \quad where \quad \frac{1}{\mu} = \frac{1}{\mu_0} \cdot \frac{T}{L1} \quad or \quad \frac{1}{\mu} = \frac{1}{\mu_0} \cdot \frac{T}{L2} \qquad (4.1.0.2)$$

depending on normalization of session service time against type L_1 or L_2 services.

4.2 Single Class, Uniform Traffic System

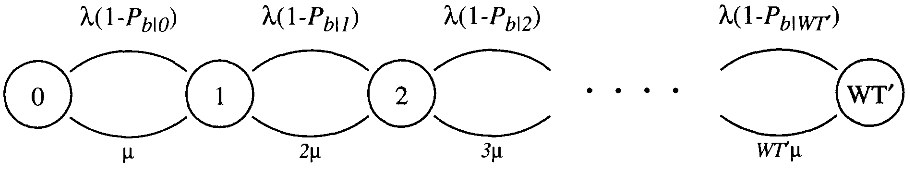
In this section, we derive mathematical approximation for blocking system and mathematical upper bound for queueing system.

4.2.1 Blocking System

In Appendix A.1 we derived $P_{b|Ns}$, the probability that a new session request will be rejected given the number of sessions in service *Ns*. Two assumptions were made, the existence of *column conflict* only and that all *Ns* sessions are equally distributed over slots in the system. The *column conflict* only assumption is valid for *Contiguous L+1 Assignment Algorithm* but not for *Contiguous* and *Random L Assignment Algorithms*. The assumption of equally distributed sessions over all slots in the system is a statistical approximation of the system. Therefore the $P_{b|Ns}$ thus derived is a conservative approximation but not the minimum for the system. And the channel utilization and the blocking probability derived below using this value of $P_{b|Ns}$ are also not the lower bounds but conservative approximations only.

Mathematical Approximation for M/M/1/B/L+1

By using *SW-CS* algorithm, each row can accommodate a maximum of $T' = \lfloor T/(L+1) \rfloor$ sessions. If we align these sessions at the boundary, we can consider the system as that of frame size T' and all sessions require one slot per frame. The system could be modeled by using Markov Chain.



$\lambda(1\text{-}P_{b|0})$
$\lambda(1\text{-}P_{b|1})$
$\lambda(1\text{-}P_{b|2})$
$\lambda(1\text{-}P_{b|WT'})$
0
1
2
. . . .
WT'
μ
2μ
3μ
$WT'\mu$

Figure 4.2.1.1 *M / M / WT' / WT'* System

As shown in Figure 4.2.1.1, the system is equivalent to *M/M/m/m* blocking system with $m = WT'$. Let state i be the number of sessions in service, P_i be the probability that system will be in state i when in equilibrium. Obviously the system will advance from state i to state $i+1$ only if the new session request is accepted given state i. And the probability of acceptance is $1-P_{b|i}$, where $P_{b|i}$ is derived in Appendix A.1. Solving the following balance equation, and express it in terms of offered load $\rho = (\lambda/\mu)(L/WT)$,

$$Pi \cdot \lambda(1 - Pb|\,i) = P(i+1) \cdot (i+1)\mu$$

we have

$$Pi = Po \cdot \left(\rho\frac{WT}{L}\right)^i \cdot \frac{1}{i!} \cdot \prod_{j=0}^{i-1}(1 - Pb|\,j) \qquad for \qquad i = 0, \ldots, WT' \tag{4.2.1.1}$$

where

$$Po^{-1} = 1 + \sum_{i=1}^{WT'}\left(\rho\frac{WT}{L}\right)^i \cdot \frac{1}{i!} \cdot \prod_{j=0}^{i-1}(1 - Pb|\,j) \tag{4.2.1.2}$$

$$CU = \frac{L}{WT} \cdot \sum_{i=0}^{WT'} i \cdot Pi \qquad and \qquad Pb = \sum_{i=0}^{WT'} Pi \cdot Pb|\,i \tag{4.2.1.3}$$

Where *CU* is the channel utilization, and P_b is the blocking probability of the system.

In Figure 4.2.1.2 we have shown the channel utilization and the blocking probability for systems with throughput requirement of one, three, and twelve slots per frame, frame size of 128, two, four, or eight wavelength channels, and eight or forty nodes. The same set of parameters will be used for simulation. The calculation is based upon the formulas above, the program is shown in Appendix A.2.1.

We can see from our analysis that the system performance depends on the ratio of *N/W*, where *N* is the number of nodes and *W* the wavelength channels in the system. We denote this ratio as γ. It describes the size of the system population relative to the wavelength number. The larger the population the less likely the session request is blocked by transmitter or receiver conflict, and the better the performance. From Figure 4.2.1.2 we see that a system with small γ performs considerably worse. However once $\gamma >> 1$, the relative size of γ does not seem to be critical.

Notice that due to the SW-CS assignment nature of the algorithm, there are wasted slots in each row. Combining this inefficiency and the one slot tuning overhead per session, we can derive the maximum channel utilization for sessions when ignoring the transmitter and receiver conflicts. Assuming frame size of 128 and throughput requirement of *L* slots per frame, the maximum channel utilization is $L*\lfloor T/(L+1)\rfloor / T$. So for *L* equals to one, three and twelve, the maximum channel utilization is 0.5, 0.75 and 0.84 respectively (shown as solid horizontal lines in Figure 4.2.1.2). The results for systems with large γ is very close to this absolute bound.

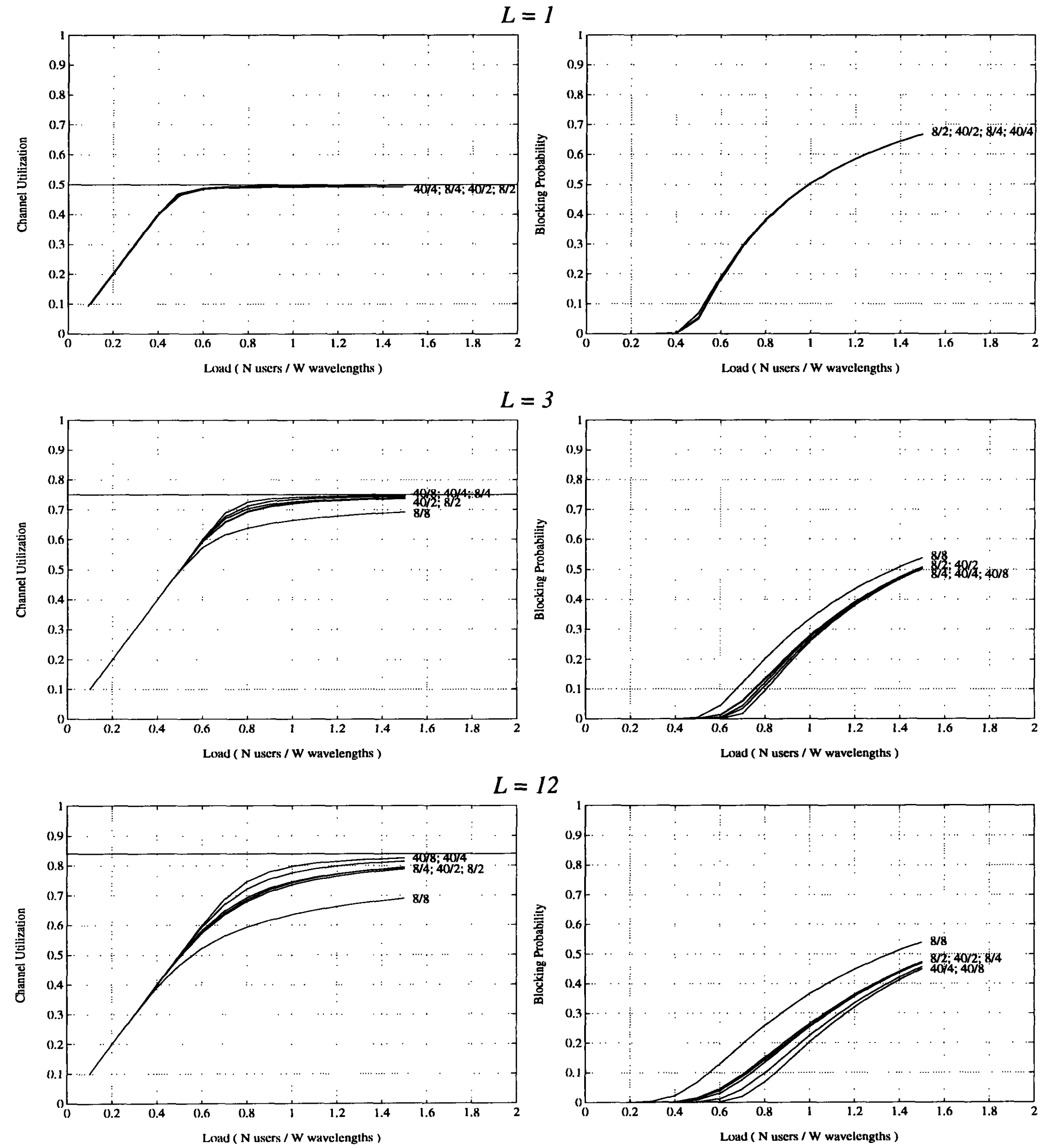


Figure 4.2.1.2 Mathematical Approximation for *M/M/1/B/L+1* System

Mathematical Approximation for M/M/1/B/CL

For *Contiguous L Assignment Algorithm*, the existence of *pre-column* and *post-column conflict* increases the probability that a new session will be rejected to above the $P_{b|Ns}$ value derived in Appendix A.1. Therefore $P_{b|Ns}$ is a conservative approximate lower bound for the system. Under the condition that we have large number of nodes (compared to the wavelength number), the occurrence of *column conflict* as well as *pre-column* and *post-column* conflict is rare. Therefore $P_{b|Ns}$ can be a tight approximate lower bound for the system.

We use the same Markov Chain approach as in M/M/1/B/L+1 to solve for the system parameters. The only difference is now $T' = \lfloor T/L \rfloor$.

In Figure 4.2.1.3, we presented the results for systems with forty nodes, four wavelength channels, frame size of 128, and throughput requirement of one, three, and twelve slots per frame. Notice that M/M/1/B/CL system of large number of users is more efficient at steady state compared to M/M/1/B/L+1. The program in Appendix A.2.1 was used to do the calculations.

Notice that due to the SW-CS assignment nature of the algorithm, there are wasted slots in each row. Assuming frame size of 128 and throughput requirement of L slots per frame, the maximum channel utilization is $L{*}\lfloor T/L \rfloor / T$. So for L equals to one, three and twelve, the maximum channel utilization is 1.0, 0.98 and 0.94 respectively (shown as dotted horizontal lines in Figure 4.2.1.3). The results shown in Figure 4.2.1.3 are close to this absolute bound.

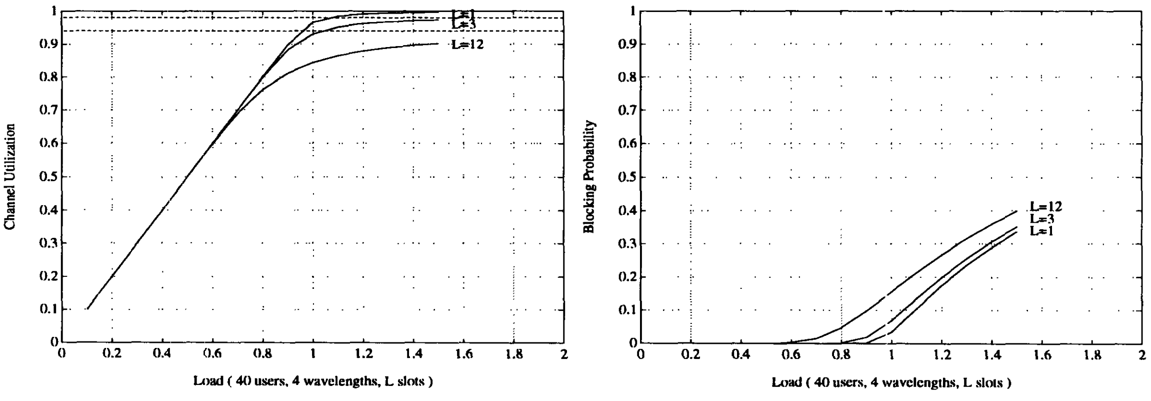


Figure 4.2.1.3 Mathematical Approximation for *M/M/1/B/CL* System

Mathematical Bounds for M/M/1/B/RL

Ignoring transmitter and receiver conflicts, we can use *M/M/m/m* model to obtain the bound for the M/M/1/B/RL system. The number of servers $m = \lfloor WT/L \rfloor$. The channel utilization CU and the blocking probability Pb are as follows,

$$CU = \frac{L}{WT} \cdot \frac{\sum_{n=0}^{m} n \cdot (\lambda/\mu)^n / n!}{\sum_{n=0}^{m} (\lambda/\mu)^n / n!} \quad \text{and} \quad Pb = \frac{(\lambda/\mu)^m / m!}{\sum_{n=0}^{m} (\lambda/\mu)^n / n!} \qquad (4.2.1.4)$$

where $\lambda/\mu = \rho WT/L$.

A system of WT slots can accommodate a maximum of $\lfloor WT/L \rfloor$ sessions as in a system of $\lfloor WT/L \rfloor$ servers. When taking transmitter and receiver conflicts into consideration, the system resource would be used less efficiently. So *M/M/m/m* model provides the upper bound for channel utilization and lower bound for blocking probability. And the absolute maximum channel utilization is $L^*\lfloor WT/L \rfloor /WT$. In our system, the frame size is 128. For *L* of 1, the maximum channel utilization is 1.0; for *L* of 3, the maximum channel utilization is 0.996, 0.996, 0.999 for *W* of 2, 4 and 8 respectively; for *L* of 12, the maximum channel utilization is 0.984, 0.984, 0.996 for *W* of 2, 4 and 8 respectively.

In Figure 4.2.1.4 channel utilization and blocking probability are plotted for systems with two, four and eight wavelength channels, frame size of 128, and throughput requirement of one, three, and twelve slots per frame. Notice that the channel utilization approaches the absolute maximum value. The program for calculating the result is shown in Appendix A.2.1.

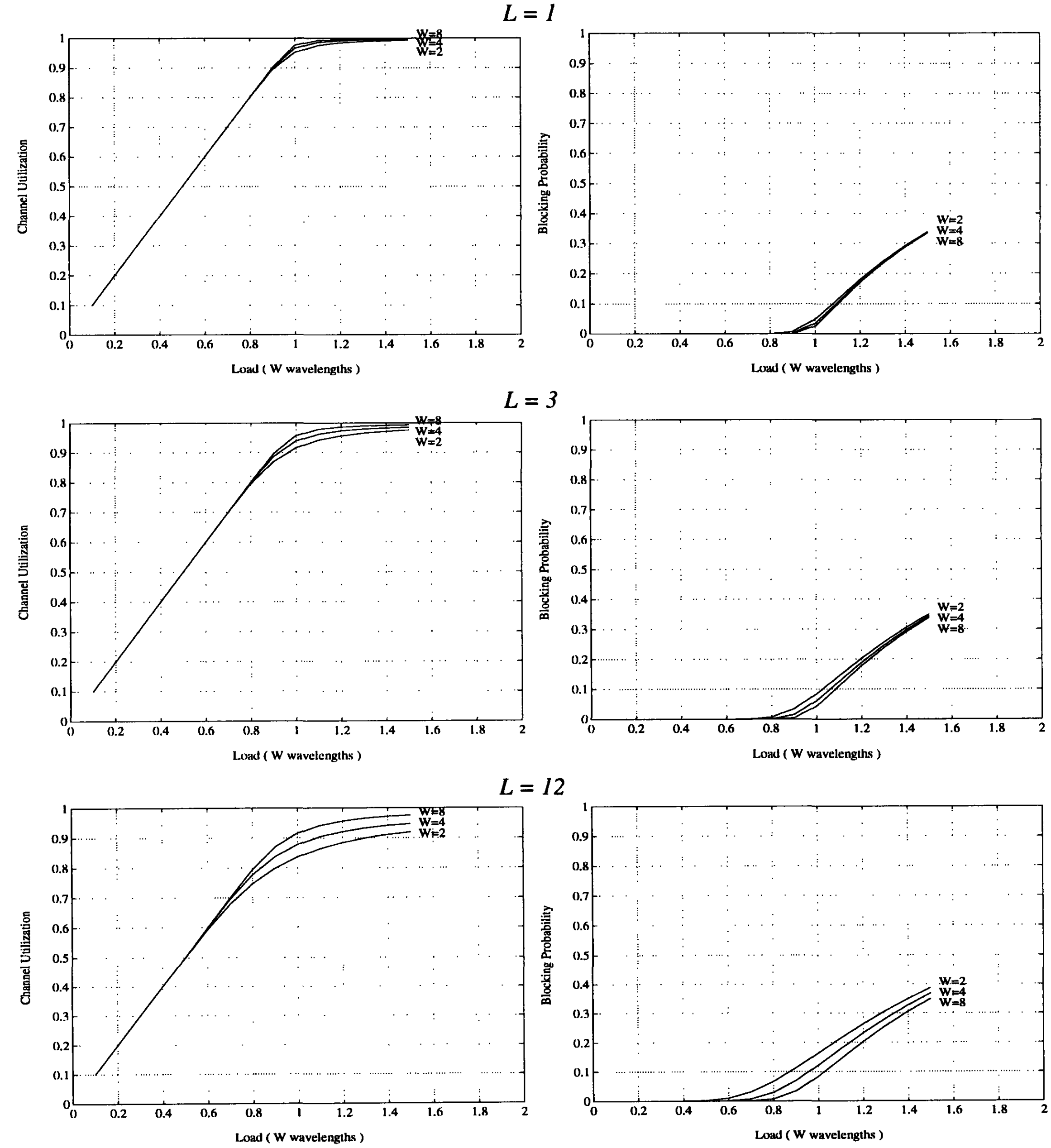


Figure 4.2.1.4 Mathematical Bounds for *M/M/1/B/RL* System

4.2.2 Queueing System

Ignoring transmitter and receiver conflicts, results from *M/M/m* system are used as bounds for M/M/1/Q/* systems. For *M/M/m* system, the following equations hold[7]:

$$Pq = \frac{p_0 (\lambda/\mu)^m}{m!\,(1-\lambda/m\mu)}; \qquad Nq = \frac{(\lambda/m\mu)\,Pq}{1-\lambda/m\mu}; \qquad Wq = \frac{(\lambda/m\mu)\,Pq}{\lambda\,(1-\lambda/m\mu)} \tag{4.2.2.1}$$

$$p_0 = \left\{ \sum_{k=0}^{m-1} \frac{(\lambda/\mu)^k}{k!} + \frac{(\lambda/\mu)^m}{m!\,(1-\lambda/m\mu)} \right\}^{-1} \tag{4.2.2.2}$$

where P_q is the probability that an arriving session has to wait in queue, N_q is the average number of sessions in queue, W_q is the average waiting time in queue of a session, and $\lambda/\mu = \rho WT/L$.

For M/M/1/Q/L+1 system, the number of servers $m = W\lfloor T(/L+1)\rfloor$; for M/M/1/Q/CL system, $m = W\lfloor T/L\rfloor$; and for M/M/1/Q/RL system, $m = \lfloor WT/L\rfloor$. The results thus obtained, N_q and W_q, are lower bounds for the system since the value used for m is the maximum number of sessions the system can serve at any time for each of the scheduling algorithm. With the transmitter and receiver conflicts taken into consideration, the equivalent number of servers are smaller and the system is less efficient, which translates into larger queue size and longer waiting time in queue.

The program used for calculation is included in Appendix A.2.2.

Mathematical Bounds for M/M/1/Q/L+1

The number of servers m equals to $W\lfloor T/(L+1)\rfloor$ as mentioned earlier. The arrival rate used in calculation is $\lambda=\rho W$ as obtained from Equation (4.1.0.1) and by setting $1/\mu_0$ to one as in simulations (W is the number of wavelength channels).

Same as that in M/M/1/B/L+1 system, the maximum load the system can handle without running into infinite queue and delay is $L^*\lfloor T/(L+1)\rfloor/T$. So for L equals to one, three and twelve, the maximum channel utilization is 0.5, 0.75 and 0.84 respectively (shown as vertical solid lines in Figure 4.2.2.1).

The results for system with frame size 128, wavelength channels two, four, eight, and session throughput requirement of one, three, twelve slots per frame are shown in Figure 4.2.2.1. Notice that the system approaches the absolute bounds mentioned.

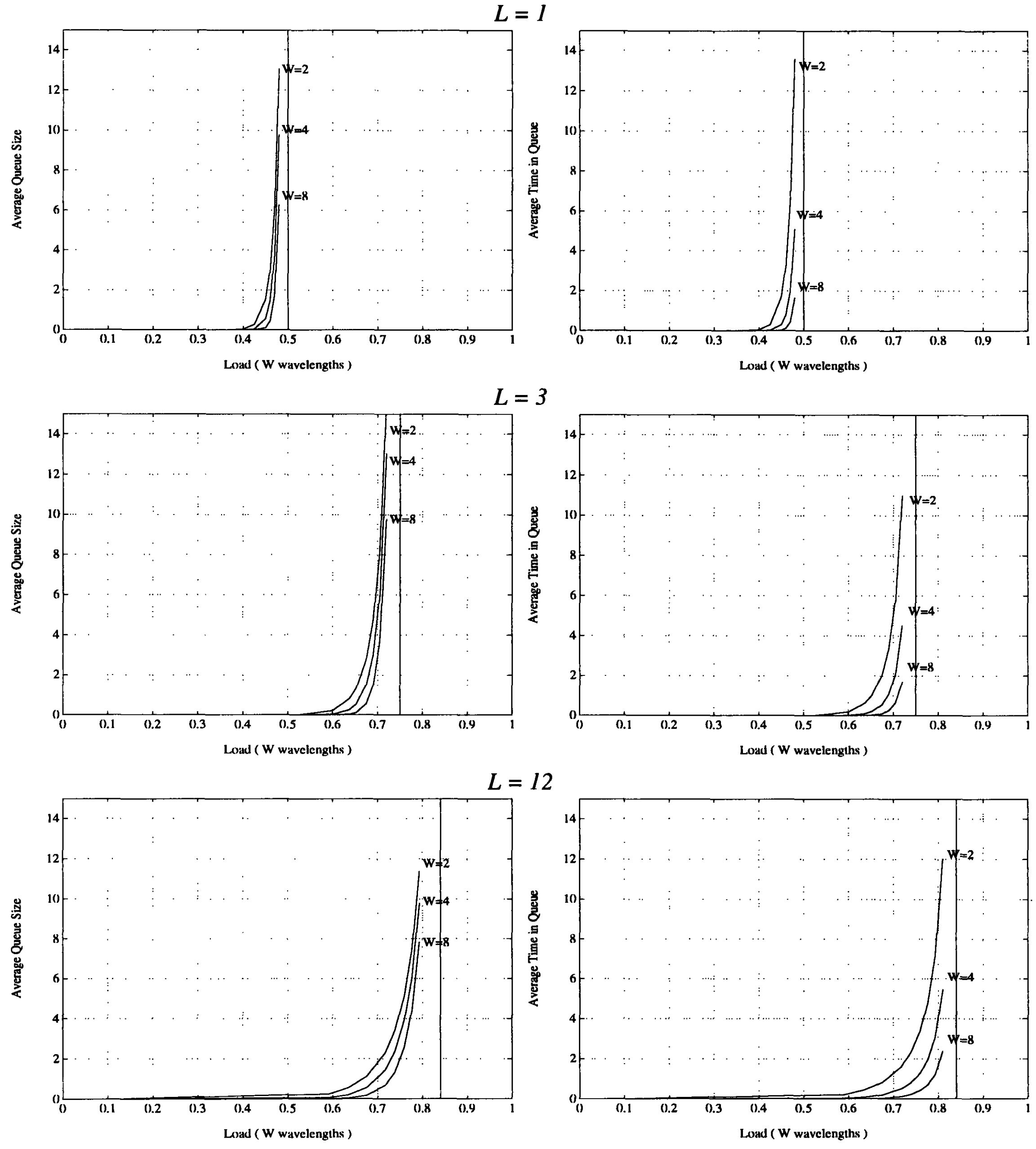


Figure 4.2.2.1 Mathematical Bounds for *M/M/1/Q/L+1* System

Mathematical Bounds for M/M/1/Q/CL

The number of servers *m* equals to $W\lfloor T/L \rfloor$ as mentioned earlier. The arrival rate used in calculation is $\lambda = \rho W$ as obtained from Equation (4.1.0.1) and by setting $1/\mu_0$ to one as in simulations (*W* is the number of wavelength channels).

Same as that in M/M/1/B/L system, the maximum load the system can handle without running into infinite queue and delay is $L*\lfloor T/L \rfloor/T$. So for *L* equals to one, three and twelve, the maximum channel utilization is 1.0, 0.98 and 0.94 respectively (shown as vertical solid lines in Figure 4.2.2.2).

The results for system with frame size 128, wavelength channels two, four, eight, and session throughput requirement of one, three, twelve slots per frame are shown in Figure 4.2.2.2. Notice that the system approaches the absolute bounds mentioned.

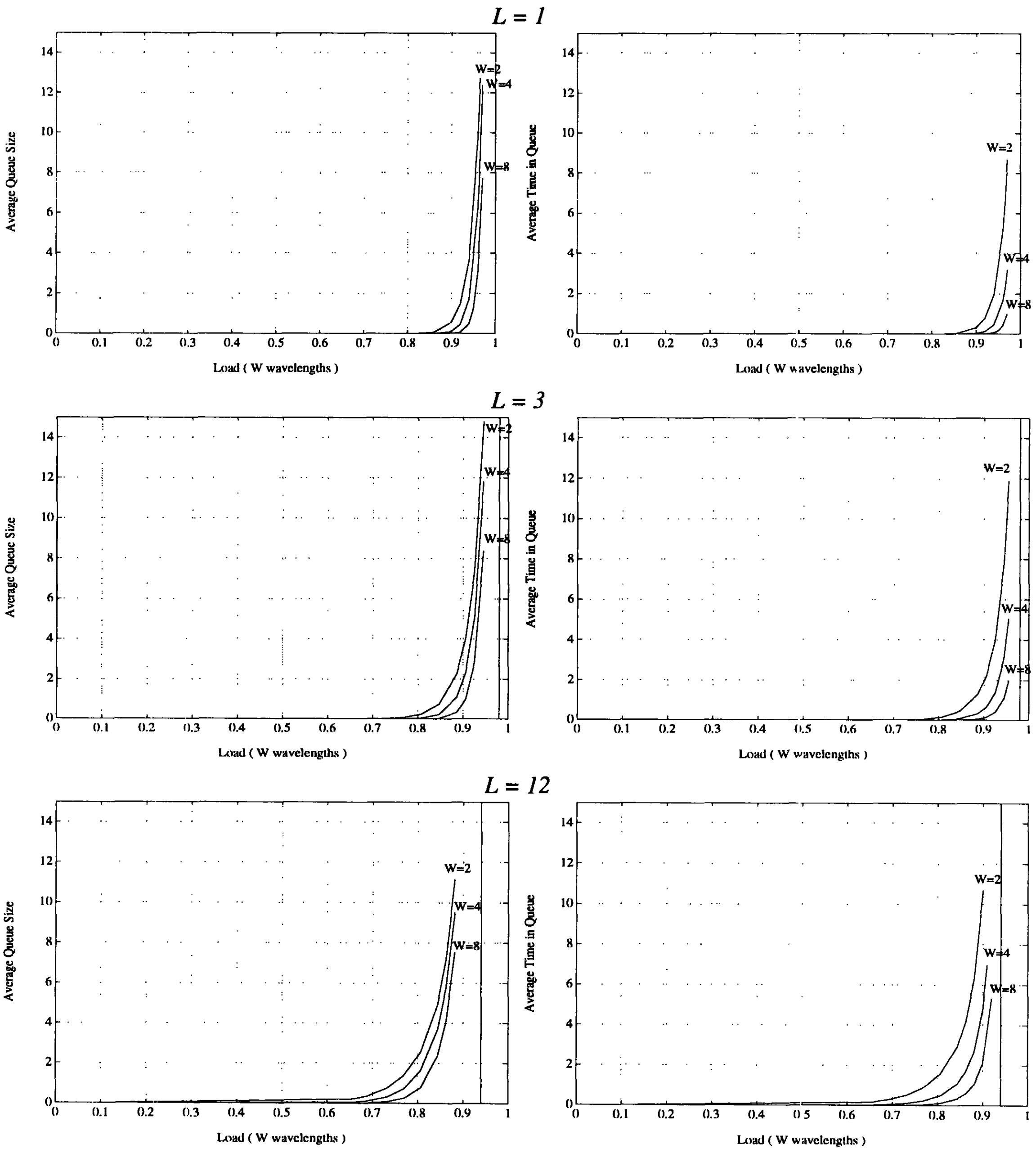


Figure 4.2.2.2 Mathematical Bounds for *M/M/1/Q/CL* System

Mathematical Bounds for M/M/1/Q/RL

The number of servers m equals to $\lfloor WT/L \rfloor$ as mentioned earlier. The arrival rate used in calculation is $\lambda=\rho W$ as obtained from Equation (4.1.0.1) and by setting $1/\mu_0$ to one as in simulations (W is the number of wavelength channels).

Same as that in M/M/1/B/R system, the maximum load the system can handle without running into infinite queue and delay is $L*\lfloor WT/L \rfloor/WT$. In our system, the frame size is 128. For L of 1, the maximum channel utilization is 1.0; for L of 3, the maximum channel utilization is 0.996, 0.996, 0.999 for W of 2, 4 and 8 respectively; for L of 12, the maximum channel utilization is 0.984, 0.984, 0.996 for W of 2, 4 and 8 respectively.

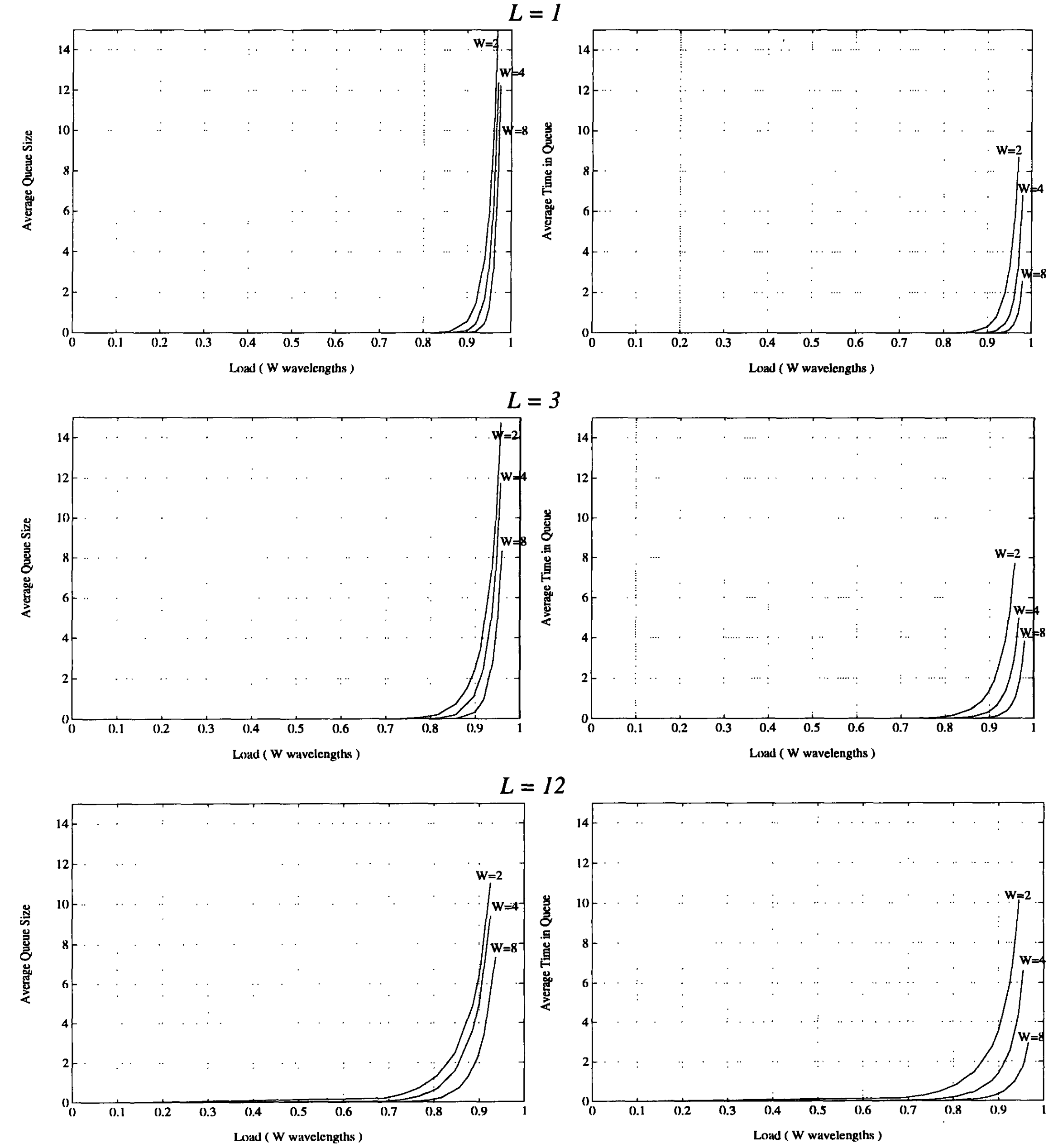


Figure 4.2.2.3 Mathematical Bounds for *M/M/1/Q/RL* System

The results for systems with frame size 128, two, four, eight wavelength channels, and session throughput requirement of one, three, twelve slots per frame are shown in Figure 4.2.2.3. Notice that the system approaches the absolute bounds mentioned.

4.3 Two Class, Uniform Traffic System

In this section we derive mathematical bounds for two class, uniform traffic, blocking systems by Markov Chain approach. We will also look at mathematical bounds for two class, uniform traffic, blocking system.

4.3.1 Blocking System

Again we use Markov Chain to analyze the system while ignoring transmitter and receiver conflicts. Let (i_1, i_2) be the state variables for the number of type L_1 and L_2 sessions in service respectively. Let λ_1 and λ_2 be the corresponding arrival rate, and $1/\mu$ the average session holding time, identical for both type of sessions. Furthermore, assume $L_1 < L_2$, and define $i_{1m} = \lfloor WT/L_1 \rfloor$ and $i_{2m} = \lfloor WT/L_2 \rfloor$. We have a two dimensional Markov Chain bounded by $i_1L_1 + i_2L_2 \leq WT$ as shown in Figure 4.3.1.1.

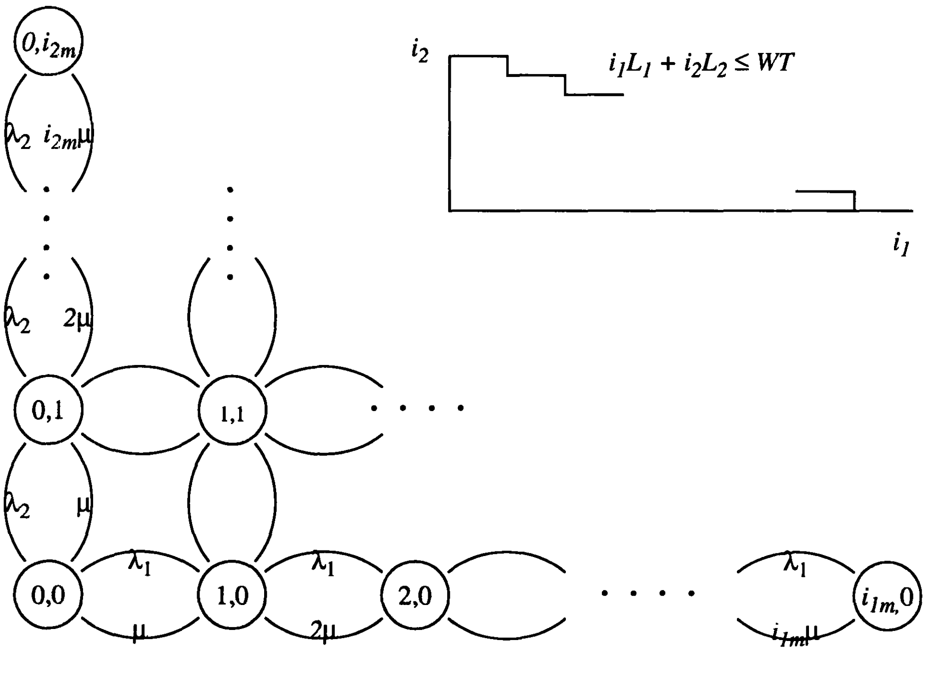


Figure 4.3.1.1 Markov Model for *M/M/2/B/** System

Instead of obtaining an overall blocking probability for the system, we use a weighted blocking probability. The reason is that since type L_1 and L_2 sessions have different throughput requirements, blocking one type of session is not the same as blocking the other as far as the overall impact on the system throughput is concerned. The weighted blocking probability takes that into consideration. We will show shortly that the weighted blocking probability derived here is the lower bound for the system, thus can be used as an yard stick to measure simulation results.

$$\lambda_1 \cdot P_{i_1, i_2} = (i_1 + 1)\mu \cdot P_{i_1+1, i_2} \qquad and \qquad \lambda_2 \cdot P_{i_1, i_2} = (i_2 + 1)\mu \cdot P_{i_1, i_2+1}$$

Solving above balance equations, we can obtain the channel utilization CU, the blocking probability Pb_1 and Pb_2 for type L_1 and L_2 sessions, the weighted blocking probability Pb of the system, and the steady state probability $P_{i1, i2}$ of system in state (i_1,i_2), assuming $L_1 < L_2$.

$$P_{i1, i2} = \frac{\rho_1^{i1}}{i_1!} \cdot \frac{\rho_2^{i2}}{i_2!} \cdot P_{0,0} \tag{4.3.1.1}$$

$$CU = \sum_{i_2=0}^{\lfloor WT/L_2 \rfloor} \sum_{i_1=0}^{\lfloor (WT - i_2L_2)/L_1 \rfloor} \frac{i_1L_1 + i_2L_2}{WT} \cdot P_{i1, i2} \tag{4.3.1.2}$$

$$Pb = \{\alpha L_1 \cdot Pb_1 + (1-\alpha) L_2 \cdot Pb_2\} / \{\alpha L_1 + (1-\alpha) L_2\} \tag{4.3.1.3}$$

$$Pb_1 = \sum_{i_2=0}^{\lfloor WT/L_2 \rfloor} P_{\lfloor (WT - i_2L_2)/L_1 \rfloor, i2} \tag{4.3.1.4}$$

$$Pb_2 = \sum_{i_1=0}^{\lfloor WT/L_1 \rfloor} P_{i1, \lfloor (WT - i_1L_1)/L_2 \rfloor} \tag{4.3.1.5}$$

where

$$P_{0,0} = \left\{ \sum_{i_2=0}^{\lfloor WT/L_2 \rfloor} \frac{\rho_2^{i2}}{i_2!} \sum_{i_1=0}^{\lfloor (WT - i_2L_2)/L_1 \rfloor} \frac{\rho_1^{i1}}{i_1!} \right\}^{-1} \tag{4.3.1.6}$$

$$\rho_1 = \frac{\lambda_1}{\mu} = \frac{\alpha WT\rho}{\alpha L_1 + (1-\alpha) L_2} \qquad and \qquad \rho_2 = \frac{\lambda_2}{\mu} = \frac{(1-\alpha) WT\rho}{\alpha L_1 + (1-\alpha) L_2} \tag{4.3.1.7}$$

The blocking probability Pb_1 and Pb_2 for type L_1 and L_2 sessions are not the lower bounds for the system. They are what one would get when using a "fair" scheduling algorithm which gives equal access of system resources to both type L_1 and L_2 sessions. (By contrast, an "unfair" scheduling algorithm prefers one type of the session over the other. In our system, most of the scheduling algorithms are inherently unfair since sessions with larger throughput requirements are more likely to be rejected not only because they ask more slots at once but also because they are more likely to encounter transmitter and receiver conflicts).

However the weighted blocking probability Pb is the lower bound for the system, it does not matter if the system uses fair or unfair scheduling algorithms. (i.e. $Pb = Pb(fair) = Pb(unfair)$, where $Pb(fair)$ and $Pb(unfair)$ are weighted blocking probabilities for fair and unfair systems respectively). Only when taking transmitter and receiver conflicts into consideration, the weighted blocking probability of the system will be lower than that obtained here. Thus Pb from Equation (4.3.1.3) is indeed the lower bound for the system.

Assuming we are using an unfair scheduling algorithm that increases the blocking probability for type *L1* sessions from $Pb_1(fair)$ by δ (i.e. $Pb_1(unfair) = Pb_1(fair) + \delta$), so for every type *L1* sessions, an additional δL_1 slots are freed up for type *L2* sessions' use. Since for every type *L1* session request there is $\alpha^{-1}-1$ type *L2* sessions, the blocking probability for type *L2* sessions decreases from $Pb_2(fair)$ by $\varepsilon = (\delta L_1/L_2)/(\alpha^{-1}-1)$, that is

$Pb_2(unfair) = Pb_2(fair) - \varepsilon$. From Equation (4.3.1.3), $Pb(unfair) = \{\alpha L_1 Pb_1(unfair) + (1-\alpha)L_2 Pb_2(unfair)\} / \{\alpha L_1+(1-\alpha)L_2\} = Pb(fair) = Pb$.

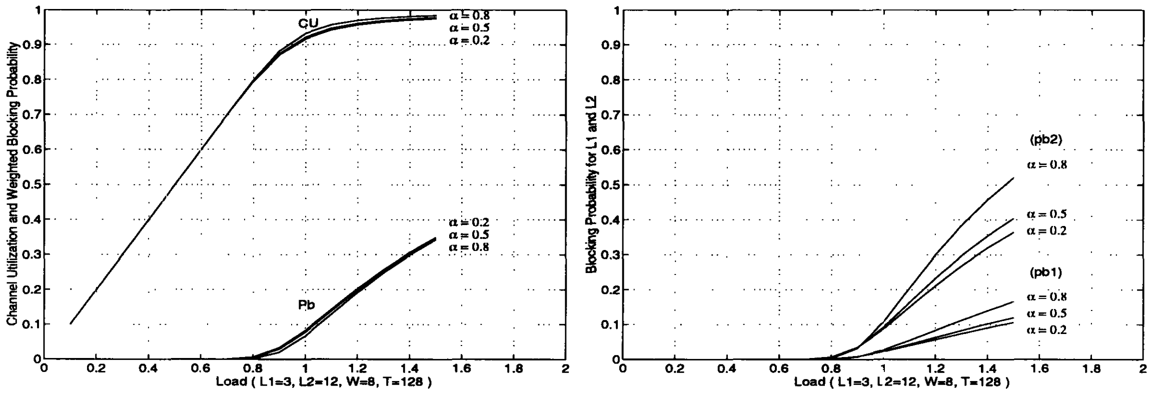


Figure 4.3.1.2 Mathematical Analysis for *M/M/2/B/** System

In Figure 4.3.1.2, we showed the channel utilization, weighted blocking probability, and blocking probability for type L_1 and L_2 sessions for a system of L_1=3, L_1=12, W=8, T=128. Notice that *CU* and *Pb* vary little as a function of α. The program used is included in Appendix A.3.1.

OPtimized Network Engineering Tools (OPNET) is a comprehensive software program capable of simulating large communications networks with detailed protocol modeling and performance analysis[6]. Its features include: graphical specification of models; a dynamic, event-scheduled Simulation Kernel; integrated data analysis tools; and hierarchical, object-based modeling. We will use OPNET v2.3/M(c) on SUN SPARC stations to model our systems.

The highest "domain" in OPNET's hierarchical modeling structure is the network model. Nodes and links can be placed directly in a network or within subnetworks, which can be treated as single objects in the network model. A node is defined by connecting various module types with packet streams and statistics wires. The connection between modules allow for guided packet and status information exchange between modules. Each module placed in a node serves a unique purpose, such as generating packets, queueing packets, processing packets, or transmitting and receiving packets. At the core of most OPNET simulations are user-defined process models in addition to that provided by OPNET. Process models can represent the logic of communications hardware, network protocol, distributed algorithms, or high-level server-client processes. OPNET allows construction of process models by graphical representation of extended finite state machine.

In our system, since we ignore propagation delay of the packet, we can collapse physical nodes (terminals) into one logical node. So we've designed our network consisting of only one node, shown as in Figure 5.0.1. This node is made up of three modules, a source, the *scheduler* and the *release* process modules. The source, *src*, uses the ideal source generator provided by OPNET, it generates session requests according to a Poisson process with a specified rate. The *scheduler* model processes a session request by making appropriate resource assignments. It differs slightly for the blocking system and the queueing system. The *release* module erases the session after it's finished, it is also different for blocking and queueing system.

In Appendix B.1, we presented the network model used in OPNET provided report form, and in Appendix B.2 is the OPNET report form of the node model used.

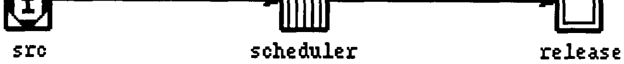


Figure 5.0.1 Node Module

5.1 Blocking System

The *release* module is shown in Figure 5.1.1. Upon entering the module, it erases the packet that represents session request and clear the associate variables(i.e. release the system resource used by the session).

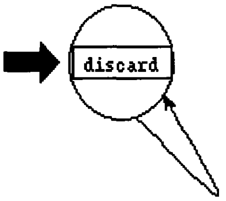


Figure 5.1.1 *Release* Process Module

The *scheduler* module for blocking system is shown in Figure 5.1.2. At the beginning of the simulation, the system will first enter the *init* state, where variables are initialized. Environmental variables such as the number of wavelength channels in the system *W*, the frame size *T*, number of nodes in the system *N*, the session throughput requirement in terms of number of slots per frame *L* are registered. At the end of the *init* state, if there is a session request arrival, it transits into the *pk_prepare* state, else it goes into the *idle* state. In the *pk_prepare* state, the session request gets assigned the source and destination address and the session holding time before getting sent into the *schedule* state. Once in the *schedule* state, based on the source and destination address and the available system resource, the session request is either rejected or accepted. In the former case, the session request is registered as "blocked" and deleted. In the later case, the scheduler send a delayed interrupt to the *release* module. So after a delay period equal to the session hold time, the release module will be notified and delete the expired session. Once the scheduler has finished with a session request, it goes back to the *idle* state, where it waits for the next arrival. When the simulation ends, the function *record_stats* records system statistics such as channel utilization, blocking probability before exiting.

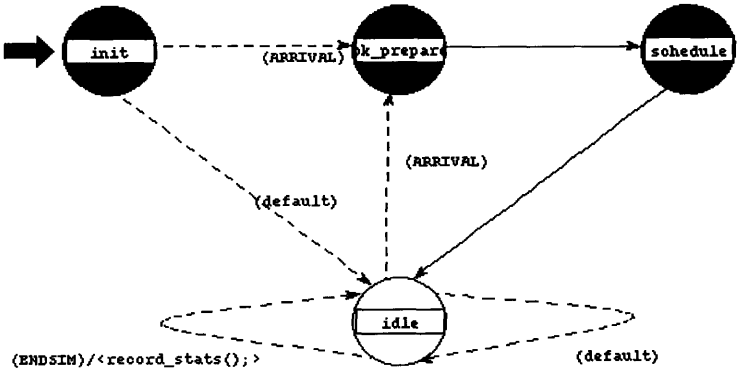


Figure 5.1.2 *Scheduler* Process Module for Blocking System

5.2 Queueing System

The queueing system behaves very much like the blocking system, the only difference is that it keeps a queue of session requests that cannot be immediately satisfied.

The *release* process model for the queueing system is almost the same as that shown in Figure 5.1.1. In addition to erasing the expired session and freeing associated system resource, it also sends an interrupt to the *scheduler* module to notify it of the additional resource available. Upon receiving such an interrupt, if the *scheduler* finds itself with an non-empty queue, it will go into state *schedule,* and process the session requests hold in the queue.

The *scheduler* process model for the queueing system is shown in Figure 5.2.1. As in the blocking system, the module will enter from the *idle* state into the *pk_prepare* state when a new session request comes in, and proceed to the *schedule* state to process the request. Additionally when an interrupt from the *release* module comes in signaling additional resource available, and the queue is not empty at the time, the module enters from the *idle* state to the *schedule* state to process session requests hold in the queue. Everything else is identical to that in the blocking system.

There is a slight difference between single class and two class systems. In single class system a single sub-queue is maintained. While in two class system two sub-queues are maintained, one for each type of the sessions. Upon receiving the interrupt from the *release* module signaling the release of a session, the system will try to fill the "vacancy" with the same type of session from the sub-queue as the one just released. The different type of session in sub-queue has lower priority to be filled.

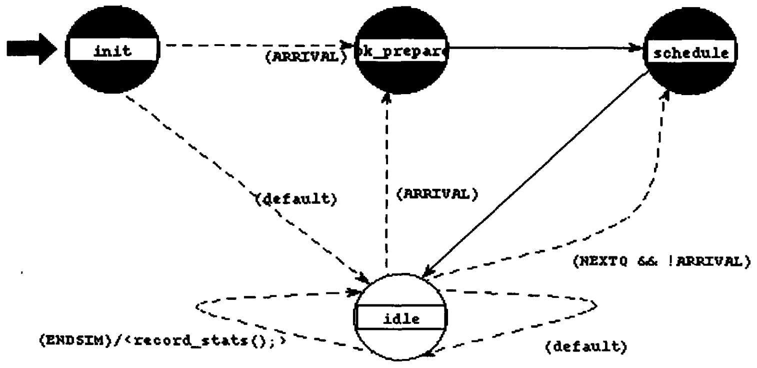


Figure 5.2.1 *Scheduler* Process Module for Queueing System

5.3 Scheduling Algorithms

In this section we describe the three scheduling algorithms used. And point out the differences between the single class and the two class systems.

5.3.1 Single Class System

<u>Contiguous *L+1 Assignment Algorithm*</u>

Since all sessions require L slots for data and one additional slot for overhead tuning, we can regard the system as consisting of W rows and $\lfloor T/(L+1) \rfloor$ columns of cells. Each cell consists of contiguous $L+1$ slots. The scheduling algorithm searches through each column and row to find a free cell without *Column Conflict* and assign it to the session. The following is the general description of the algorithm.

```
♦ for each column i ($0 \le i < \lfloor T / (L+1) \rfloor$) do
    • determine if there is a Column Conflict and find a free cell in the column.
    • if without Column Conflict and a free cell is found, mark cell found, end the search;
      else, continue to next column.

♦ if cell found at the end of the search, assign the cell to the session;
  else, reject the session request and mark it as blocked.
```

<u>Contiguous *L Assignment Algorithm*</u>

Similar to the *L+1 Assignment*, this system can be regarded as consisting of W rows and $\lfloor T/L \rfloor$ columns of cells. Each cell consists of contiguous L slots. However a free cell is not necessarily usable due to *Column, Pre-column* and *Post-column Conflict* as discussed in Section 3.2. From now on we call a *free cell* without *Column Conflict* an *available cell.* An available cell encountering *Pre-column* or *Post-column Conflict* and unable to resolve it cannot be used either, and we call it a rejected cell. An non-rejected cell is a *usable cell* ready for assignment to the session. We will assign priority to each usable cell depending on the number of conflicts it resolved. For example a usable cell that encounters none of the *Pre-column* and *Post-column Conflict* has zero priority. A usable cell that encounters and resolves a *Pre-column Conflict* due to a transmitter has priority one. A usable cell that encounters and resolves all of the *Pre-column Conflicts* due to a transmitter and a receiver, and *Post-column Conflicts* due to a transmitter and a receiver has priority four. The algorithm will search through all cells in the system and replace the *keeper cell* with higher priority usable cell. This priority system favors maximum conflicts resolution, and hopefully helps us lowering session blocking due to unresolved conflicts. The search will end when the keeper cell reaches the highest priority of four or when all cells are searched. At the end of the search, the session will be assigned to the keeper cell if it exists, or marked as rejected. The following is the general description of the algorithm.

```
♦ for each column i ($0 \le i < \lfloor T/L \rfloor$) do
    • determine if there is a Column Conflict and find all free cells in the column.
    • if there is Column Conflict or no free cell is found, continue to next column.
    • for all available cells in the column, determine their usability by checking Pre-column
      and Post-column Conflicts and assign priority to usable ones, replace the keeper
      cell with higher priority usable cell.
    • if the keeper cell's priority reaches four (highest priority), end the search;
      else, continue to next column.

♦ if keeper cell exists at the end of the search, assign it to the session;
  else, reject the session request and mark it as blocked.
```

Random L Assignment Algorithm

This algorithm allows for MW-RS resource allocation as discussed in Section 3.6. Given a request of *L* slots per sessions, the final assignment may consist of slots spread across all wavelength channels. The algorithm described below is not limited to single or two class traffic. It works for any value of *L*.

A block (l,w,s,o) is made of slots with identical wavelength channel *w* and consecutive slot numbers starting at position *s* in the frame. The *size* of a block *l* is the number of slots contained in it. The overhead indicator of a block *o* can be represented by a binary number *xy*, where *x* and *y* indicate if the block encounters *Pre-column* and *Post-column Conflicts* respectively. It takes on the value of one for unresolved conflict and zero otherwise. Notice that *o* with decimal value 2 represents binary 10. The actual overhead of the block has the decimal value of $x+y$ which represents the number of slots needed for tuning. So a block with *o* of decimal 2 needs 1 slot for overhead due to Pre-column Conflict. The *available size* of a block is the size of the block (*l*) minus the actual overhead of the block. The available slots of a block are all slots in the block except the first one if *x* is one and the last one if *y* is one.

What the algorithm does first is to find all slots that could be used for the session and sort them into a list of blocks in order of decreasing size. From this list of blocks, the algorithm will allocate slots to the session using criteria of minimizing waste and using the smallest block possible. The concept of waste is as following. The number of slots provided by a block is the lesser of the available size of the block and the number of slots needed. When the size of a block is greater than the number of slots provided by it, the difference between the two is called the waste associated with the block. For example, three slots are needed while the block has size seven. So the number of slots provided by the block is three while the waste is four. The occurrence of waste indicates fragmentation of the block, which we want to avoid. Another way of reducing fragmentation is to reserve larger blocks for sessions with larger *L*, which means using the smallest block possible when making assignment.

Since we are using MW-RS approach, multiple blocks can be assigned to a session. There are also possibly many ways to assign blocks to a session. What we are going to do is to work down the list of blocks until we find a "possible assignment" (pa) which may contain one or more blocks. We assign priority to the pa, and may replace "keeper assignment" (ka) with the pa if the pa has higher priority. We will continue this process until the end of the list, and assign ka to the session if one exists. The criteria for updating pa is to compare successive pa's waste, update only when new pa has less waste. Since a pa may contain more than one block, the waste of a pa is the cumulated waste of all blocks in it.

The process of finding a pa is as following. As we moved down the list, examine the block against the ones already assigned to the pa to determine if any slots have identical position in the frame. If that is the case, discard the block due to *Column Conflict* at that position. We call this *"usability test"*. Next step is to determine if the block is bigger than needed. If the available size of the block is smaller than the number of slots needed, we add the block to the pa and update the number of slots needed. Repeat the process as we move down the list until enough blocks are assigned to the pa to provide a total of L slots. However if the available size of the block is bigger than the number of slots needed, we mark this block as "temporary assignment" (ta) since we don't want to break up a big block. We move to the next block which has a smaller size and so on until we get to the smallest block possible and assign it to the pa. The criteria for updating ta is to compare successive ta's waste and overhead, update only when the newer one has less waste and no bigger overhead. We call this *"ta-update process"*. The following is the general description of the algorithm.

- ♦ step 1:
  - find all free slots without *Column Conflict* and express them in terms of blocks.
  - discard all blocks whose available size is less than one.
  - sort remaining blocks into a list first by decreasing size and second by increasing overhead.

- ♦ step 2:
  - fetch the next available block in the list, repeat step 2 if failing "usability test".
  - if the block is a ta, repeat step 2 until completion of "ta-update process";
    else, assign the block to pa.
  - if the pa is complete, replace ka with it if it has less waste;
    else, repeat step 2.

- ♦ step 3:
  - if ka exists when reaching end of the list, assign these slots to the session by rule 1;
    else, reject the session request and mark it as blocked.

- ♦ rule 1:
  - for all blocks in the ka except the last one, assign their available slots to the session.
  - the number of slots needed (ns) is L minus the number of slots provided thus far.
  - for the last block in the ka, we only need ns slots starting from position 1 if overhead of the block is 0; position 2 if overhead is 1 and 3; position "second from the last" and counting backwards if overhead is 2.

We will use an example to illustrate step 2. Assume we have following list of blocks: $\{(10,1,1,0)\,(8,2,8,0)\,(7,4,13,0)\,(6,3,13,0)\,(6,1,20,1)\,(6,2,27,2)\,(5,1,40,0)\,(3,4,1,0)\}$ and we need L equals to 16 slots.

1) fetch block 1 (10,1,1,0), pass "usability test".
2) block 1 is a non-ta since available size 10 is less than needed 16 slots, assign block 1 to pa 1, waste 0, number of slots needed is now 6.
3) fetch block 2 (8,2,8,1), fail "usability test" since *Column Conflict* at position 8, 9 and 10.
4) fetch block 3 (7,4,13,0), pass "usability test", make it ta since available size 7 is greater than needed 6, waste 1.
5) fetch block 4 (6,3,13,0), pass "usability test", waste 0, make it the new ta and the end of "ta-update process", assign block 4 to pa1, number of slots needed is now 0.
6) pa1 complete with block 1 and 4 with a total waste of 0, assign pa1 to ka.
7) fetch block 5 (6,1,20,1), pass "usability test".
8) block 5 is non-ta, waste 1, assign to pa2, number of slots needed 11.
9) fetch block 6 (6,2,27,2), pass "usability test".
10) block 6 is non-ta, waste 1, assign to pa2, number of slots needed 6.
11) fetch block 7 (5,1,40,0), pass "usability test".
12) block 7 is non-ta, waste 0, assign to pa2, number of slots needed 1.
13) fetch block 8 (3,4,1,0), pass "usability test".
14) block 8 is ta, waste 2.
15) end of list, assign block 8 to pa 2.
16) pa 2 complete with block 5, 6, 7 and 8 with a total waste of 4, ka=pa1.
17) assign available slots in block 1 and 4 to the session.

5.3.2 Two Class System

Contiguous *L+1(L) Assignment Algorithm*

To ensure fair distribution of system resource among type L_1 and L_2 sessions, we partition the frame of size T into two independent portions of size T_1 and T_2 serving type L_1 and L_2 sessions respectively. The system is in essence two independent single class uniform traffic subsystems. We will use *Contiguous L+1(L) Assignment Algorithm* described in previous section for each of the two subsystems.

Let α be the percentage of the total sessions that is type L_1 sessions. A system of W wavelength channels and size T_1 frame can support on average $WT_1/(L_1+b)$ number of type L_1 sessions. A system of size T_2 frame can support on average $WT_2/(L_2+b)$ number of type L_2 sessions. In both cases b takes on the value of one for *L+1 Assignment* and zero for *L Assignment*. We have,

$$\{WT_1/(L_1+b)\}/\{WT_2/(L_2+b)\} = \alpha/(1-\alpha) \qquad (5.3.2.1)$$

which gives

$$T_2/(L_2+b) = T/\{(L_2+b)+(L_1+b)\alpha/(1-\alpha)\} \qquad (5.3.2.2)$$

The subsystem for L_2 sessions can be regarded as c_2 columns of cells, where c_2 equals to $\lfloor T_2/(L_2+b) \rfloor$ as specified in Equation (5.3.2.2). The subsystem for L_1 sessions can be regarded as c_1 columns of cells, where c_1 equals to $\lfloor (T-(L_2+b)c_2)/(L_2+b) \rfloor$. The following is the general description of the algorithm.

- determine the number of cells c_1, c_2 for subsystem 1 and 2 respectively.
- if type L_1 sessions, using *Contiguous L+1(L) Assignment Algorithm* (Section 5.3.1) to schedule the session in subsystem 1.
- if type L_2 sessions, using *Contiguous L+1(L) Assignment Algorithm* (Section 5.3.1) to schedule the session in subsystem 2.

Random *L Assignment Algorithm*

The *Contiguous L+1(L) Assignment Algorithms* outlined above are fair for both type L_1 and L_2 sessions, however an accurate estimate of the parameter α is needed. By using *Random L Assignment Algorithm* no estimation of α is needed. For blocking system, the *Random L Assignment Algorithm* outlined in the previous section can be used exactly as it is. For queueing system, notice that two separate queues for L_1 and L_2 sessions will be kept. So when a session request gets queued, it gets sent to the queue for that particular session type. Everything else is the same as described in the previous section.

Chapter 6 Simulation Results

In this chapter we present the simulation results for single/two class uniform traffic systems and single class client/server systems, and compare them to mathematical approximations/bounds derived in Chapter 4.

6.1 Single Class, Uniform Traffic System

6.1.1 Blocking System

The simulation results for *M/M/1/B/L+1, M/M/1/B/CL and M/M/1/RL* systems are shown in Figure 6.1.1.1 through Figure 6.1.1.3. The simulations are run for eight and forty users; two, four and eight wavelength channels; and throughput requirement of one, three, and twelve slots per frame. The solid lines in the graphs are calculated results based on the mathematical approximations/bounds obtained in Section 4.2.1. The dotted lines are simulation results.

As far as achieving maximum channel utilization is concerned, two factors come in to play, the number of wavelength channels W and the ratio of the number of nodes to the number of wavelength channels represented by parameter γ.

When γ is much greater than one, adding more wavelength channels to the system yields more efficiency as in the case of the forty user & eight wavelength system, the forty user & four wavelength system, and the forty user & two wavelength system. The reason is that when γ is large, session rejection due to transmitter and receiver conflicts is negligible. Therefore adding more wavelength channels translates directly into more system resources for the users.

When is γ close to one, systems with higher γ yields higher efficiency as in the case of the eight user & four wavelength system ($\gamma = 2$), and the eight user & eight wavelength system ($\gamma = 1$). The reason is that smaller γ indicates more session rejection due to transmitter or receiver conflicts and therefore less channel utilization rate. When γ is less than one, the transmitter and receiver become the bottleneck of the system, adding more wavelength channels will not give more resources to each of the users.

We can also see that systems with γ larger than two have relatively the same channel utilization rate regardless the number of wavelength channels. Only when γ is significantly small (less than two) does the system have a much lower channel utilization rate.

The figures also showed that systems with larger L reach the maximum channel utilization more slowly (i.e. has a larger 'knee').

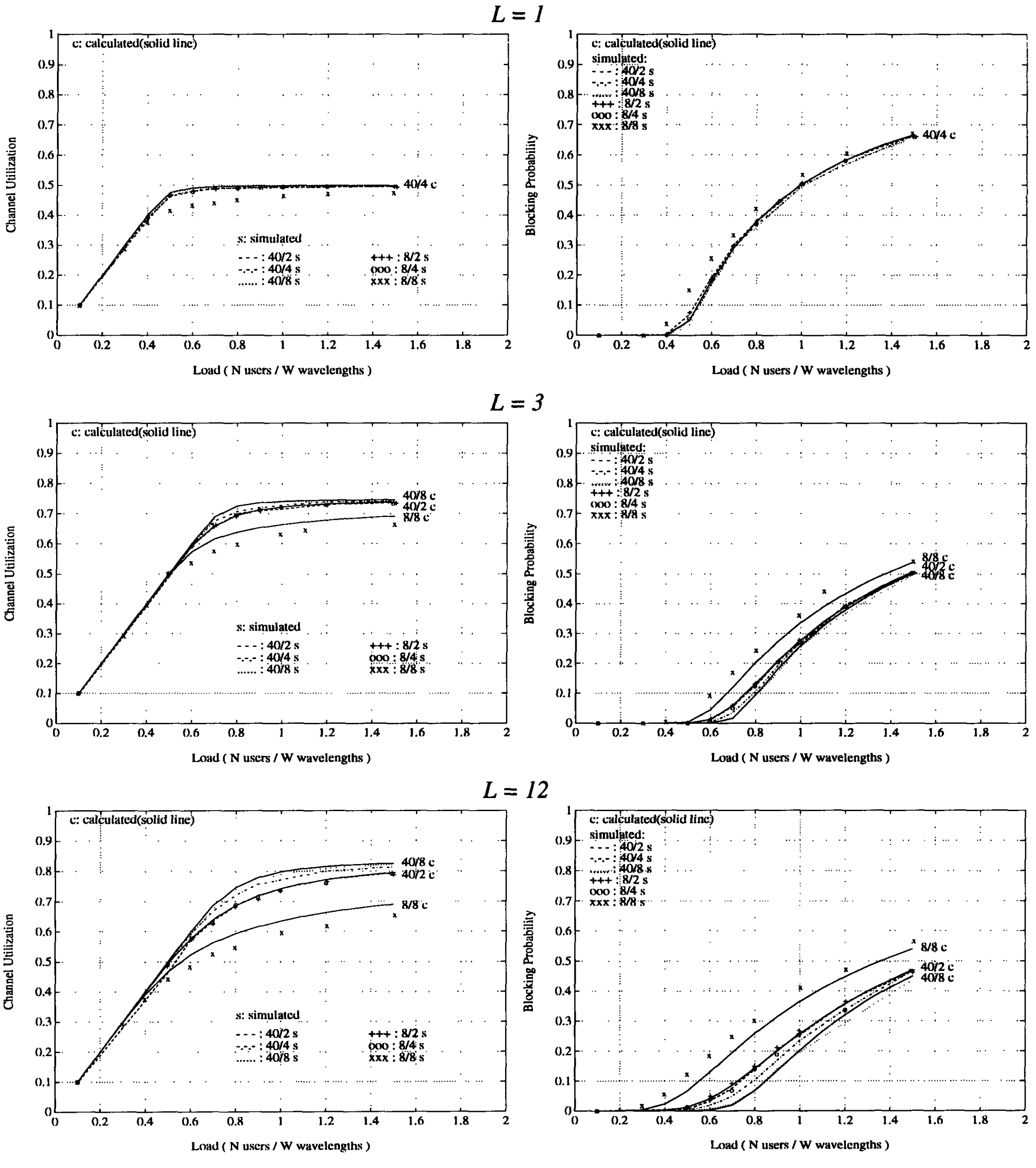


Figure 6.1.1.1 *M/M/1/B/L+1* System Results

For *M/M/1/B/L+1* systems, simulation results approach the derived mathematical approximations rather closely. For systems with relatively large number of users ($\gamma > 1$), the channel utilization approaches that of the absolute maximum derived in Section 4.2.1 (0.5, 0.75 and 0.84 for L equals to 1, 3 and 12 respectively).

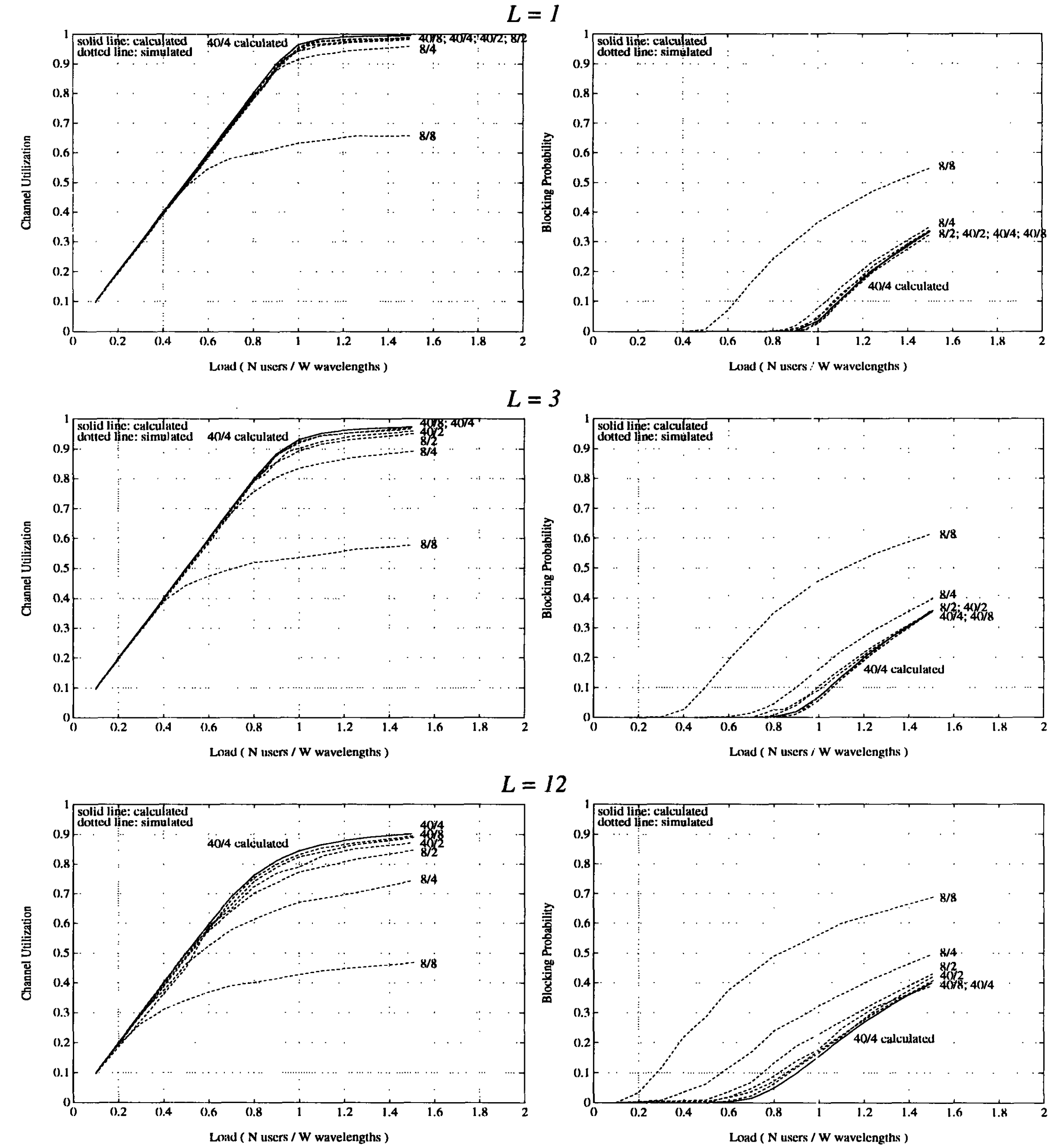
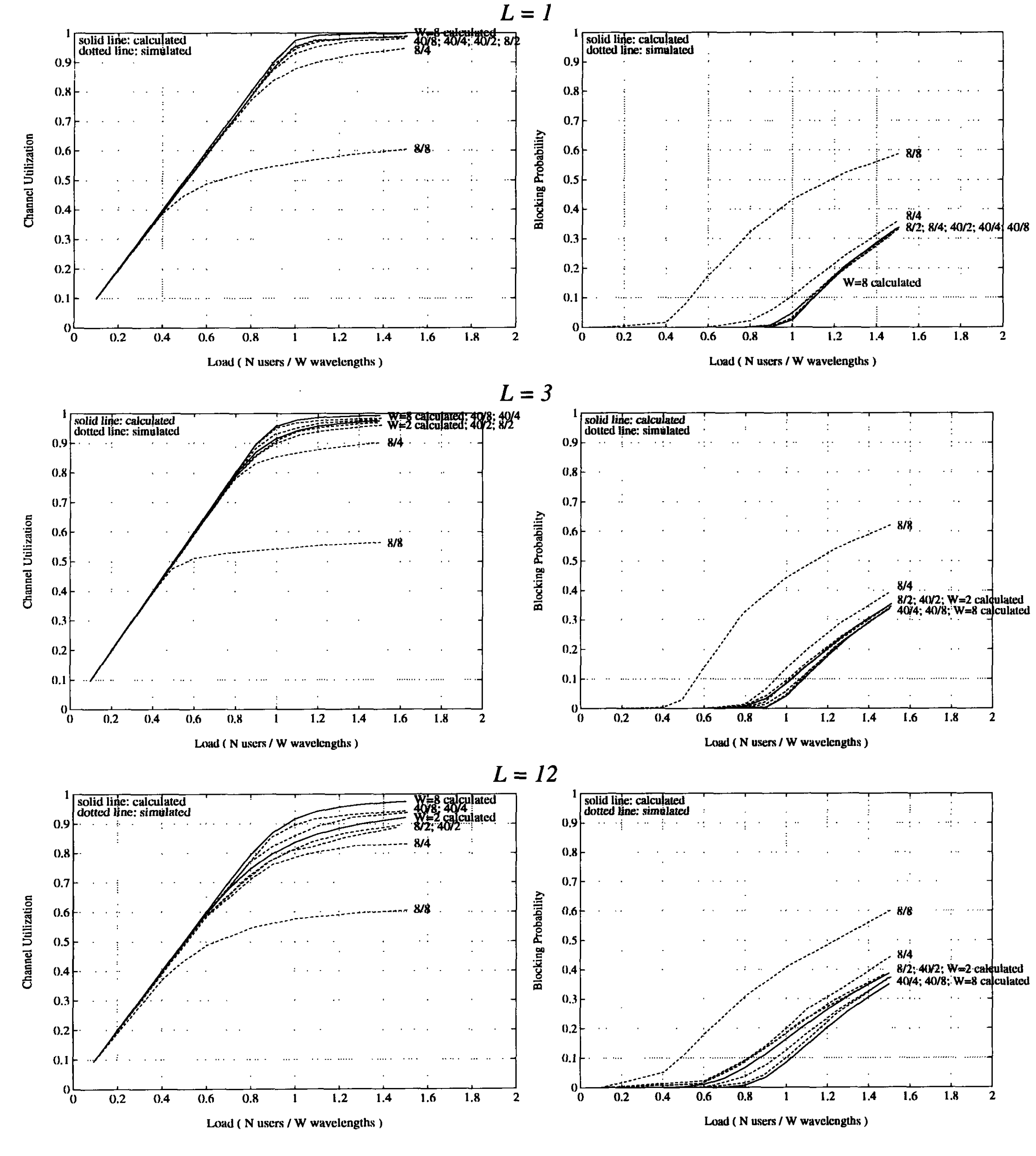


Figure 6.1.1.2 *M/M/1/B/CL* System Results

For *M/M/1/B/CL* system, the same two factors (number of wavelength channels *W* and the ratio γ of the number of nodes to wavelength channels) influence the system performance. It's clear that when the number of users is relatively large (γ > 1), the channel utilization approaches the mathematical bounds as well as the absolute maximum derived in Section 4.2.1 (1.0, 0.98 and 0.94 for *L* equals to 1, 3 and 12 respectively). Notice that the systems with *L* of 12 approaches the maximum much more slowly.



L = 1

L = 3

L = 12

Figure 6.1.1.3 *M/M/1/B/RL* System Results

For *M/M/1/B/RL* system, the same two factors (number of wavelength channels *W* and the ratio γ of the number of nodes to wavelength channels) influence the system performance. It's clear that when the number of users is relatively large ($\gamma > 1$), the channel utilization approaches the mathematical bounds as well as the absolute maximum derived in Section 4.2.1 (1.0, 0.999 and 0.996 for *L* equals to 1, 3 and 12 respectively). Notice that the systems with *L* of 12 approaches the maximum much more slowly.

We want to compare the efficiency of the three algorithms for systems represented by large and small γ.

For systems with relatively large number of nodes (e.g. $N = 40,\ W = 8 : \gamma = 5$), session rejection due to transmitter or receiver conflicts is relatively rare. Therefore *Contiguous L+1 Assignment Algorithm* performs the worst due to the added tuning overhead of one slot. The smaller the *L*, the more of the overhead there is, and the lower the channel utilization it achieves. *Random L Assignment Algorithm* performs the best while *Contiguous L Assignment Algorithm* trails behind. This is expected since *Contiguous L Assignment Algorithm* wastes slots at the end of the frame when *L* doesn't divide *T* evenly.

For systems with relatively small number of nodes (e.g. $N = 8,\ W = 8 : \gamma = 1$), *Contiguous L+1 Assignment Algorithm* performs much better than the other two algorithms especially for systems with large number of *L*. The reason is two fold. First for *Contiguous L+1 Assignment Algorithm* the percentage of tuning overhead is *1/L*, smaller for larger *L*. Which means higher channel utilization for larger *L*. Secondly for the *Contiguous* and *Random L Assignment Algorithms*, whenever a group of *L* free slots are rejected due to *Column, Pre-column* or *Post-column Conflicts*, these *L* slots are "wasted". The larger the *L*, the higher the chance of these slots being rejected, and also more slots wasted when rejected. Which again means worse channel utilization for the *Contiguous* and *Random L Assignment Algorithms* for systems with larger *L*. From the figure, it is clear that *Contiguous L+1 Assignment Algorithm* performs the best for *L* equals to three and twelve, and the worst for *L* equals to one where the tuning overhead is 100%. When comparing *Contiguous* and *Random L Assignment Algorithms*, notice that for the system shown *Contiguous L Assignment Algorithm* performs better than *Random L Assignment Algorithm* for smaller *L* ($L = 1$ & 3) while the opposite is true for larger *L* ($L = 12$). Again this can be attributed to the trade-offs between two factors. First, the random algorithm fragments the system resources, which leads to more rejections due to transmitter and receiver conflicts and therefore lower channel utilization. On the other hand, it is harder to find *L* contiguous slots than *L* random slots. The larger the *L*, the harder it is to find *L contiguous* slots even if the system has the resources available. Which means lower channel utilization for the contiguous algorithms. When *L* is large enough, the second effect dominates and *Random L Assignment Algorithm* outperforms *Contiguous L Assignment Algorithm* ($L=12$).

When comparing the channel utilization (or blocking probability) achieved by systems with small γ (e.g. $N = 8,\ W = 8 : \gamma = 1$) to that with large γ (e.g. $N = 40,\ W = 8 : \gamma = 5$), we see that systems using *Contiguous L+1 Assignment Algorithm* has the smallest difference while those using *Contiguous* or *Random L Assignment Algorithms* have similarly larger differences. The reason is that by adding one slot for tuning overhead, *Contiguous L+1 Assignment Algorithm* helps systems with small γ while degrades systems with larger γ, therefore narrows the difference between the two.

Hopefully the load of the system is proportional to the number of users in the system. So when the users are few, the load is low and the system operates at below the knee of the curve and we don't have to worry about achieving maximum channel utilization. Only when more users come into the system, the load increases and we want to operate at the higher "plateau" of the curve. In this situation we see that *Random L Assignment Algorithm* gives the best performance.

The OPNET reports of the process models used for *M/M/1/B/L+1, M/M/1/B/CL, M/M/1/B/RL* are presented in Appendix B.3.1. The termination method and seeds selection are also discussed.

6.1.2 Queueing System

The results of the simulations are presented in Figure 6.1.2.1 through Figure 6.1.2.3. The solid lines are mathematical bounds based on results in Section 4.2.2. The dotted lines are simulation results. Notice that the simulation results are very close to the calculated bounds for all three systems when the number of nodes in the system is relatively large ($\gamma>1$). The reason is the same as that discussed in the previous section for the blocking system.

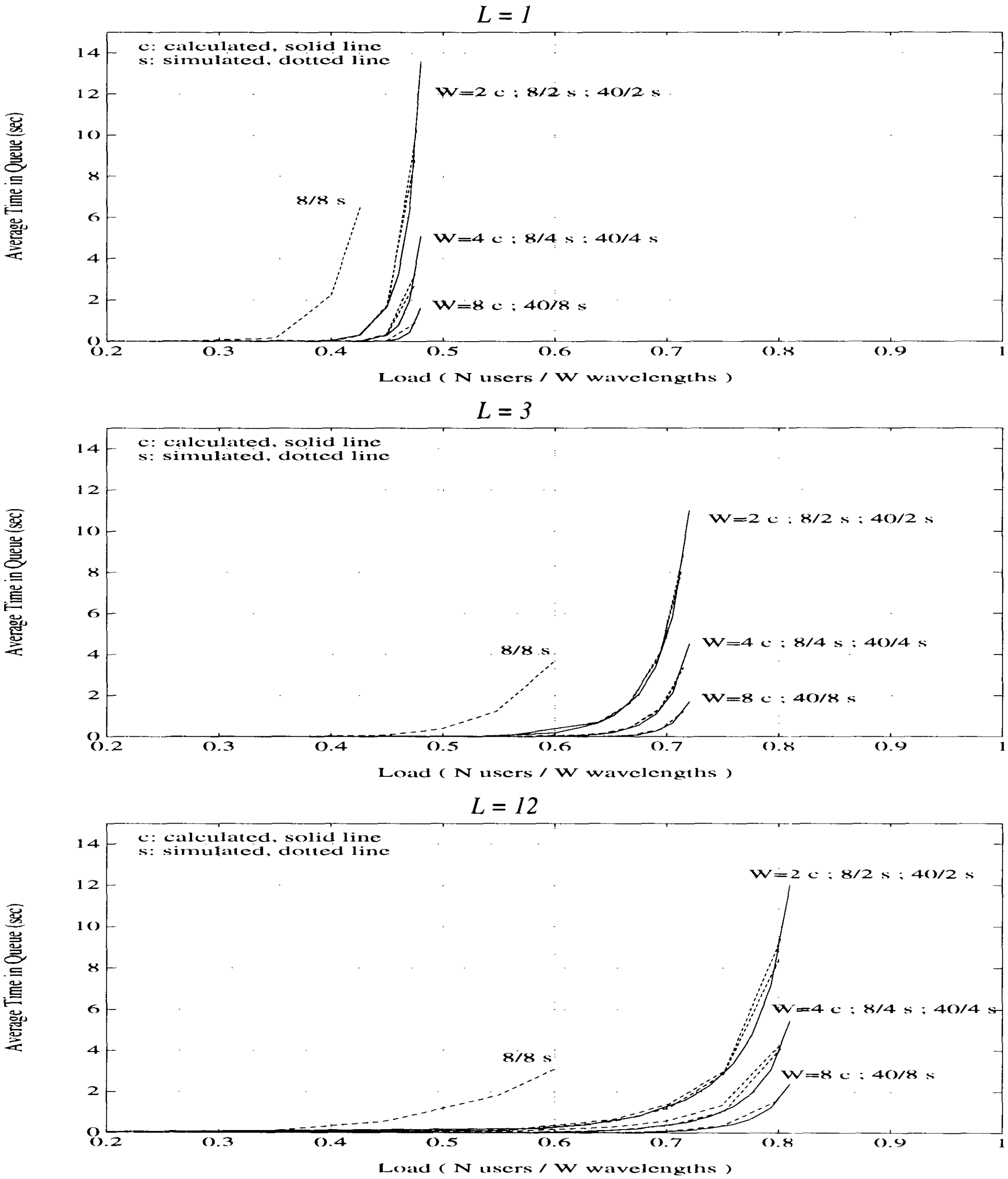


Figure 6.1.2.1 *M/M/1/Q/L+1* System Results

For *M/M/1/Q/L+1* system, when the number of users is relatively large ($\gamma > 1$), the channel utilization approaches the mathematical bounds as well as the absolute maximum derived in Section 4.2.2 (0.5, 0.75 and 0.84 for *L* equals to 1, 3 and 12 respectively). In case of γ equals to one (*N=8 & W=8*), the "knee of the curve" occurs at load around 0.4, 0.55 and 0.5 for *L* equals to 1, 3 and 12 respectively due to heavy transmitter and receiver conflicts. Refer to Figure 6.1.1.1, we see that *M/M/1/B/L+1* systems start to reach channel utilization "plateau" (or non-zero blocking probability) at the same load values. Also systems with larger *L* have slower rising curves. So the results are consistent.

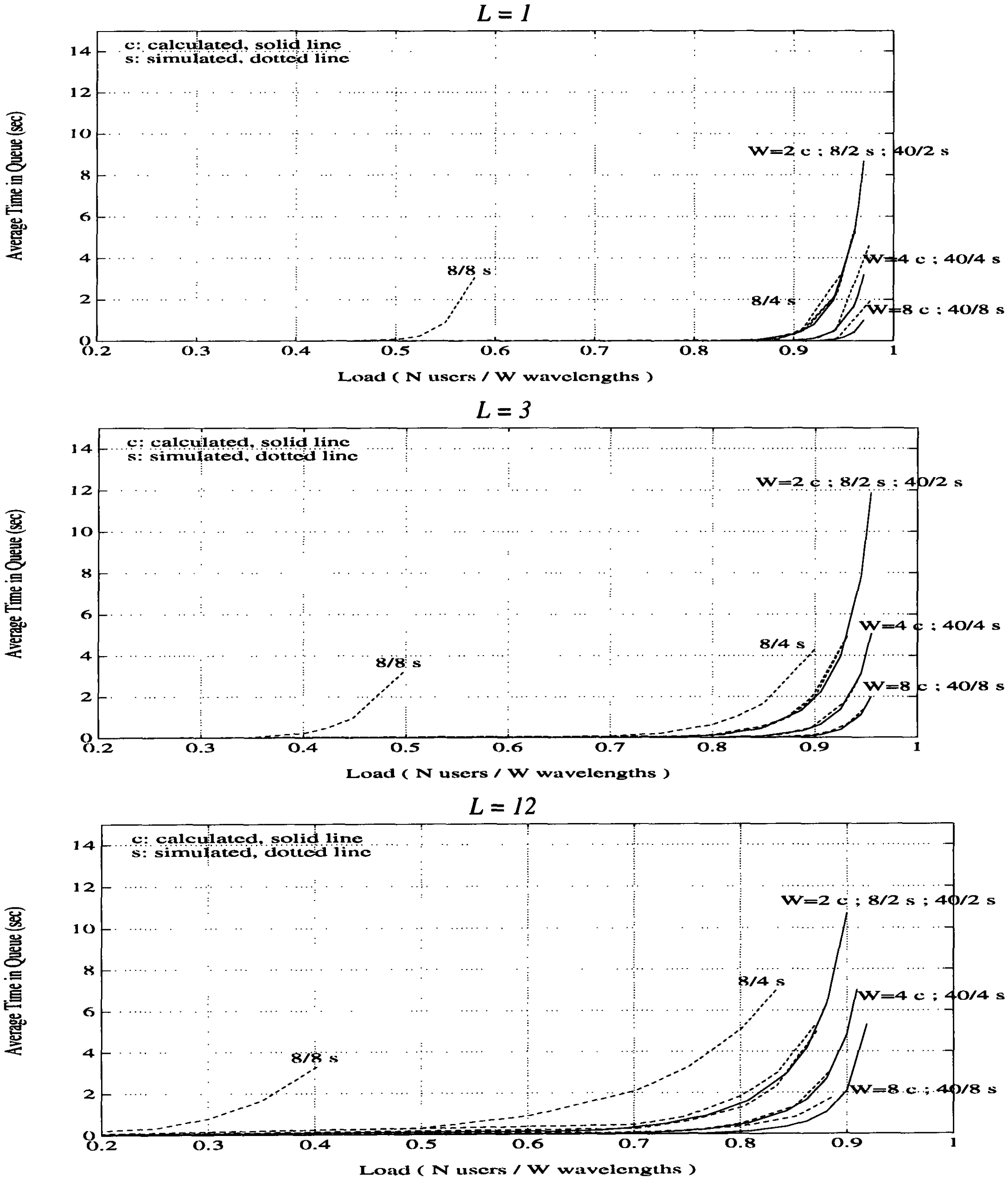


Figure 6.1.2.2 *M/M/1/Q/CL* System Results

For *M/M/1/Q/CL* system, when the number of users is relatively large ($\gamma > 1$), the channel utilization approaches the mathematical bounds as well as the absolute maximum derived in Section 4.2.2 (1.0, 0.98 and 0.94 for L equals to 1, 3 and 12 respectively). In case of γ equals to one ($N=8$ & $W=8$), the "knee of the curve" occurs at load around 0.55, 0.45 and 0.35 for L equals to 1, 3 and 12 respectively due to heavy transmitter and receiver conflicts. Refer to Figure 6.1.1.1, we see that *M/M/1/B/L+1* systems start to reach channel utilization "plateau" (or non-zero blocking probability) at the same load values. Also systems with larger L have slower rising curves. So the results are consistent.

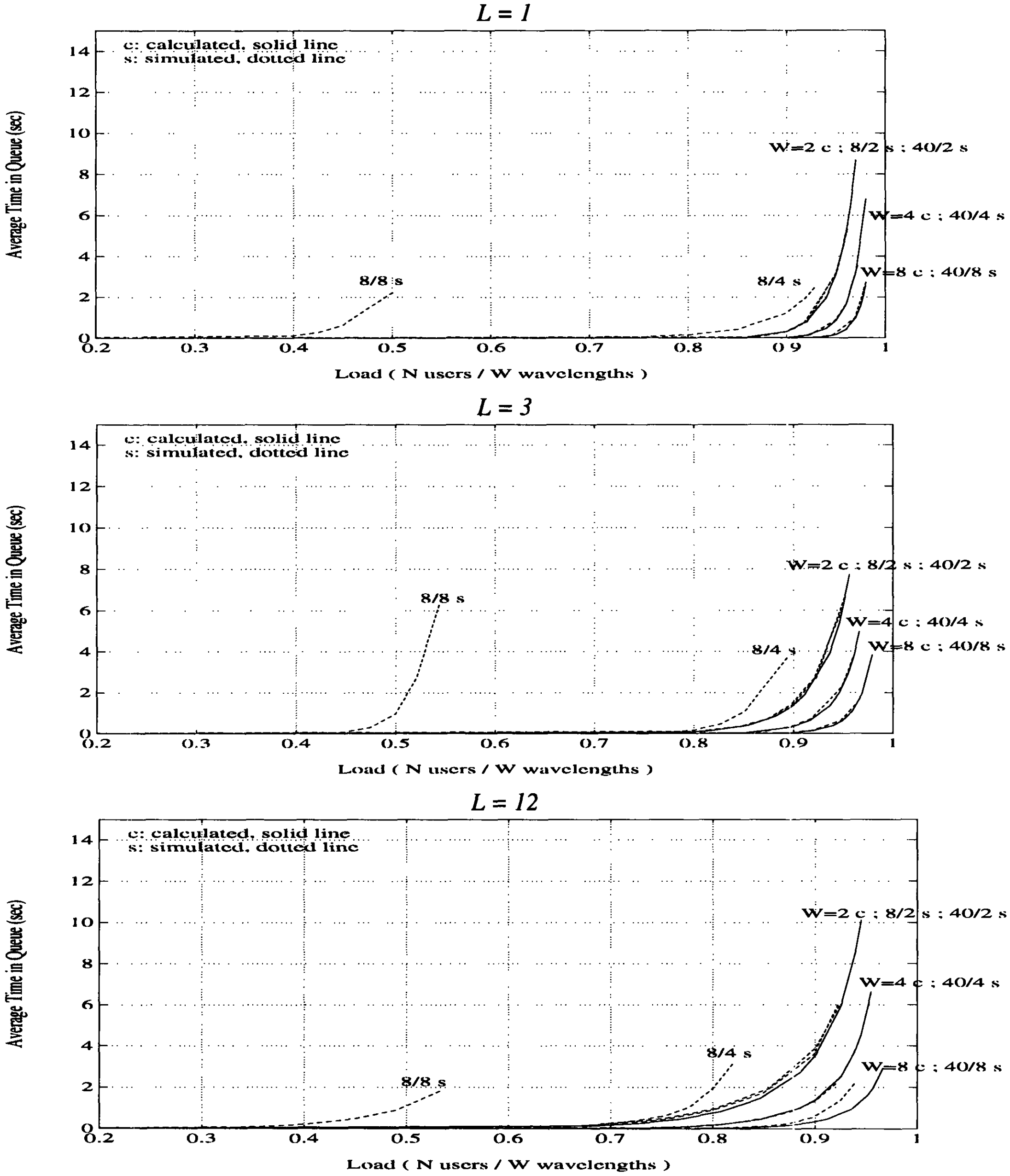


Figure 6.1.2.3 *M/M/1/Q/RL* System Results

For *M/M/1/Q/RL* system, when the number of users is relatively large ($\gamma > 1$), the channel utilization approaches the mathematical bounds as well as the absolute maximum derived in Section 4.2.2 (1.0, 0.999 and 0.996 for *L* equals to 1, 3 and 12 respectively). In case of γ equals to one (*N=8 & W=8*), the "knee of the curve" occurs at load around 0.45, 0.5 and 0.5 for *L* equals to 1, 3 and 12 respectively due to heavy transmitter and receiver conflicts. Refer to Figure 6.1.1.1, we see that *M/M/1/B/L+1* systems start to reach channel utilization "plateau" (or non-zero blocking probability) at the same load values. Also systems with larger *L* have slower rising curves. So the results are consistent.

Again we want to compare the efficiency of the three scheduling algorithms for systems with large number of users (e.g. $N = 40,\ W = 8 : \gamma = 5$) and small number of users (e.g. $N = 8,\ W = 8 : \gamma = 1$). As in the single class uniform traffic blocking systems *Random L Assignment Algorithm* for the queueing systems also gives the best performance for systems with $\gamma > 1$. For systems with smaller γ *Contiguous L+1 Assignment Algorithm* seems to give the best performance when L is large for the same reason as discussed in the previous section. When comparing the performance (e.g. the load value at the "knee of the curve") of systems with small γ (e.g. $N = 8,\ W = 8 : \gamma = 1$) to that with large γ (e.g. $N = 40,\ W = 8 : \gamma = 5$), we see that systems using *Contiguous L+1 Assignment Algorithm* has the smallest difference while those using *Contiguous* or *Random L Assignment Algorithms* have similarly larger differences. The reason is again the same as that for the blocking system. We can therefore conclude that M/M/1/B/* and M/M/1/Q/* systems have similar performance characteristics. In both systems *Random L Assignment Algorithm* gives the best performance when $\gamma > 1$ while *Contiguous L+1 Assignment Algorithm* seems to give the best performance when γ is small and L is large.

In Appendix B.3.2 are the OPNET reports of the process models used for *M/M/1/Q/C+1, M/M/1/Q/CL, M/M/1/Q/RL.* The termination criteria and seeds selection are also discussed.

6.2 Two Class, Uniform Traffic System

6.2.1 Blocking System

The simulation results for *M/M2/B/L+1, M/M/2/B/CL* and *M/M/2/B/RL* systems are shown in Figure 6.2.1.1 and Figure 6.2.1.2. The simulations are run for forty users, eight wavelength channels, and throughput requirement of three (L_1) and twelve (L_2) slots per frame. The parameter α represents the percentage of session arrivals that is type *L1*. Given α, the percentage of system resources used by type L_1 sessions is β $\{\beta = \alpha L_1/(\alpha L_1 + (1-\alpha)L_2)\}$.

In Figure 6.2.1.1, channel utilization and weighted blocking probability are plotted. The solid lines are mathematical bounds obtained in Section 4.3.1, the dotted lines are simulation results. It is clear that for systems with relatively large number of nodes (e.g. $N = 40,\ W = 8 : \gamma = 5$) *Random L Assignment Algorithm* gives the best performance and is close to the mathematical bound. *Contiguous L+1 Assignment Algorithm* has the poorest performance due to tuning overhead. For systems with small number of nodes (e.g. $N = 8,\ W = 8 : \gamma = 1$) *Contiguous L+1 Assignment Algorithm* performs the best while *Contiguous L Assignment Algorithm* the worst. Again the same reasons discussed in Section 6.1.1 apply here.

In Figure 6.2.1.2, blocking probabilities for type *L1* and *L2* sessions are plotted. The dotted lines are simulation results. The solid lines are derived in Section 4.3.1. They represent a system with large number of nodes and where both type of sessions have equal access to *all* system resources. We call them "fair" blocking probabilities.

For systems with relatively large number of users (e.g. $N = 40,\ W = 8 : \gamma = 5$), *Random*

L Assignment Algorithm yields better than "fair" blocking probability for type *L1* sessions and worse than "fair" one for type *L2* sessions. This is expected since sessions with larger value of *L* are more likely to be rejected than those with smaller value simply because it is harder to find more slots at once. For *Contiguous L+1* and *L Assignment Algorithms*, the system is divided into two independent subsystems each serving either type *L1* or *L2* sessions. There are lots of waste in this arrangement since different type of sessions do not share resources with each other. The blocking probabilities for type *L1* and *L2* sessions are both worse than the "fair" ones most of the time. The exception is noted in the case of α equals to 0.8. The blocking probability for type *L2* sessions cross over the "fair" one. The reason is that in the "fair" system a higher proportion of the system resources is used by type *L1* sessions when α is large. Therefore the system becomes more fragmented and less favorable for type *L2* sessions which have larger throughput requirement. In contiguous *L* and *L+1* systems, a fixed amount of resources are reserved for type *L2* sessions. So when the offered load on the system becomes high, type *L2* sessions still have their share of the system resources, and thus have lower than the "fair" blocking probability.

For systems with small number of users (e.g. $N = 8,\ W = 8 : \gamma = 1$), blocking probabilities for type *L1* and *L2* sessions are higher than the "fair" ones for all three algorithms due to heavy transmitter and receiver conflicts. The blocking probability for type *L1* sessions is the highest for *Contiguous L Assignment Algorithm*, and the lowest for *Random L Assignment Algorithm*. The reason is that *Random L Assignment Algorithm* allows sharing of *all* system resources and is the most efficient. For type *L2* sessions, the blocking probability by *Random L Assignment Algorithm* starts lower than that by *Contiguous L+1* and *L Assignment Algorithms*, but eventually becomes higher. The reason is the same as that explained earlier when comparing the blocking probability for type *L2* sessions by *Contiguous L+1* and *L Assignment Algorithm* to the "fair" one.

To summarize, for systems with γ greater than one, the *Random L Assignment Algorithm* achieves the highest channel utilization, also has close to the "fair" blocking probabilities for both type *L1* and *L2* sessions.

In Appendix B.4.1 is the OPNET process model report and the termination criteria.

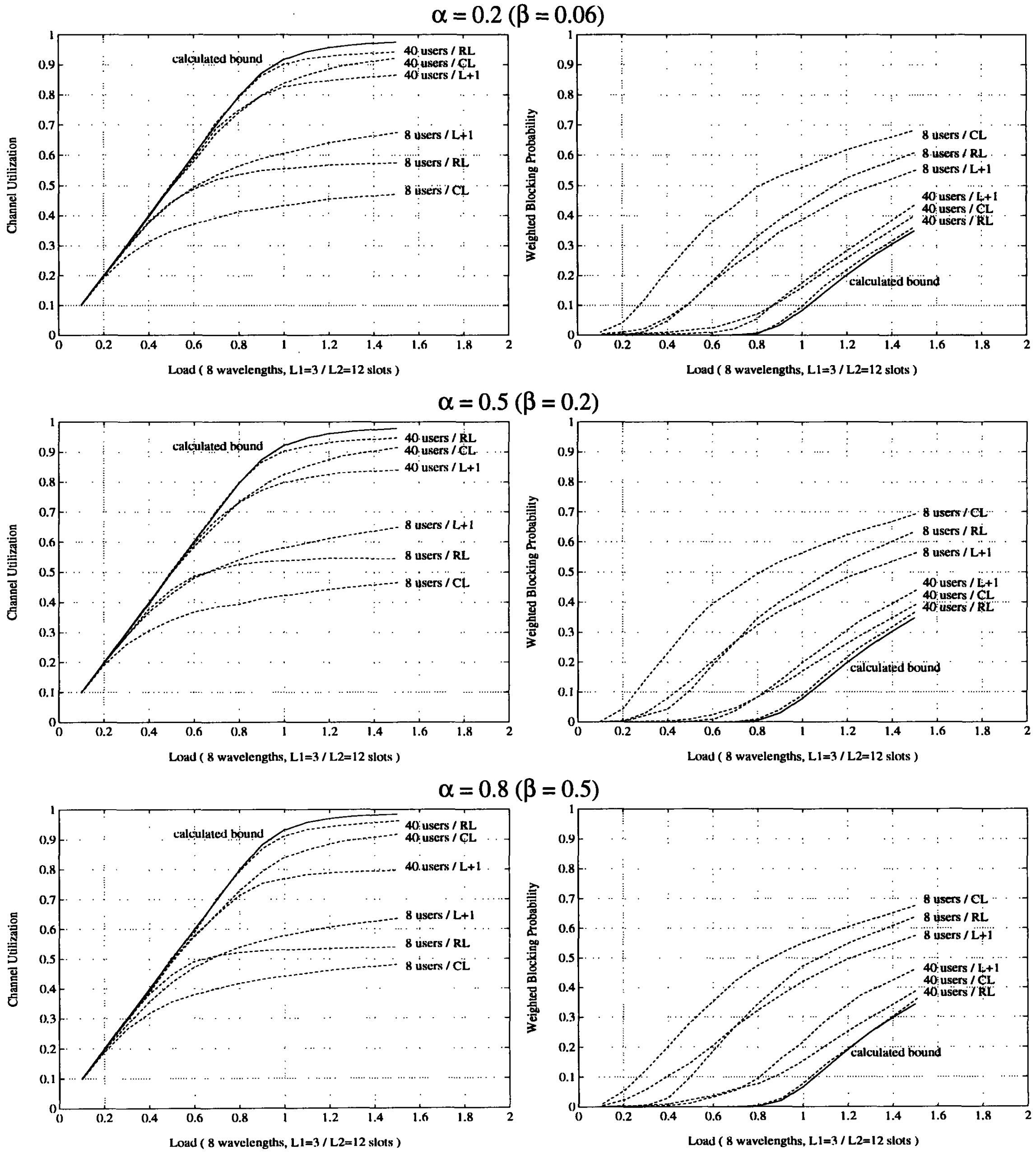


Figure 6.2.1.1 *M/M/2/B/** System (Channel Utilization & Weighted Blocking Probability)

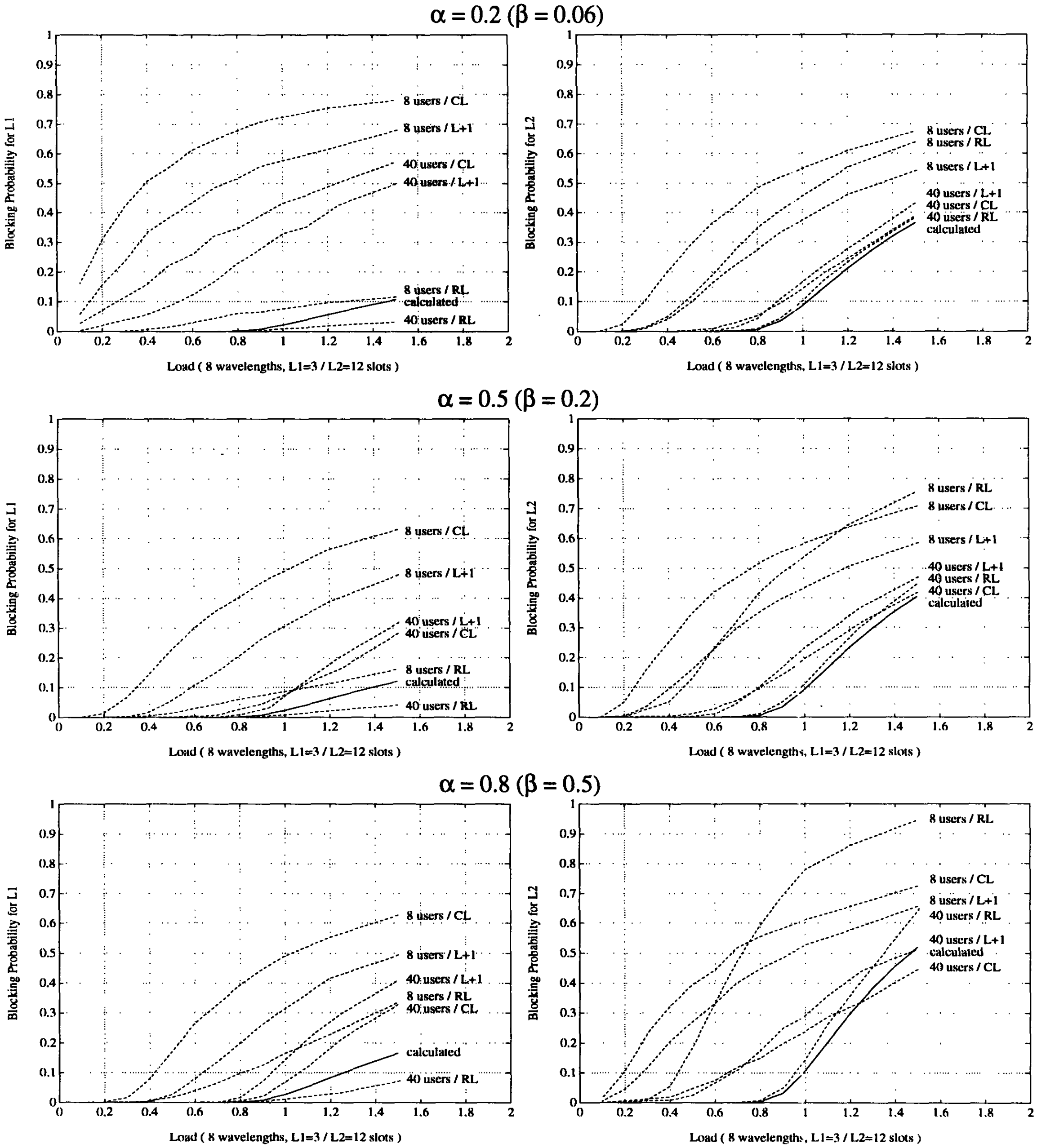
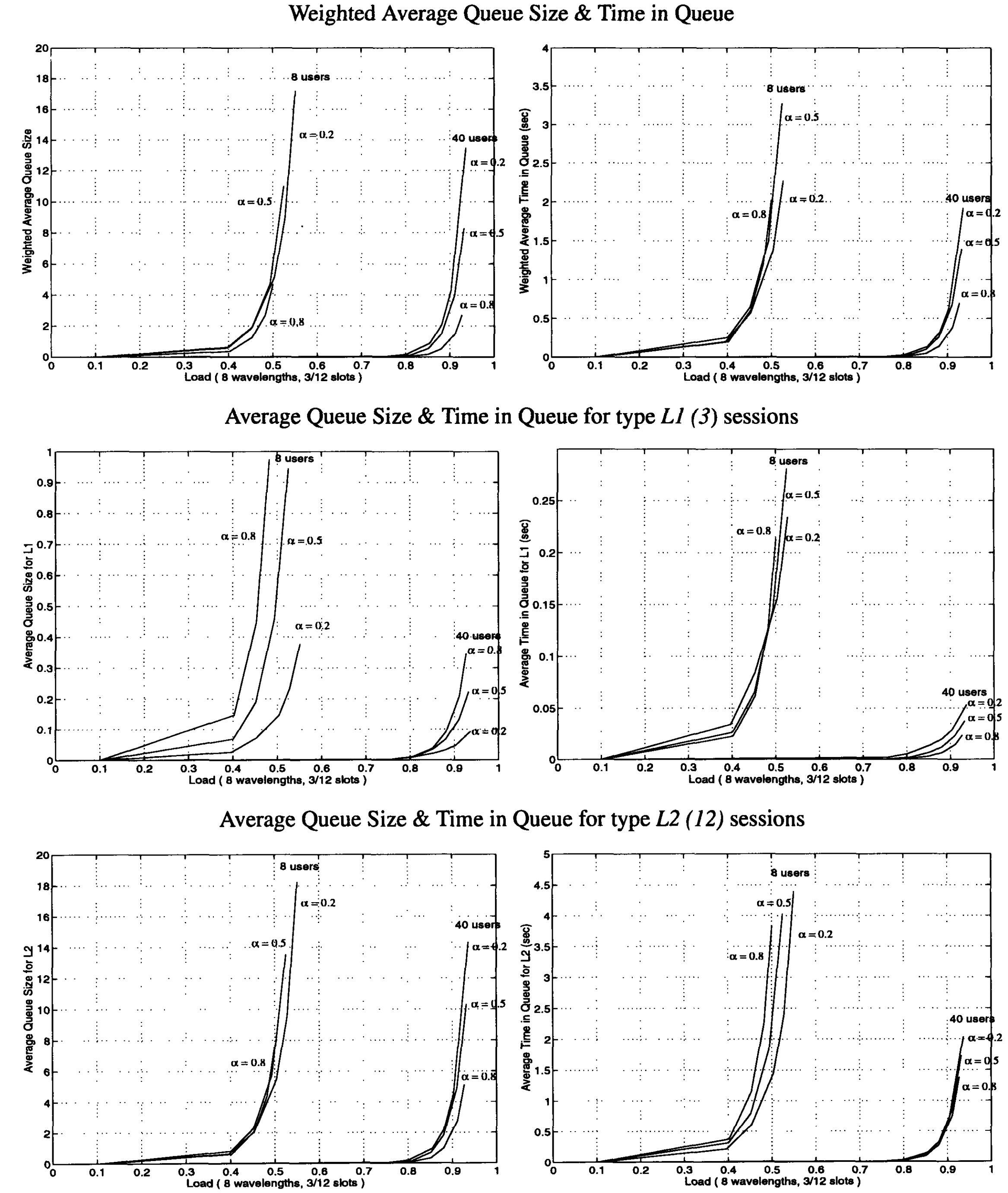


Figure 6.2.1.2 *M/M/2/B/** System (Blocking Probability for L_1 & L_2 Sessions)

6.2.2 Queueing System

Simulation results for *M/M/2/Q/RL* systems are shown in Figure 6.2.2.1. The weighted average queue size (Q) and time in queue (D) are obtained the same way as the weighted blocking probability for the two class blocking system. Let Q_1 and Q_2 be the average queue size for type *L1* and *L2* sessions, D_1 and D_2 the average time in queue. We have $Q = \{\alpha L_1 Q_1 + (1-\alpha) L_2 Q_2\} / \{\alpha L_1 + (1-\alpha) L_2\}$, and $D = \{\alpha L_1 D_1 + (1-\alpha) L_2 D_2\} / \{\alpha L_1 + (1-\alpha) L_2\}$.



Weighted Average Queue Size & Time in Queue

Average Queue Size & Time in Queue for type *L1 (3)* sessions

Average Queue Size & Time in Queue for type *L2 (12)* sessions

Figure 6.2.2.1 *M/M/2/Q/RL* System

As in *M/M/2/B/RL* systems, sessions with larger *L* do not fare as well as those with smaller value. However for systems with relatively large number of users (40 users, 8 wavelength channels), the *Random L Assignment Algorithm* does perform well. A crude comparison is to *M/M/1/Q/RL* systems.

First look at forty users and eight wavelength channels system. For *L* equals to three and at load of 0.9, the average queue size and time in queue are 0.37 and 0.05 respectively for M/M/1/Q/RL system. In this system, the average queue size for type *L1* (3) sessions at load of 0.9 varies from 0.04 to 0.17 (approximate reading from Figure 6.2.2.1) depending on the value of α, and the average time in queue varies from around 0.01 to 0.03. For *L* equals to twelve and at load of 0.9, the average queue size and time in queue are 4.0 and 0.56 respectively for M/M/1/Q/RL system. In this system, the average queue size for type *L2* (12) sessions at load of 0.9 varies from 2 to 4 (approximate reading from Figure 6.2.2.1) depending on the value of α, and the average time in queue is approximately 0.7. So the performance characteristics of both type of classes in the M/M/2/Q/RL systems correspond closely to that in M/M/1/Q/RL systems.

Next look at eight users and eight wavelength channels system. It's clear that the average queue size and time in queue rise sharply at around load of 0.5. In M/M/1/Q/RL system the average queue size and time in queue also rise sharply at around load of 0.5 for both L equals to three and twelve cases.

We therefore conclude that this system compares closely to M/M/1/Q/RL system for each type of the sessions.

In Appendix B.4.2 OPNET process model report is shown, and termination criteria discussed.

6.3 Single Class, Client/Server System

6.3.1 Blocking System

Simulation results for *M/M/1/B/RL* uniform traffic and client/server systems are shown in Figure 6.3.1.1 and Figure 6.3.1.2. The solid lines in Figure 6.3.1.1 are the channel utilization and blocking probability for *M/M/1/B/RL* uniform traffic system. The dotted lines are results for *M/M/1/B/RL* client/server traffic systems. The simulations were run for systems with eight wavelength channels, one or three server nodes, and a total of forty nodes including the servers. The parameter *st* represents the percentage of sessions involving each server's transmitter or receiver. It takes on the value of 5% and 10% in our simulations. For our system of eight wavelength channels, when *st* approaches 1/8, the server's transmitter and receiver become the bottle neck of the system. It's clear when *st* is small (5%) the channel utilization and the blocking probability of the client/server systems are almost identical to the uniform systems. When *st* becomes larger (10%), the client/server systems performs less well than the uniform systems especially for larger number of servers (three servers system is much worse than one server system).

In Figure 6.3.1.2, blocking probabilities for server and uniform traffic are show. Given the same number of servers (1 or 3 here), the blocking probability for server is smaller for system with smaller *st*, and the reverse is true for uniform traffic's blocking probability. The reason is that smaller *st* means that the server's transmitter and receiver are less congested, thus decreasing the server traffic's blocking probability. It also means that higher percentage of the traffic is uniform in nature, thus increasing uniform traffic's blocking probability. For system with the same *st* parameter (0.05 or 0.1 here), the blocking probability for server and uniform traffic is smaller for system with more servers. The reason is that more servers means that lower percentage of the traffic is uniform in nature, thus decreasing uniform traffic's blocking probability. For server traffic, more server means higher percentage of the server traffic (even though the server traffic handled by each server is constant), thus higher blocking probability.

In Appendix B.5.1 OPNET reports for the processor models are included.

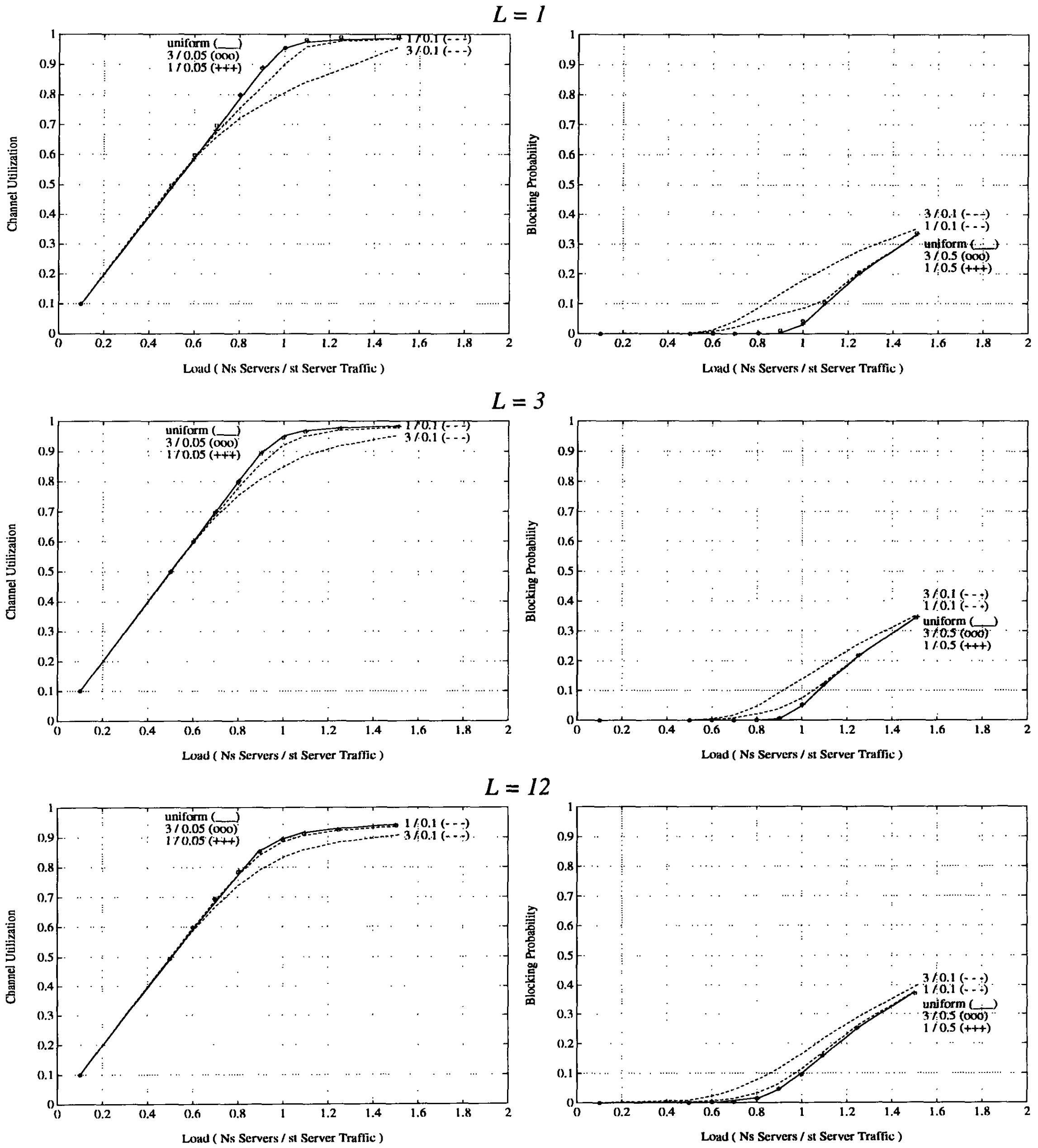


Figure 6.3.1.1 *M/M/1/B/RL* Client-Server System (Channel Utilization & Blocking Probability)

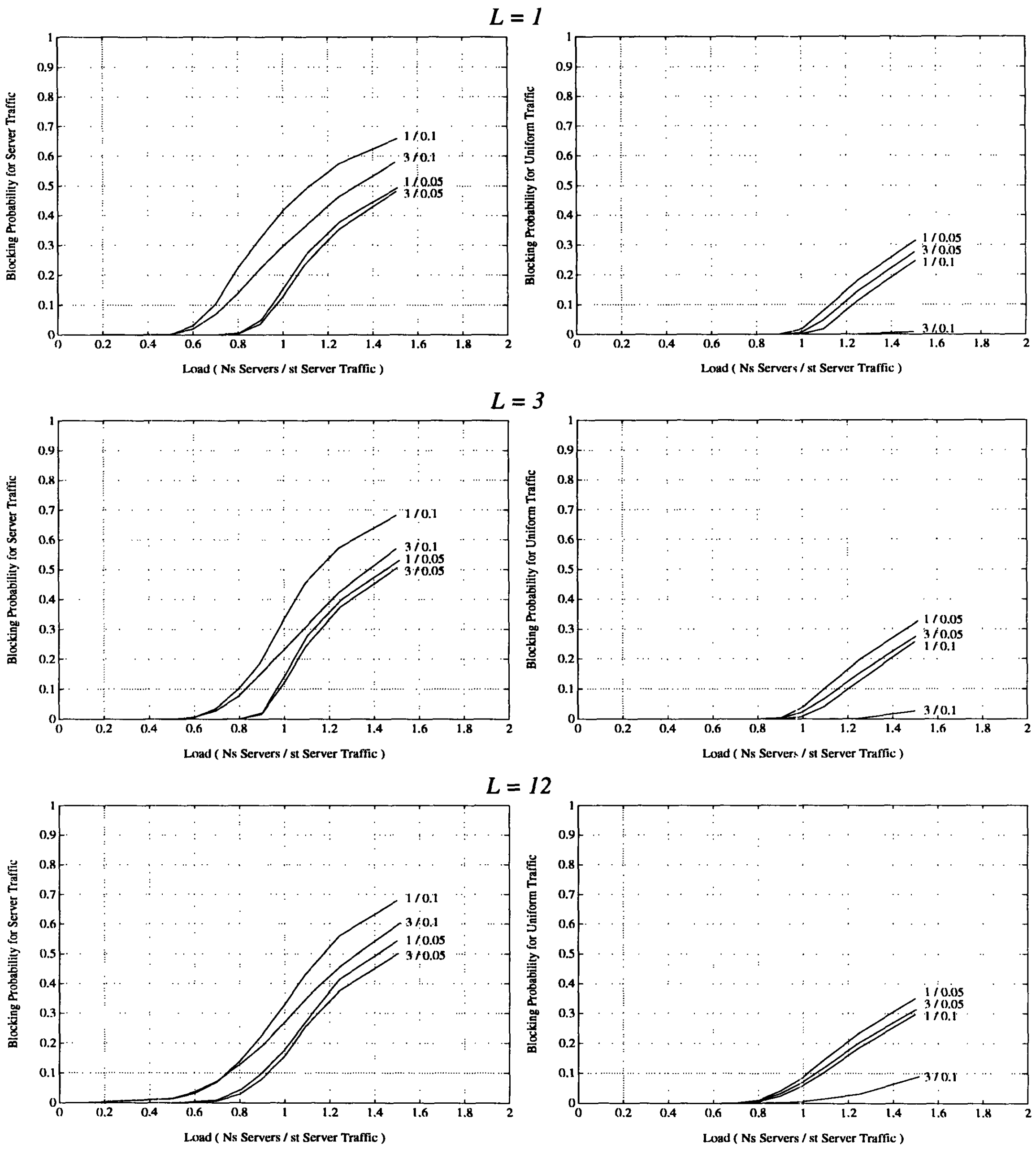


Figure 6.3.1.2 *M/M/1/B/RL* Client-Server System (Blocking Probability for Server & Uniform Sessions)

6.3.2 Queueing System

Simulation results for *M/M/1/Q/RL* uniform traffic and client/server system are shown in Figure 6.3.2.1 through Figure 6.3.2.2. In Figure 6.3.2.1, average queue size (Q) and time in queue (D) for the overall system are shown. In Figure 6.3.2.2 and Figure 6.3.2.2, average queue size for server (Q_s) and uniform (Q_u) traffic, average time in queue for server (D_s) and uniform (D_u) traffic are show, respectively. Since a single queue is maintained for the server and the uniform traffic, $Q = Q_s + Q_u$ and $D = 2*st*N_sD_s + (1-2*st*N_s)D_u$, where *st* is the percentage of total sessions that each server's transmitter or receiver carries and N_S is the number of servers. By comparing Figure 6.3.2.1 to Figure 6.3.1.1 and Figure 6.3.2.2 to Figure 6.3.1.2, we see that the queueing systems have similar relative performance characteristics as the blocking systems, and similar conclusion can be drawn.

In Appendix B.5.2 OPNET reports for the processor models are included.

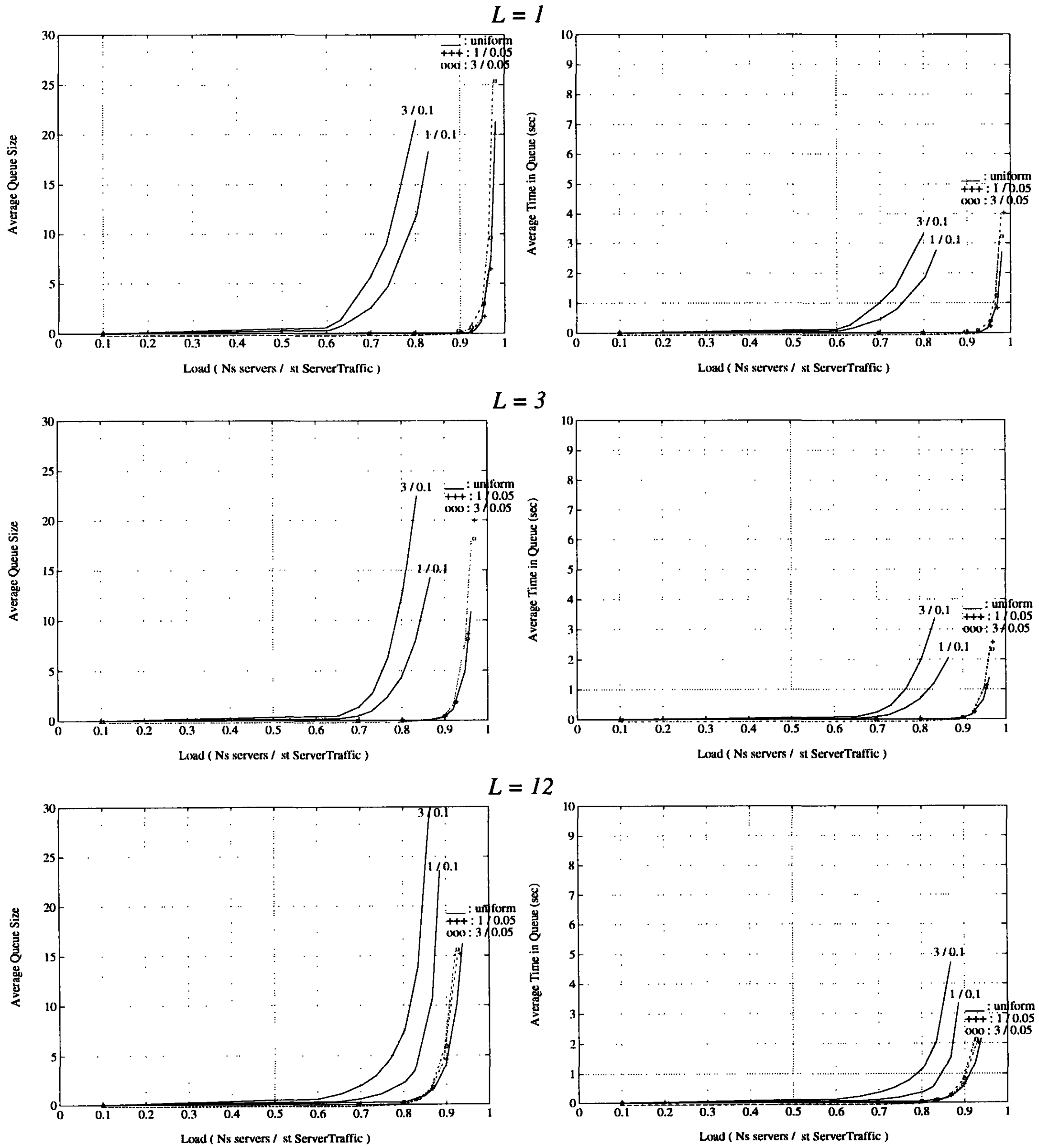


Figure 6.3.2.1 *M/M/1/Q/RL* Client-Server System (for overall system)

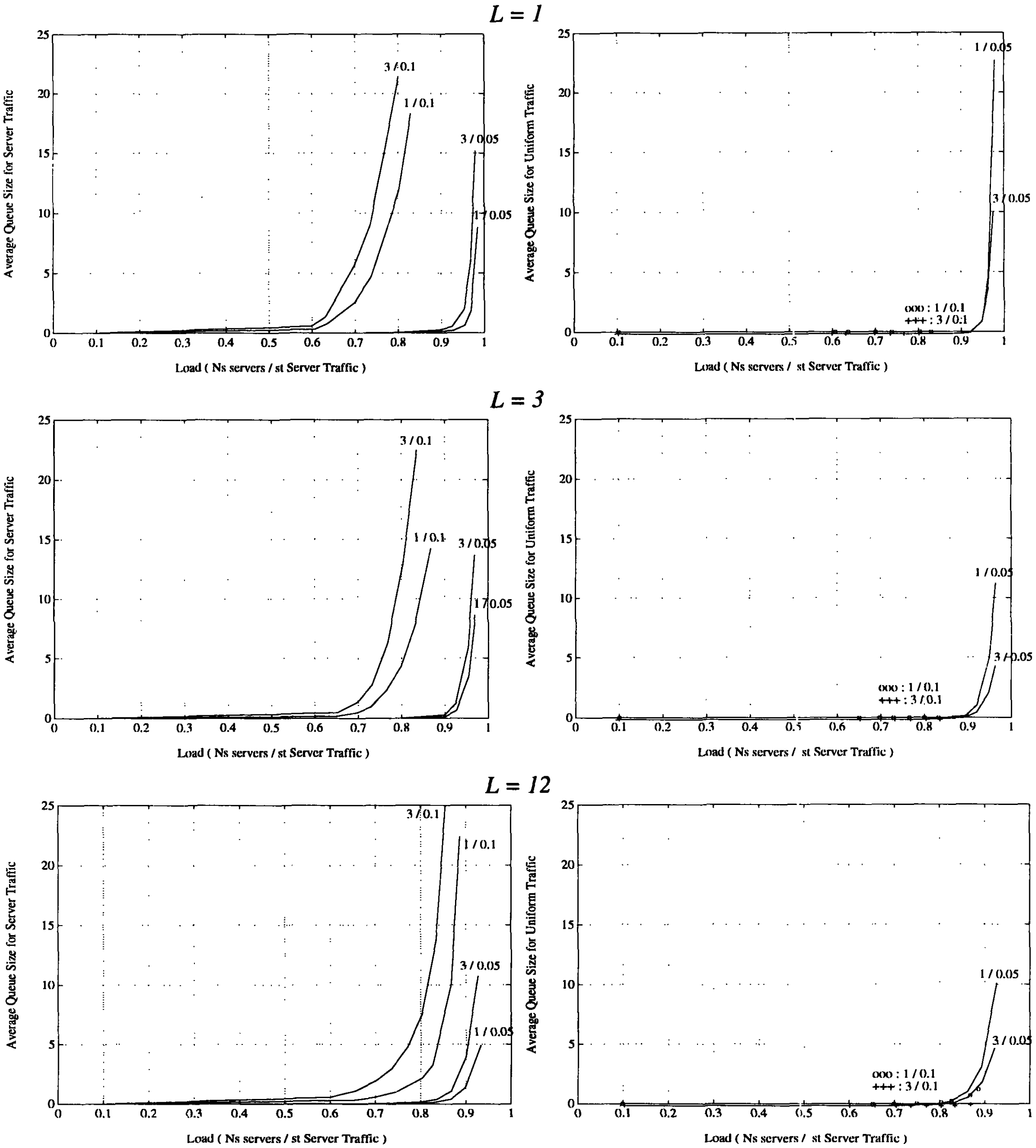


Figure 6.3.2.2 *M/M/1/Q/RL* Client-Server System (Average Queue Size for Server & Uniform Traffic)

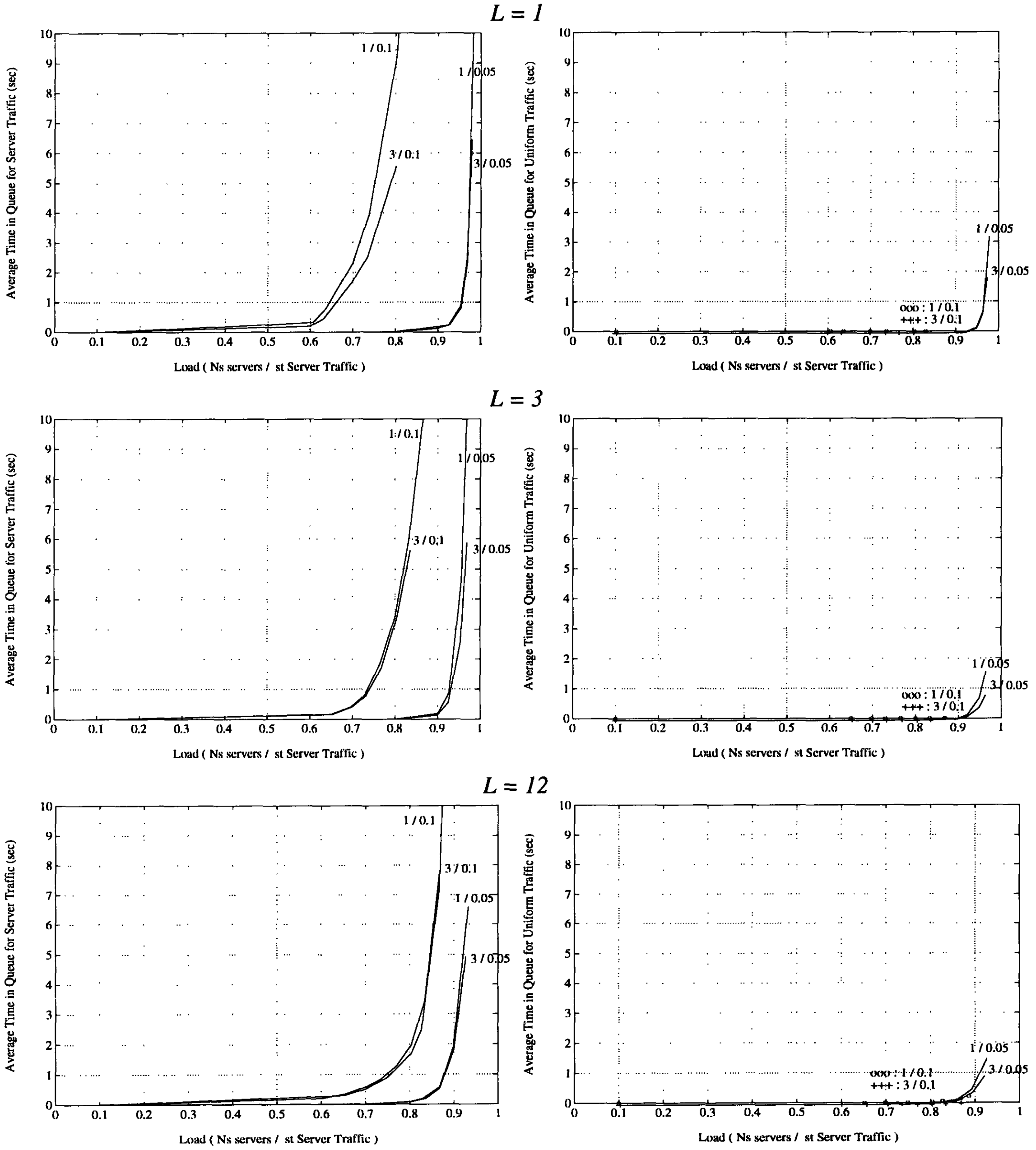


Figure 6.3.2.3 *M/M/1/Q/RL* Client-Server System (Average Time in Queue for Server & Uniform Traffic)

Chapter 7 Summary

From our study of single, two class uniform traffic systems and single class client/server systems, we have shown that a *Random L Assignment Scheduling Algorithm* for level 0 AON subnetworks gives close to the optimal utilization of system resources when the ratio of users to wavelengths γ is large. In which case tuning overhead due to transmitter and receiver conflicts was shown to be a minimal issue. Fragmentation caused by using random slots does not seem to degrade the system performance either. We have also shown that for single class systems with large γ, the *Contiguous L Assignment Scheduling Algorithm* performs only slightly worse than the best *Random L Assignment Scheduling Algorithm*. The trade off here is much reduced algorithmic complexity.

For systems with small numbers of nodes ($\gamma < 2$), scheduling using *Random L Assignment Algorithm* does not always give the best performance compared to *Contiguous L* and *L+1 Assignment Algorithms*. In single class systems when the throughput requirement *L* is larger than one, *Contiguous L+1 Assignment Algorithm* gives the best performance compared to the other two algorithms. However we argue that since the number of nodes (users) are few in the system, the load will also be very light. So the system will be operating at below it's capacity, and *Random L Assignment Scheduling Algorithm* will also work well.

From the study of single class uniform traffic systems with γ greater than one, we showed that the *Random L Assignment Scheduling Algorithm* gives the best performance and it approaches the optimal system performance bound.

For two class uniform traffic systems, we showed that for blocking systems with $\gamma > 1$, the *Random L Assignment Scheduling Algorithm* gives the best performance and again approaches the optimal performance bound. For two class uniform traffic queueing systems, we ran the simulation only using *Random L Assignment Scheduling Algorithm* due to limited computational resources. The results compare well to the system capacity limits (*WT*). The average queue size and average time in queue for each type of session are close to the corresponding ones in the M/M/1/Q/RL case.

We also studied the single class client/server systems using the *Random L Assignment Scheduling Algorithm*. The results were compared to the corresponding single class uniform traffic systems. The parameter *st,* representing the percentage of the total traffic that each server's transmitter or receiver processes, indicates the degree to which the server's transmitter or receiver becomes a bottleneck. When *st* is small compared to $1/W$ (*W* the number of wavelengths), the system performs at close to uniform traffic level even for multiple servers. Therefore the results obtained by assuming uniform traffic pattern appear rather robust.

Our overall conclusion is that the *Random L Assignment Scheduling Algorithm* seems to be a very promising scheduling approach for the AON level 0 subnetwork for a wide variety of traffic requirements including, uniform traffic, multiclass traffic, and client/server traffic.

Appendix A Mathematical Calculations

A.1 Session Blocking Probability

For a blocking uniform traffic system with *W* wavelength channels, frame size *T*, and session throughput request of one slot per frame, we will first derive $P_{b|Ns}$ the probability that a new session *(t,r)* request will be rejected given the number of sessions in service is *Ns*. Assume all *Ns* sessions are distributed equally among the *WT* slots (or *T* columns). To simply the model, we will take into consideration of *Column Conflict*, but not *Pre-Column* and *Post Column Conflict.*

Let the *Ns* sessions in service be distributed among column *(1, ... ,T)* according to $(N_1, \ldots, N_T)$, where N_i is the number of sessions using slots in column *i*, and $Ns = N_1 + \ldots + N_T$. Let *N* be the number of nodes in the system, the probability of blocking for the new session is:

$$P(b|Ns) = \sum_{\Sigma Ni = Ns;\ 1 \le Ni \le min(W, N)} P(b|N_1, \ldots, N_T, Ns) \bullet P(N_1, \ldots, N_T|Ns) \qquad \text{(a.1.1)}$$

The summation is taken over $1 \le N_i \le min(W, N)$. The upper bound is obtained by the fact that the number of slots in use in each columns can be no larger than the number of wavelength channels. In addition it can be no larger than the number of transmitters or receivers) in the system, which is *N* for a system of *N* nodes each with one transmitter(receiver). The lower bound is one since that N_i equals to zero means free slot available for the session, therefore it doesn't contribute to the blocking probability of the session.

To calculate the second term in equation (a.1.1), notice that under the assumption of equally distributed sessions among all columns, we have

$$P(N_1, \ldots, N_T|Ns) = \frac{Ns!}{N_1! \ldots N_T!} \left(\frac{1}{T}\right)^{N_1} \ldots \left(\frac{1}{T}\right)^{N_T} N(\ ) = N(\ ) \cdot \frac{Ns!}{T^{Ns}} \cdot \prod_{i=1}^{T} \frac{1}{N_i!} \qquad \text{(a.1.2)}$$

Where *N()* is the normalization factor, can be obtained by summing $P_{N1, \ldots, Nt|Ns}$ over all combinations of *(N1, ... , Nt)* to one.

$$P(N_1, \ldots, N_T|Ns) = \frac{1}{N(Ns, T, min(W, N))} \cdot \prod_{i=1}^{T} \frac{1}{N_i!} \qquad \text{(a.1.3)}$$

where

$$N(Ns, T, min(W, N)) = \sum_{\Sigma Ni = Ns;\ 0 \le Ni \le min(W, N)} \prod_{i=1}^{T} \frac{1}{N_i!} \qquad \text{(a.1.4)}$$

To calculate the first term in equation (a.1.1), note a session is rejected by the system only if it is rejected by every column in the system. And since we assume only *Column Conflict*, $P_{bi|Ni}$ the probability that a session is rejected by column *i*, given there is N_i busy slots in the column, is independent of each other. Therefore,

$$P(b|N_1, \dots, N_T, Ns) = \prod_{i=1}^{T} P(b_i|N_i) \qquad \text{(a.1.5)}$$

Combine equation (a.1.3) through (a.1.5) into equation (a.1.1), we have

$$P(b|Ns) = \frac{M(Ns, T, min(W, N))}{N(Ns, T, min(W, N))} \qquad \text{(a.1.6)}$$

where *$N(N_S, T, min(W,N))$* is given in equation (a.1.4), and

$$M(Ns, T, min(W, N)) = \sum_{\Sigma Ni = Ns;\ 1 \le Ni \le min(W, N)} \prod_{i=1}^{T} \frac{P(b_i|N_i)}{N_i!} \qquad \text{(a.1.7)}$$

The value of *M(A,B,C) and N(A,B,C)* can be obtained recursively. Here *A* donates the total number of sessions, *B* number of columns, *C* the upper bound for number of sessions in each column. Let *l = A mod C,* the maximum number of columns that can take on the upper bound, then

$$M(A, B, C) = \sum_{i=0}^{l} C(B, i) \cdot \left(\frac{P(bi|C)}{C!}\right)^{i} \cdot M(A - iC, B - i, C - 1) \qquad \text{(a.1.8)}$$

$$N(A, B, C) = \sum_{i=0}^{l} C(B, i) \cdot \left(\frac{1}{C!}\right)^{i} \cdot N(A - iC, B - i, C - 1) \qquad \text{(a.1.9)}$$

where *C(m,n) = m! / n!(m-n)! (m>=n)*, the initial and special conditions are:

$$M(A, B, C) = \begin{cases} 0 & A < B \quad or \quad A > C \cdot B \\ \{P(bi|1)/1!\}^{B} & A = B \\ \{P(bi|C)/C!\}^{B} & A = C \cdot B \\ P(bi|A)/A! & B = 1 \quad and \quad 1 \le A \le C \\ \{P(bi|1)/1!\}^{2B-A} \cdot \{P((bi|2)/2!)\}^{A-B} \cdot C(B, A-B) & C = 2 \end{cases}$$

$$N(A, B, C) = \begin{cases} 0 & ((B = 0 or C = 0)\ and\ (A \ne 0)) \quad or \quad A > C \cdot B \\ 1 & (B = 0 \quad or \quad C = 0) \quad and \quad A = 0 \\ \{1/C!\}^{B} & A = C \cdot B \\ 1/A! & B = 1 \quad and \quad 0 \le A \le C \\ C(B, A) & C = 1 \end{cases}$$

Now the only thing we need is $P_{bi|Ni}$ the probability that a session is blocked by column *i*. It is the same as one minus the probability that it will be accepted. For a session to be accepted, there must be free slots available, i.e $N_i \ne min(W,N)$. Given free slots available, the new session *(t,r)*'s transmitter and receiver cannot be in use by any of the sessions already in service associated with the busy slots in the column. Let $\bar{t}$ and $\bar{r}$ donate that node *t*'s transmitter and node *r*'s receiver are free, we have

$$\begin{aligned} 1 - P(b_i | N_i) &= P(N_i \neq min(W,N) | N_i) \bullet P(\bar{t} | N_i \neq (min(W,N), N_i)) \bullet P(\bar{r} | \bar{t}, N_i \neq (min(W,N), N_i)) \\ &= \delta(N_i - min(W,N)) \bullet P(\bar{t} | N_i) \bullet P(\bar{r} | \bar{t}, N_i) \end{aligned} \qquad \text{(a.1.10)}$$

where $\delta_n = \begin{cases} 0 & n = 0 \\ 1 & n \neq 0 \end{cases}$.

The derivation for $P_{\bar{t}|Ni}$ is straightforward, it is the probability that picking N_i transmitters out of total N transmitters resulted in node *t*'s transmitter not picked.

$$P(\bar{t} | Ni) = \frac{C(N-1, Ni)}{C(N, Ni)} = 1 - \frac{Ni}{N} \qquad \text{(a.1.11)}$$

Let $L(N,N_i)$ be the number of choices of picking N_i sessions out of a system of N nodes, where each node can transmit to any other node but itself. Let $L_1(N,N_i)$ be the number of choices of picking N_i sessions out of a system of N nodes, where node *t* is not transmitting at all, and all other node can transmit to any other but itself. Let $L_2(N,N_i\text{-}1)$ be the number of choices of picking N_i sessions out of a system of N nodes, where node *t* is transmitting to one of the possible $N\text{-}1$ nodes, say node *r*, and the rest of the nodes can transmit to each other but itself and node *r*. Then

$$L(N, Ni) = L1(N, Ni) + (N-1) \cdot L2(N, Ni-1) \qquad \text{(a.1.12)}$$

The probability that node *t* is transmitting is $(N\text{-}1)L_2(N,N_i\text{-}1) / L(N,N_i)$ from the definition of *L, L1,* and *L2*. It is also the probability of picking N_i transmitters out of N with equal probability with node *t*'s transmitter picked, therefore

$$\frac{(N-1)L2(N, Ni-1)}{L(N, Ni)} = \frac{C(N-1, Ni-1)}{C(N, Ni)} = \frac{Ni}{N} \qquad \text{(a.1.13)}$$

The definition of $L_2(L,N_i\text{-}1)$ can be reiterated as the number of choices of picking $N_i\text{-}1$ sessions out of a system of N nodes, where a node, say *t*, is not transmitting and another node, say *r*, not receiving, and all other nodes can transmit to each others but itself. Then it's easy to see that

$$P(\bar{r} | \bar{t}, Ni) = \frac{L2(N, Ni)}{L1(N, Ni)} \qquad \text{(a.1.14)}$$

Combine equation (a.1.12) through (a.1.14), we can solve for $P_{\bar{r}|\bar{t},Ni}$ in terms of L_2, plug that and equation (a.1.11) into equation (a.1.10), we have:

$$1 - P(bi | Ni) = \delta(Ni - min(W,N)) \cdot \frac{Ni}{N(N-1)} \cdot \frac{L2(N, Ni)}{L2(N, Ni-1)} \qquad \text{(a.1.15)}$$

To solve for *L2*, we need one more set of equation. Consider $L_2(N,N_i)$, the number of choices of picking N_i sessions out of a system of N nodes where a node, say *t*, is not transmitting and a different node, say *r*, is not receiving. It can be further broken into cases according to the status of node *r*'s transmitter. In case of node *r* not transmitting, we can take this node out of the system of N nodes since neither of its transmitter or receiver is busy. So the number of choices is the same as picking N_i sessions out of a system of *N-1* nodes where node *t* is not transmitting, i.e $L_1(N\text{-}1,N_i)$. Now consider the case where node *r* is transmitting to either node *t* or one of the rest of the *N-2* nodes. In the former case, we have node *t* not transmitting, node *r* not receiving, and a session *(r,t)*. The rest of the N_i*-1* sessions are happening among the other *N-2* nodes, so the number of choices are $L(N\text{-}2,N_i\text{-}1)$. In the later case, we have node *t* neither transmitting nor receiving, and therefore can be discounted. Node *r* is transmitting to one of the *N-2* nodes, but not receiving. By definition the number of choices of picking N_i session in such a system is $(N\text{-}2)L_2(N\text{-}1,N_i\text{-}1)$. Therefore

$$L2(N, Ni) = L1(N-1, Ni) + L(N-2, Ni-1) + (N-2)\cdot L2(N-1, Ni-1) \qquad (a.1.16)$$

Combine equation (a.1.12), (a.1.13) and (a.1.16), and with initial condition of $L2(*,0) = 1$, we can solve L_2 recursively:

$$L2(N, Ni) = \frac{(N-1)(N-2)}{Ni}\cdot L2(N-1, Ni-1) + \frac{(N-2)(N-3)}{Ni-1}\cdot L2(N-2, Ni-2) \qquad (a.1.17)$$

where

$$L2(N, 0) = 1 \qquad and \qquad L2(N, 1) = N^2 - 3N + 3 \qquad (a.1.18)$$

A.2 Single Class, Uniform Traffic System

The C programs used to calculate mathematical approximation and bounds are included.

A.2.1 Blocking System

Mathematical Approximation for M/M/1/B/L+1 and M/M/1/B/CL

The following program is used to calculate the mathematical approximation/bounds as in Section 4.2.1 for M/M/1/B/L+1 system by setting parameter *b* to 1, and for M/M/1/B/L by setting *b* to 0. The added complexity in the program is due to resolving number overflow.

```
#include <math.h>

double fact(int x);
double comb(int x, int y);
double N(int A, int B, int C);
double M(int A, int B, int C);
double L(int N, int i);
void  pbf();

int   Nn,T,min1,mct,mct_old,nct,nct_old;
double pbi[20];


void main()
{
int   i,j,b,W,Nx,L,T0,Ns;
double pb,m,m1,pa[1030],ps[1030];
double p,p1,s1[10],s2[10],s3[10];

extern int   Nn,T,min1,mct,nct;
extern double pbi[];

W=8;  L=1;  T0=128;  Nn=8;  b=1;
p=T0/(L+b);  T=floor(p);  Nx=W*T;
/*-----------------------------------------------
Getting pa[i]
-----------------------------------------------*/
min1=W;
```

```
if (Nn<W) min1=Nn;
pbf();
for (Ns=0; Ns<=Nx; Ns++)
  {
  if (Ns<T)     { pa[Ns]=1; continue; }
  if (Ns>=Nn*T || Ns==Nx)     { pa[Ns]=0; continue; }
  pb=M(Ns,T,min1);  m1=N(Ns,T,min1);  pb=pb/m1;
  for (i=0; i<(mct+nct); i++) pb=pb/1e15;
  pa[Ns]=1-pb;
  }

/*---------------------------------------
     Getting ps[i]
------------------------------------- */
for (p=0.1;p<1.6;p+=0.1)
  {
  p1=p*W*T0/L; ps[0]=1; pb=0; m1=0; m=1; j=0;
  for (i=1;i<=Nx;i++)
    {
    ps[i]=ps[i-1]*p1*pa[i-1]/i;
    m+=ps[i];
    pb+=ps[i]*(1-pa[i]);
    m1+=i*ps[i]*L/(T0*W);
    if (m>1e30)
      { s1[j]=m; m=0; s2[j]=pb; pb=0; s3[j]=m1; m1=0; ps[i]=ps[i]/1e30; j+=1; }
    }
  for (i=0; i<j; i++)
    {
    for (b=0; b<(j-i); b++)
      { s1[i]=s1[i]/1e30;  s2[i]=s2[i]/1e30;  s3[i]=s3[i]/1e30; }
    m+=s1[i];  pb+=s2[i];  m1+=s3[i];
    }
  pb=pb/m;  m1=m1/m;
  printf("%20.9f\t%20.9f\n",pb,m1);
  }
}


void pbf()
{
int i,k;
double m,l,n1,n2;
extern int   Nn,min1;
extern double pbi[];

k=Nn*(Nn-1);  n1=1;  n2=L(Nn,1);  pbi[0]=0;  pbi[min1]=1;
for (i=1;i<min1;i++)
  { pbi[i]=1-i*n2/(n1*k);  n1=n2;  n2=L(Nn,i+1);  }
}


double L(int N, int i)
{
double R;

if (i == 0) { R=1.0; return(R); }
if (i==1) { R=(double)(N*N-3*N+3); return(R); }
R=(N-1)*(N-2)*L(N-1,i-1)/i + (N-2)*(N-3)*L(N-2,i-2)/(i-1);
return(R);
}


double comb(int x, int y)
{
double R=1.0;
int    i,j;

j=x-y;
if (x<0 || y<0) return(0);
if (j<0) return(0);
for (i=1;i<=j;++i) R=R*(y+i)/i;
return(R);
}


double fact(int x)
{
double R=1.0;
int    i;
for (i=1;i<=x;i++) R=R*i;
return(R);
}


double N(int A, int B, int C)
{
int i,j,p,nct_keep,nct1_keep,nct1;
double R=0,k,l,m,n,n1;
extern int nct,nct_old;

nct=0;
if (B==0) if (A==0) return(1); else return(0);
if (C==0) if (A==0) return(1); else return(0);
if (A > B*C) return(0);
if (A == B*C)
 {
 l=1/fact(C);  R=1.0;
 for (i=1;i<=B;i++)
  {
  R=R*l;
  if (R<1e-15) { R=R*1e15; nct-=1; }
  }
 return(R);
 }

if (B==1)
 if (A>=0 && A<=C)
  {
  R=1/fact(A);
  if (R<1e-15) { R=R*1e15; nct-=1; }
  return(R);
  }
 else return(0);

if (C == 1)
 {
 R=1.0;
 for (i=1; i<=A; i++)
  {
  R=R*(B-A+i)/i;
  if (R>1e15) { R=R/1e15; nct+=1; }
  }
 return(R);
 }

l=(double)A/(double)C;  l=floor(l);  m=1/fact(C);  p=A-B*(C-1);
if (p<0) p=0;

R=N(A-p*C,B-p,C-1);  nct1=0;  n=1;
for (i=1; i<=p; i++)
 {
 n=n*m*(B-p+i)/i;
 if (n<1e-15) { n=n*1e15; nct1-=1; }
 if (n>1e15) { n=n/1e15; nct+=1; }
 }
nct+=nct1;  R=R*n;
if (R<1e-15) { R=R*1e15; nct-=1; }
if (R>1e15) { R=R/1e15; nct+=1; }
nct_old=nct;  p+=1;
for (i=p;i<=l;i++)
 {
 n=n*m*(B+1-i)/i;
 if (n<1e-15) { n=n*1e15; nct1-=1; }
 if (n>1e15) { n=n/1e15; nct+=1; }
 nct_keep=nct_old; nct1_keep=nct1;
 k=n*N(A-i*C,B-i,C-1);
 nct_old=nct_keep; nct1=nct1_keep;
 nct=nct+nct1;
 if (k<1e-15) { k=k*1e15; nct-=1; }
 if (k>1e15) { k=k/1e15; nct+=1; }
 if (nct_old==nct) R+=k;
 else  {
     if (nct_old>nct)
      {
      for (j=0; j<(nct_old-nct); j++) k=k/1e15;
      R+=k;
      }
     else
      {
      for (j=0; j<(nct-nct_old); j++) R=R/1e15;
      nct_old=nct;  R+=k;
      }
     }
 }
nct=nct_old;
return(R);
}


double M(int A, int B, int C)
{
int i,j,p,mct_keep,mct1,mct1_keep;
double R=0,k,l,m,n;
extern int   mct,mct_old;
extern double pbi[];

mct=0;
if (A<B) return(0);
if (A>C*B) return(0);

if (A==B)
 {
 l=pbi[1];
 R=1.0;
 for (i=0; i<B; i++)
  {
  R=R*l;
  if (R<1e-15) { R=R*1e15; mct+=1; }
  }
 return(R);
 }

if (A==C*B)
 {
 l=pbi[C]/fact(C);
 R=1.0;
 for (i=1;i<=B;i++)
  {
  R=R*l;
  if (R<1e-15) { R=R*1e15; mct+=1; }
  }
 return(R);
 }

if (B==1)
 if (A>=1 && A<=C)
  {
  R=pbi[A]/fact(A);
  if (R<1e-15) { R=R*1e15; mct+=1; }
  return(R);
  }
 else return(0);

if (C==1)
 {
 R=1.0;
 for (i=1;i<=A;i++)
  {
  R=R*pbi[1];
  if (R<1e-15) { R=R*1e15; mct+=1; }
  }
 return(R);
 }

if (C==2)
 {
 l=pbi[1];  m=pbi[2]/2;  R=1.0;
 for (j=1; j<=(A-B); j++)
  {
  R=R*m*(2*B-A+j)/j;
  if (R<1e-15) { R=R*1e15; mct+=1; }
  }
 for (j=1; j<=(2*B-A); j++)
  {
  R=R*l;
  if (R<1e-15) { R=R*1e15; mct+=1; }
  }
 return(R);
 }

l=(double)A/(double)C;  l=floor(l);  m=(double)(A-B)/(double)(C-1);  m=floor(m);
if (l>m) l=m;
m=pbi[C]/fact(C);  p=A-B*(C-1);
if (p<0) p=0;

R=M(A-p*C,B-p,C-1);
mct1=0;  n=1;
for (i=1;i<=p;i++)
 {
 n=n*m*(B-p+i)/i;
 if (n<1e-15) { n=n*1e15; mct1=mct1+1; }
 }
mct+=mct1;  R=R*n;
if (R<1e-15) { R=R*1e15; mct+=1; }
mct_old=mct;   p+=1;
for (i=p;i<=l;i++)
 {
 n=n*(B+1-i)*m/i;
 if (n<1e-15) { n=n*1e15; mct1+=1; }
 mct_keep=mct_old; mct1_keep=mct1;
 k=n*M(A-i*C,B-i,C-1);
 mct_old=mct_keep; mct1=mct1_keep;
 mct=mct+mct1;
 if (k<1e-15) { k=k*1e15; mct+=1; }
 if (k>1.0) { k=k/1e15; mct-=1; }
 if (mct_old==mct) R+=k;
 else {
    if (mct_old>mct)
      {
      for (j=0; j<(mct_old-mct); j++) R=R/1e15;
      mct_old=mct;  R+=k;
      }
     else
      {
      for (j=0; j<(mct-mct_old); j++) k=k/1e15;
      R+=k;
```

```
        }
      }
  }
  mct=mct_old;
  return(R);
  }
```

Mathematical Bounds for M/M/1/B/RL

```
#include <math.h>

void main()
{
int i,j,k,b,W,T,L,m;
double p,a,q,s,Pq,Nq,ss[20];

W=8;  L=7;  T=128;
p=(double)(W*T)/(double)L;  m=floor(p);

for (p=0.1; p<1.6; p+=0.1)
  {
  a=p*W*T/L;  q=1;  s=1;  jj=0;
  for (i=1; i<m; i++)
    {
    q=q*a/i;  s+=q;
    if (s>1e30) { ss[j]=s; s=0; q=q/1e30; j+=1; }
    }
  for (i=0; i<j; i++)
    {
    for (k=0; k<(j-i); k++) ss[i]=ss[i]/1e30;
    s+=ss[i];
    }
  q=q*a/m;  Pq=q/(s+q);  Nq=(1-Pq)*p;
  printf("%f\t%f\t%f\n",p,Nq,Pq);
  }
}
```

A.2.2 Queueing System

The following program is used to calculate mathematical bounds as in Section 4.2.2 for M/M/1/Q/L+1, M/M/1/Q/CL, and M/M/1/Q/RL systems. The added complexity is due to resolving number overflow.

```
#include <math.h>

void main()
{
int i,j,k,b,W,T,L,m;
double p,a,q,s,Pq,Nq,ss[20],D;

b=0;  W=4;  L=3;  T=128;

/* following for CL(b=0) & L+1(b=1) assignment */
/* p=(double)T/(double)(L+b);  m=W*floor(p);*/

/* following for RL assignment */
p=(double)(W*T)/(double)L;  m=floor(p);

for (p=0.1; p<1.0; p+=0.1)
  {
  a=p*m;  q=1;  s=1;   j=0;
  for (i=1; i<m; i++)
    {
    q=q*a/i;  s+=q;
    if (s>1e30) { ss[j]=s; s=0; q=q/1e30; j+=1; }
    }
  for (i=0; i<j; i++)
    {
    for (k=0; k<(j-i); k++) ss[i]=ss[i]/1e30;
    s+=ss[i];
    }
  q=q*a/m;  Pq=q/(1-p);  Pq=Pq/(s+Pq);  Nq=Pq*p/(1-p);  a=a*L/(W*T);  D=Nq/(a*W);
  printf("%f\t%f\t%f\n",a,Nq,D);
  }
}
```

A.3 Two Class, Uniform Traffic System

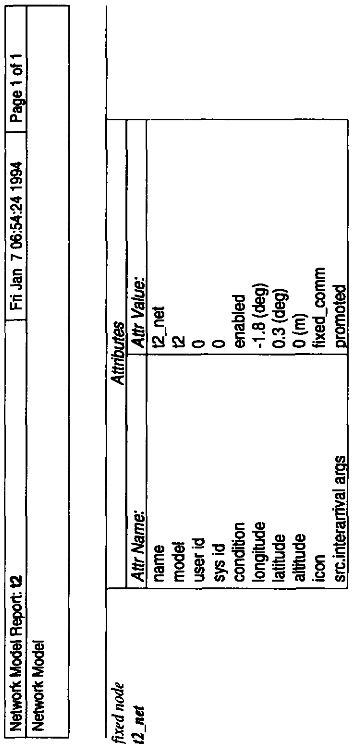
A.3.1 Blocking System

The following program is used to calculate mathematical bounds as in Section 4.3.1 for M/M/2/B/* systems. The added complexity is due to number overflow.

```
#include <math.h>

#define L 50

void main()
{
int   W,T,L1,L2,i,j,k,m1,m2,il,ict,ict1,jct,jct1;
double a,p,p1,p2,q,s0,s1,s2,s3,c0,c1,c2,c3,c4,cu,pb1,pb2,pb,pp;
double ss2[L],ss3[L],cc2[L],cc3[L],cc4[L],pp1[L],pp2[L];

W=8;  a=0.8;  L1=3;  L2=12;  T=128;

for (p=0.1; p<1.6; p+=0.1)
  {
  p1=a*p*W*T/(a*L1+(1-a)*L2);    p2=(1-a)*p1/a;
  q=(double)(W*T)/(double)L2;     m2=floor(q);
  q=(double)(W*T)/(double)L1;     m1=floor(q);
  s0=1; c0=0; jct=0;
  for (j=0; j<L; j++) { ss2[j]=0; cc2[j]=0; cc3[j]=0; pp1[j]=0; pp2[j]=0; }
  for (j=0; j<=m2; j++)
    {
    q=(double)(W*T-(j+1)*L2)/(double)L1;     il=floor(q);
    s1=1; c1=0; ict=0; pp=0;
    for (i=0; i<L; i++) { ss3[i]=0; cc4[i]=0; }
    for (i=0; i<=m1;  i++)
      {
      if (i!=0) s1*=p1/i;                          /* s1=p1~i/i! */
      if (i==1) c1=p1; if (i>1) c1*=p1/(i-1);      /* c1=ip1~i/i! */
      if (ict>=0) { ss3[ict]+=s1; cc4[ict]+=c1; }  /* s3=sum_i(s1) ;  c4=sum_i(c1) */
      if (i>il && i<=m1 )                          /* pb=sum_il_ml(s1) */
        {
        if (i==(il+1)) ict1=ict;
        pb=s1;
        if (ict>ict1) for (k=0; k<(ict-ict1); k++) pb*=1e15;
        if (ict<ict1) for (k=0; k<(ict1-ict); k++) pb/=1e15;
        pp+=pb;
        }
      if (c1>1e15)
        { s1=s1/1e15; c1=c1/1e15; ict+=1; }
      if (s1<1e-15)
        { s1=s1*1e15; c1=c1*1e15; ict-=1; }
      }  /* end of for_i */
    if (ict>ict1) for (k=0; k<(ict-ict1); k++) pp/=1e15;
    if (ict<ict1) for (k=0; k<(ict1-ict); k++) pp*=1e15;
    pb=pp; ict1=ict;
    s3=0; c4=0; ict=-1;
    for (i=(L-1); i>=0; i--)
      {
      if (ss3[i]==0 && ict==-1) continue;
      if (ict==-1) ict=i;
      for (k=0; k<i; k++) { ss3[k]*=1e-15; cc4[k]*=1e-15; }
      s3+=ss3[i]; c4+=cc4[i];
      }
    if (j!=0) s0*=p2/j;                            /* s0=p2~j/j! */
    if (j==1) c0=p2; if (j>1) c0*=p2/(j-1);     /* c0=jp2~j/j! */
    k=ict+jct;
    if (k>=0)
      {
      ss2[k]+=s0*s3;               /* s2=sum_j( p2~j/j!.sum_i(p1~i/i!) ) */
      cc2[k]+=s0*c4;               /* c2=sum_j( p2~j/j!.sum_i(ip1~i/i!) ) */
      cc3[k]+=c0*s3;               /* c3=sum_j( jp2~j/j!.sum_i(p1~i/i!) ) */
      }
    if (j==0) jct1=ict1+jct;
    k=ict1+jct-jct1;
    if (k>=0) { pp1[k]+=s0*s1; pp2[k]+=s0*pb; }
    if (c0>1e15)
      { s0=s0/1e15; c0=c0/1e15; jct+=1; }
    if (s0<1e-15)
      { s0=s0*1e15; c0=c0*1e15; jct-=1; }
    m1=il;
    }  /* end of for_j */

  jct=-1; s2=0; c2=0; c3=0;
  for (i=(L-1); i>=0; i--)
    {
    if (ss2[i]==0 && jct==-1) continue;
    if (jct==-1) jct=i:
    for (k=0; k<i; k++) { ss2[k]/=1e15; cc2[k]/=1e15; cc3[k]/=1e15; }
    s2+=ss2[i]; c2+=cc2[i]; c3+=cc3[i];
    }
  cu=(L1*c2+L2*c3)/(W*T*s2);
  ict=-1; pb1=0; pb2=0;
  for (i=(L-1); i>=0; i--)
    {
    if (pp1[i]==0 && ict==-1) continue;
    if (ict==-1) ict=i;
    for (k=0; k<i; k++) { pp1[k]/=1e15; pp2[k]/=1e15; }
    pb1+=pp1[i]; pb2+=pp2[i];
    }
  ict+=jct1;
  pb1/=s2; pb2/=s2;
  if (ict<jct1) for (k=0; k<(jct1-ict); k++) { pb1/=1e15; pb2/=1e15; }
  if (ict>jct1) for (k=0; k<(jct1-ict); k++) { pb1*=1e15; pb2*=1e15; }
  pb=a*pb1+(1-a)*pb2;
  printf("%f\t%f\t%f\t%f\t",cu,pb,pb1,pb2);
  pb=(L1*a*pb1+L2*(1-a)*pb2)/(L1*a+L2*(1-a));
  printf("%f\n",pb);
  }
}
```

B.1 Network Model

The following is the network model "t2_net" used. It consists of only one node named "t2" whose model is included in Section B.2.



Network Model Report: t2 | Fri Jan 7 06:54:24 1994 | Page 1 of 1

Network Model

fixed node
t2_net

Attributes

Attr Name:	*Attr Value:*
name	t2_net
model	t2
user id	0
sys id	0
condition	enabled
longitude	-1.8 (deg)
latitude	0.3 (deg)
altitude	0 (m)
icon	fixed_comm
src.interarrival args	promoted

B.2 Node Model

The following is the node model "t2" used. Notice that the processor model "t2_scheduler" in the ***queue scheduler*** component and the processor model "t2_release" in the ***processor release*** component are to be substituted by "mm*_scheduler" and "mm*_release" in Section B.3 through Section B.5.

Node Model Report: t2	Fri Jan 7 06:55:28 1994	Page 2 of 2
Node Model		

processor
release

Attributes	
Attr Name:	*Attr Value:*
name	release
in strm count	8
out strm count	8
process model	t2_release
intrpt interval	disabled
begsim intrpt	disabled
endsim intrpt	disabled
failure intrpts	disabled
recovery intrpts	disabled
icon name	processor

Streams		
Index	*Input Streams:*	*Output Streams:*
0	scheduler [0]	

Node Model Report: t2	Fri Jan 7 06:55:28 1994	Page 1 of 2
Node Model		

ideal generator
src

Attributes	
Attr Name:	*Attr Value:*
name	src
interarrival pdf	exponential
interarrival args	promoted
pk size pdf	constant
pk size args	1024.0
field (0) pdf	constant
field (0) args	0.0
packet format	t2
icon name	ideal_gen

Streams		
Index	*Input Streams:*	*Output Streams:*
0		scheduler [0]

queue
scheduler

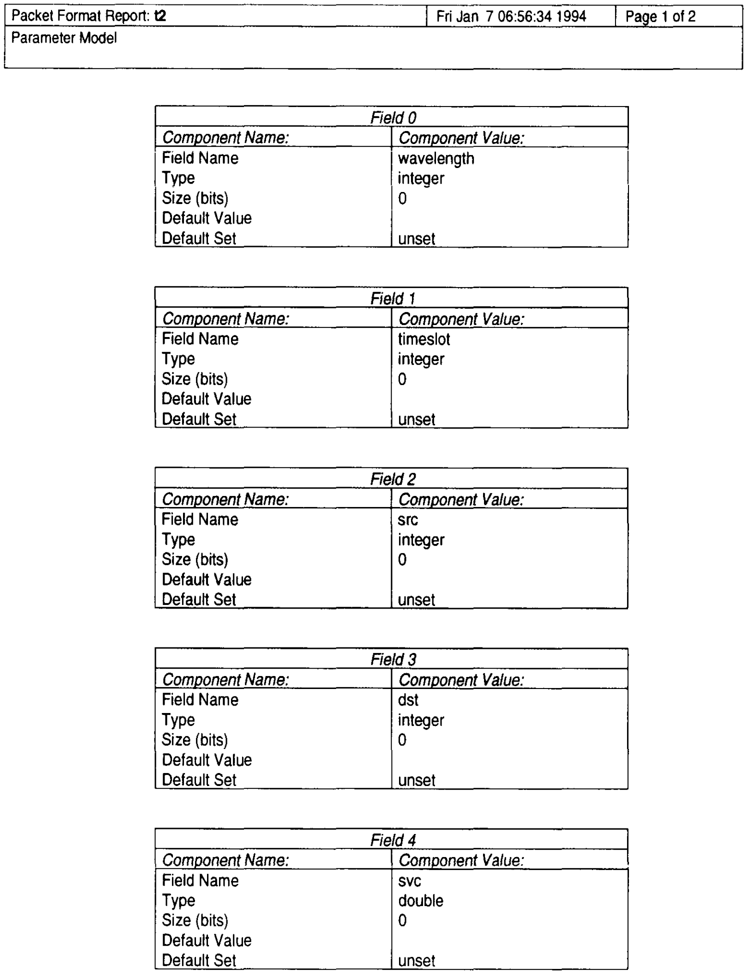
Attributes	
Attr Name:	*Attr Value:*
name	scheduler
in strm count	8
out strm count	8
subqueue count	2
process model	t2_scheduler
default priority	0
intrpt interval	disabled
begsim intrpt	enabled
endsim intrpt	enabled
failure intrpts	disabled
recovery intrpts	disabled
icon name	queue

subqueue (0) Attributes	
Attr Name:	*Attr Value:*
pk capacity	infinite (packets)
bit capacity	infinite (bits)

subqueue (1) Attributes	
Attr Name:	*Attr Value:*
pk capacity	infinite (packets)
bit capacity	infinite (bits)

Streams		
Index	*Input Streams:*	*Output Streams:*
0	src [0]	release [0]

The packet used in the system has the following format:



Packet Format Report: t2	Fri Jan 7 06:56:34 1994	Page 1 of 2
Parameter Model		

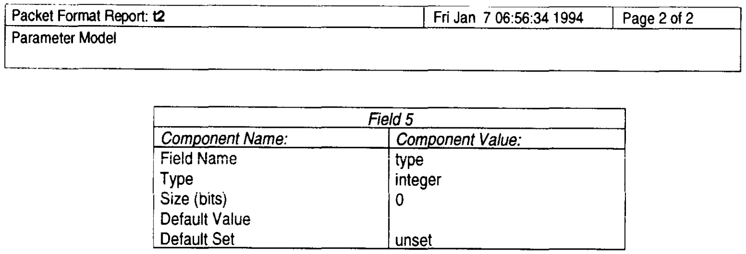
Field 0	
Component Name:	*Component Value:*
Field Name	wavelength
Type	integer
Size (bits)	0
Default Value	
Default Set	unset

Field 1	
Component Name:	*Component Value:*
Field Name	timeslot
Type	integer
Size (bits)	0
Default Value	
Default Set	unset

Field 2	
Component Name:	*Component Value:*
Field Name	src
Type	integer
Size (bits)	0
Default Value	
Default Set	unset

Field 3	
Component Name:	*Component Value:*
Field Name	dst
Type	integer
Size (bits)	0
Default Value	
Default Set	unset

Field 4	
Component Name:	*Component Value:*
Field Name	svc
Type	double
Size (bits)	0
Default Value	
Default Set	unset



Packet Format Report: t2	Fri Jan 7 06:56:34 1994	Page 2 of 2
Parameter Model		

Field 5	
Component Name:	*Component Value:*
Field Name	type
Type	integer
Size (bits)	0
Default Value	
Default Set	unset

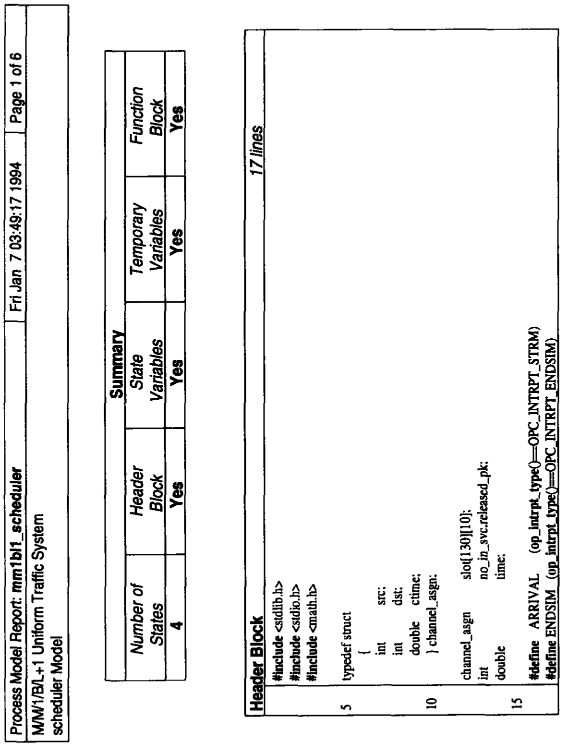
B.3 Single Class, Uniform Traffic System

B.3.1 Blocking System

The termination criterion used for *M/M/1/B/L+1, M/M/1/B/CL,* and *M/M/1/B/RL* systems is to end the program when either 99999 seconds have elapsed or the "steady state" condition has been reached. The program starts to monitor periodically the channel utilization (*cu*), the blocking probability (*pb*), and the blocking probability due to transmitter or receiver conflicts (*pbc*) once the measured load is within 1% of the offered load derived in Section 4.1. If the newly measured *cu, pb,* and *pbc* are within 1% of the previous measured values, we say that "steady state" condition is reached. The program will be ready for termination once the measured load is within 5% of the offered load.

The seed used for each simulation has value *NWL*, where *N* is the number of users, *W* the wavelength channels, and *L* the throughput requirement. So for *N* equals to forty, *W* equals to eight and *L* equals to one, the seed is 4081.

M/M/1/B/L+1: OPNET reports for the *scheduler* and *release* processor models.



Process Model Report: **mm1bl1_scheduler**	Fri Jan 7 03:49:17 1994	Page 1 of 6
M/M/1/B/L+1 Uniform Traffic System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *17 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int      src;
    int      dst;
    double   ctime;
    } channel_asgn;

channel_asgn     slot[130][10];
int              no_in_svc.released_pk;
double           time;

#define   ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define ENDSIM  (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

Process Model Report: **mm1bl1_scheduler**	Fri Jan 7 03:49:17 1994	Page 2 of 6
M/M/1/B/L+1 Uniform Traffic System scheduler Model		

State Variables *30 lines*

```
Objid          \own_id;
Objid          \src id;
int            \node_no;
int            \wavelength_no;
int            \fm_size;
int            \T1;
int            \slot_no;
int            \N;
int            \N1;
double         \svc_time;
char           \arv_arg[20];
double         \arv_rate;
Distribution   *\src_dist;
Distribution   *\dst_dist;
Distribution   *\svc_dist;
long int       \total_pk;
long int       \blocked_pk;
long int       \blocked_c_pk;
double         \cu_max;
double         \cu_prev;
double         \pb_prev;
double         \pbc_prev;
double         \load_prev;
int            \pk_count;
int            \tr_count;
int            \cu_steady;
int            \pb_steady;
int            \pbc_steady;
int            \load_steady;
int            \ready;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j;
double   ratio,temp;
```

Process Model Report: **mm1bl1_scheduler**	Fri Jan 7 03:49:18 1994	Page 3 of 6
M/M/1/B/L+1 Uniform Traffic System scheduler Model		

State 0: init (Enter Execs) *forced, 24 lines*

```
op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)fm_size/(double)(slot_no+1);
T1=floor(svc_time); N1=T1*wavelength_no; svc_time=(double)fm_size/(double)slot_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

total_pk=0; blocked_pk=0; blocked_c_pk=0; released_pk=0; no_in_svc=0; time=0;
cu_max=(double)(N1*slot_no)/(double)N; load_prev=arv_rate/(double)wavelength_no;
cu_steady=0; pb_steady=0; pbc_steady=0; load_steady=0; ready=0;

for (i=0;i<T1;i++)
    for (j=0;j<wavelength_no;j++)
        {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0;}
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 6 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

total_pk+=1; pkptr=op_pk_get(op_intrpt_strm());
```

State 1: pk_prepare (CET's)

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 57 lines*

```
if (!load_steady && released_pk>0)
      {
      temp=total_pk*slot_no*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
      if ( ratio > 0.999 && ratio < 1.001 )
            {
            load_steady=1; ratio=0;
            for (i=0; i<T1; i++)
                  for (j=0; j<wavelength_no; j++) if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
            cu_prev=(time+ratio)*slot_no/(N*op_sim_time());
            pb_prev=(double)blocked_pk/(double)total_pk; pbc_prev=(double)blocked_c_pk/(double)total_pk;
            temp=cu_prev*N1; pk_count=floor(temp); temp=(double)released_pk/(double)pk_count; tr_count=floor(temp)+20;
            }
      }

if ( load_steady && !ready && released_pk>(tr_count*pk_count) )
      {
      if (!cu_steady)
            {
            ratio=0;
            for (i=0; i<T1; i++)
                  for (j=0; j<wavelength_no; j++) if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
            temp=(time+ratio)*slot_no/(N*op_sim_time()); ratio=temp/cu_prev;
            if ( ratio > 0.999 && ratio < 1.001 && time<(total_pk*svc_time) && temp < cu_max ) cu_steady=1;
            else cu_prev=temp;
            }
      if (!pb_steady)
            {
            temp=(double)blocked_pk/(double)total_pk;
            if (pb_prev!=0) ratio=temp/pb_prev;
            else       if (temp==0) ratio=1; else ratio=0;
            if (ratio > 0.999 && ratio < 1.001)            pb_steady=1;
            else pb_prev=temp;
            }
      if (!pbc_steady)
            {
            temp=(double)blocked_c_pk/(double)total_pk;
            if (pbc_prev!=0) ratio=temp/pbc_prev;
            else       if (temp==0) ratio=1; else ratio=0;
            if (ratio > 0.999 && ratio < 1.001)            pbc_steady=1;
            else       pbc_prev=temp;
            }
      if ( cu_steady && pb_steady && pbc_steady ) ready=1;
      if (!ready)
                  {
                  ratio=0;
                  for (i=0; i<T1; i++)
                        for (j=0; j<wavelength_no; j++) if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
                  temp=(time+ratio)*slot_no*N1/(N*op_sim_time()); pk_count=2*floor(temp); tr_count+=1;
                  }
      }

if (ready)
      {
      temp=total_pk*slot_no*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
      if (ratio>0.995&&ratio<1.005&&time<(total_pk*svc_time)&&(time*slot_no)/(N*op_sim_time())<=cu_max&&!ENDSIM)
            op_sim_end("reaching steady state","","","");
      }
```

State 2: idle (CET's)

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#2	*Exec:*	record_stats();
	Trans:	**idle**

State 3: schedule (Enter Execs) *forced, 4 lines*

```
if (no_in_svc == N1)
      {blocked_pk+=1; op_pk_destroy(pkptr);}
else
      if (!find_resource(src,dst,pkptr)) {blocked_pk+=1, blocked_c_pk+=1; op_pk_destroy(pkptr);}
```

State 3: schedule (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Process Model Report: **mm1bl1_release** | Fri Jan 7 03:51:50 1994 | Page 1 of 1

M/M/1/B/L+1 Uniform Traffic System
release Model

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *9 lines*

```
typedef struct
    {
    int        src;
    int        dst;
    double     ctime;
    } channel_asgn;
extern    int              no_in_svc,released_pk;
extern    double           time;
extern    channel_asgn     slot[130][10];
```

Temporary Variables *2 lines*

```
Packet*   pkptr;
int       wavelength,timeslot;
```

State 0: discard (Enter Execs) *unforced, 8 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm());
    op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
    time+=(op_sim_time()-slot[timeslot][wavelength].ctime);
    slot[timeslot][wavelength].src=0; slot[timeslot][wavelength].dst=0; slot[timeslot][wavelength].ctime=0;
    no_in_svc-=1; released_pk+=1; op_pk_destroy(pkptr);
    }
```

State 0: discard (CET's)

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**discard**

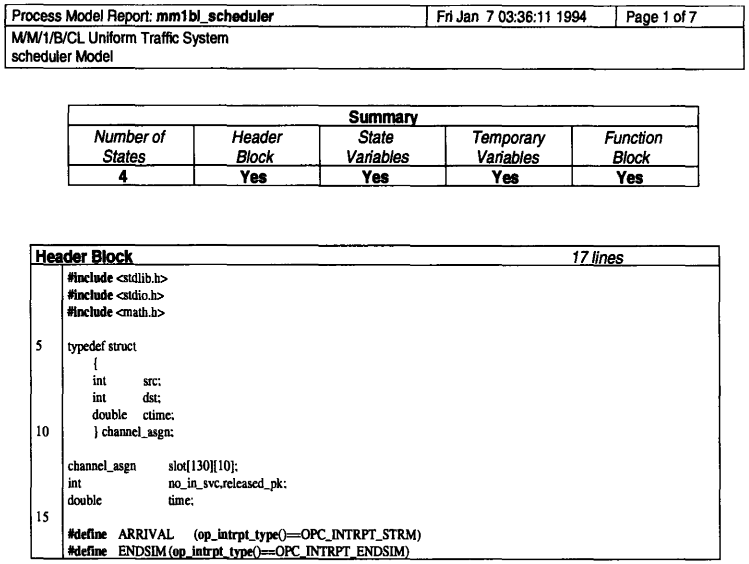
Process Model Report: **mm1bl1_scheduler** | Fri Jan 7 03:49:19 1994 | Page 6 of 6

M/M/1/B/L+1 Uniform Traffic System
scheduler Model

Function Block *51 lines*

```
record_stats()
    {
    int                     i,j;
    double                  cu=0;
    extern double           time;
    extern channel_asgn     slot[130][10];

    for (i=0; i<T1; i++)
        for (j=0; j<wavelength_no; j++) if (slot[i][j].src!=0) time+=(op_sim_time()-slot[i][j].ctime),

    cu=time*slot_no/(N*op_sim_time()); if ( cu > cu_max )            cu=cu_max;
    op_stat_write_scalar("load",total_pk*slot_no*svc_time/(N*op_sim_time()));
    op_stat_write_scalar("cu",cu);
    op_stat_write_scalar("pb",(double)blocked_pk/(double)total_pk);
    op_stat_write_scalar("pbc",(double)blocked_c_pk/(double)total_pk);
    }



int find_resource(sc,dt,packet)
    int        sc,dt;
    Packet*    packet;
    {
    int                     i,j,k,ap;
    double                  svc;
    extern int              no_in_svc;
    extern channel_asgn     slot[130][10];

    ap=0;
   for (i=0;i<T1;i++)                /* for each column */
        {
        k=-1;
        for (j=0;j<wavelength_no;j++)
            {
            if (slot[i][j].src==0 ) k=j;
            if (slot[i][j].src==sc || slot[i][j].dst==dt) goto next_col;
            }
    if (k!=-1)              { ap=1; goto assign; }
        next_col:       ap=0;
        }

    if (!ap) return 0;
    else
        {
        assign:
        slot[i][k].src=sc; slot[i][k] dst=dt; slot[i][k].ctime=op_sim_time();
        op_pk_nfd_set(packet,"wavelength",k); op_pk_nfd_set(packet,"timeslot",i),
        svc=op_dist_outcome(svc_dist); no_in_svc+=1; op_pk_send_delayed(packet,0,svc);
        return 1;
        }
    }
```

M/M/1/B/CL: OPNET report for the *scheduler* processor model. The *release* model is identical to that in M/M/1//B/C+1.



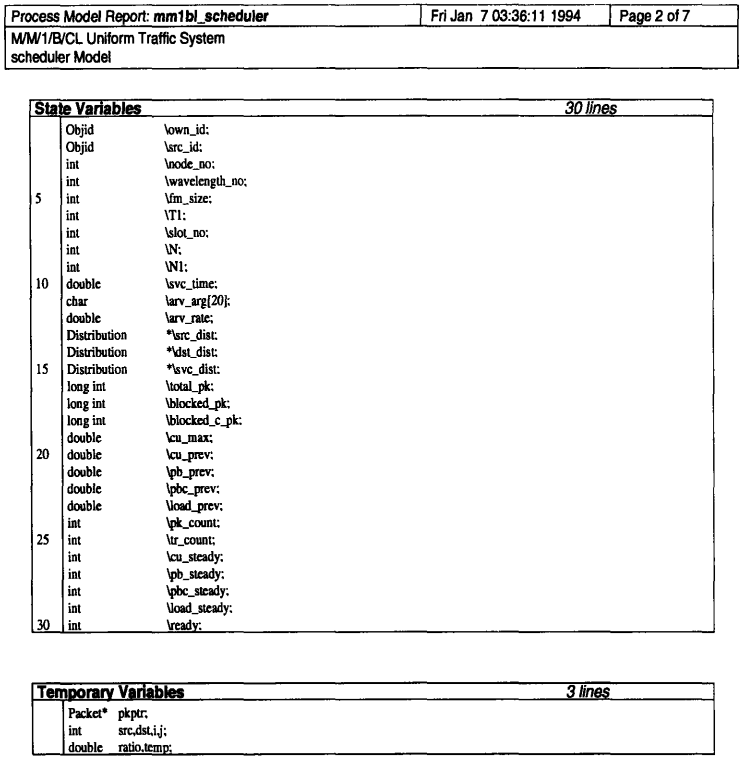
Process Model Report: **mm1bl_scheduler**	Fri Jan 7 03:36:11 1994	Page 1 of 7
M/M/1/B/CL Uniform Traffic System scheduler Model		

Summary

Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *17 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int       src;
    int       dst;
    double    ctime;
    } channel_asgn;

channel_asgn        slot[130][10];
int                 no_in_svc,released_pk;
double              time;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define  ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```



Process Model Report: **mm1bl_scheduler**	Fri Jan 7 03:36:11 1994	Page 2 of 7
M/M/1/B/CL Uniform Traffic System scheduler Model		

State Variables *30 lines*

```
Objid           \own_id;
Objid           \src_id;
int             \node_no;
int             \wavelength_no;
int             \fm_size;
int             \T1;
int             \slot_no;
int             \N;
int             \N1;
double          \svc_time;
char            \arv_arg[20];
double          \arv_rate;
Distribution    *\src_dist;
Distribution    *\dst_dist;
Distribution    *\svc_dist;
long int        \total_pk;
long int        \blocked_pk;
long int        \blocked_c_pk;
double          \cu_max;
double          \cu_prev;
double          \pb_prev;
double          \pbc_prev;
double          \load_prev;
int             \pk_count;
int             \tr_count;
int             \cu_steady;
int             \pb_steady;
int             \pbc_steady;
int             \load_steady;
int             \ready;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j;
double   ratio,temp;
```

State 0: init (Enter Execs) *forced, 24 lines*

```
op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no).
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)fm_size/(double)slot_no;
T1=floor(svc_time); N1=T1*wavelength_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

total_pk=0; blocked_pk=0; blocked_c_pk=0; released_pk=0; no_in_svc=0; time=0;
cu_max=(double)(N1*slot_no)/(double)N; load_prev=arv_rate/(double)wavelength_no;
cu_steady=0; pb_steady=0; pbc_steady=0; load_steady=0; ready=0;

for (i=0;i<T1;i++)
     for (j=0;j<wavelength_no;j++)
          {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0; }
```

State 0: init (CET's)

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 6 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

total_pk+=1; pkptr=op_pk_get(op_intrpt_strm());
```

State 1: pk_prepare (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 60 lines*

```
if (!load_steady && released_pk>0)
     {
     temp=total_pk*slot_no*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
     if ( ratio > 0.999 && ratio < 1.001 )
          {
          load_steady=1; ratio=0;
          for (i=0; i<T1; i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
          cu_prev=(time+ratio)*slot_no/(N*op_sim_time());
          pb_prev=(double)blocked_pk/(double)total_pk; pbc_prev=(double)blocked_c_pk/(double)total_pk;
          temp=cu_prev*N1; pk_count=floor(temp); temp=(double)released_pk/(double)pk_count; tr_count=floor(temp)+20;
          }
     }

if ( load_steady && !ready && released_pk>(tr_count*pk_count) )
     {
     if (!cu_steady)
          {
          ratio=0;
          for (i=0; i<T1; i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
          temp=(time+ratio)*slot_no/(N*op_sim_time()); ratio=temp/cu_prev;
          if ( ratio > 0.999 && ratio < 1.001 && time<(total_pk*svc_time) && temp < cu_max ) cu_steady=1;
          else cu_prev=temp;
          }
     if (!pb_steady)
          {
          temp=(double)blocked_pk/(double)total_pk;
          if (pb_prev!=0) ratio=temp/pb_prev;
          else     if (temp==0) ratio=1; else ratio=0;
          if (ratio > 0.999 && ratio < 1.001)          pb_steady=1;
          else     pb_prev=temp;
          }
     if (!pbc_steady)
          {
          temp=(double)blocked_c_pk/(double)total_pk;
          if (pbc_prev!=0) ratio=temp/pbc_prev;
          else     if (temp==0) ratio=1; else ratio=0;
          if (ratio > 0.999 && ratio < 1.001) pbc_steady=1;
          else     pbc_prev=temp;
          }
     if ( cu_steady && pb_steady && pbc_steady ) ready=1.
     if (!ready)
               {
               ratio=0;
               for (i=0; i<T1; i++)
                    for (j=0; j<wavelength_no; j++)
                         if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
               temp=(time+ratio)*slot_no*N1/(N*op_sim_time()); pk_count=2*floor(temp); tr_count+=1;
               }
     }

if (ready)
     {
     temp=total_pk*slot_no*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
     if (ratio>0.995&&ratio<1.005&&time<(total_pk*svc_time)&&(time*slot_no)/(N*op_sim_time())<=cu_max&&!ENDSIM
```

M/M/1/B/CL Uniform Traffic System
scheduler Model

```
          op_sim_end("reaching steady state","","");
     }
```

State 2:		**idle (CET's)**
CET	*Cond:*	(default)
#0	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#1	*Exec:*	record_stats();
	Trans:	**idle**
CET	*Cond:*	(ARRIVAL)
#2	*Exec:*	;
	Trans:	**pk_prepare**

State 3: **schedule (Enter Execs)** *forced, 5 lines*

```
if (no_in_svc == N1)
     {blocked_pk+=1; op_pk_destroy(pkptr);}
else
     if (!find_resource(src,dst,pkptr))
     {blocked_pk+=1; blocked_c_pk+=1; op_pk_destroy(pkptr);}
```

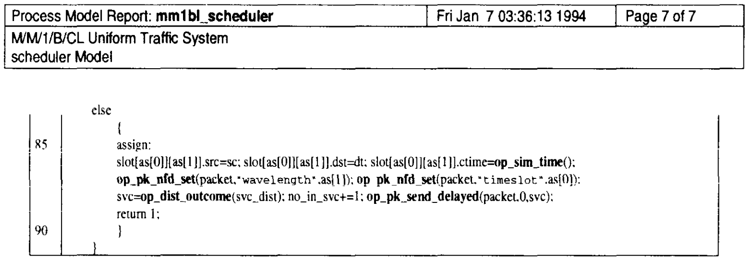
State 3:		**schedule (CET's)**
CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *91 lines*

```
record_stats()
     {
     int                    i,j;
     double                 cu;
     extern double          time;
     extern channel_asgn    slot[130][10];

     for (i=0; i<T1; i++)
          for (j=0; j<wavelength_no; j++)
               if (slot[i][j].src!=0) time+=(op_sim_time()-slot[i][j].ctime);
     cu=time*slot_no/(N*op_sim_time()); if ( cu > cu_max ) cu=cu_max;
     op_stat_write_scalar("load",total_pk*slot_no*svc_time/(N*op_sim_time()));
     op_stat_write_scalar("cu",cu);
     op_stat_write_scalar("pb",(double)blocked_pk/(double)total_pk);
     op_stat_write_scalar("pbc",(double)blocked_c_pk/(double)total_pk);
     }



int find_resource(sc,dt,packet)
     int       sc,dt;
     Packet*   packet;
     {
```

M/M/1/B/CL Uniform Traffic System
scheduler Model

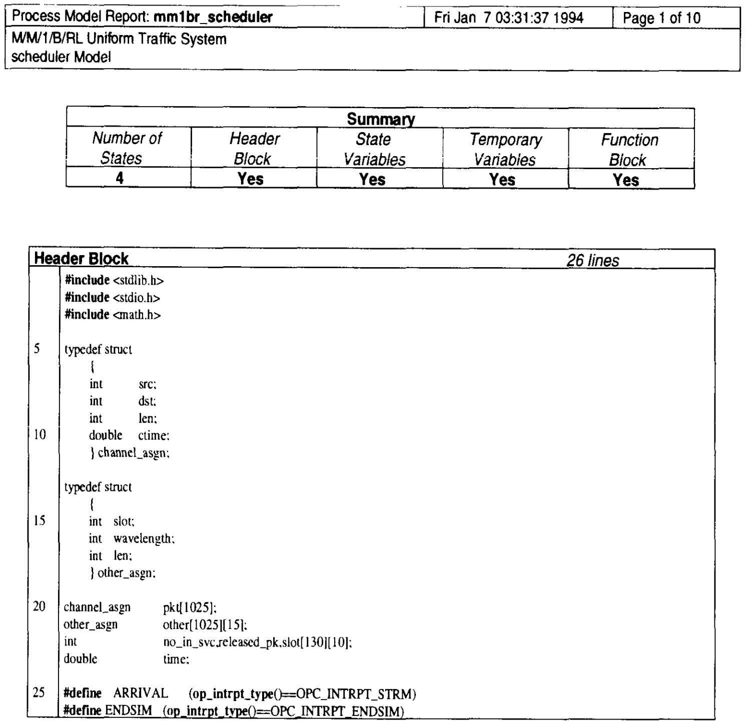
```
     int                    i,j,ap,fit,ps,pd,cf,cs,cd,nf,ns,nd,ca[8],na[8],as[2];
     double                 svc;
     extern int             no_in_svc;
     extern channel_asgn    slot[130][10];

     ap=-1; ps=-1; pd=-1;
     for (i=0;i<T1;i++)            /* for each column */
          {
          if (i==0)
               {
               cf=0; cs=-1; cd=-1;
               for (j=0;j<wavelength_no;j++)
                    {
                    if (slot[i][j].src==0) { ca[cf]=j; cf+=1; }
                    if (slot[i][j].src==sc) cs=j;
                    if (slot[i][j].dst==dt) cd=j;
                    }
               }

          if (i==(T1-1)) { ns=-1; nd=-1; }
          else
               {
               nf=0; ns=-1; nd=-1;
               for (j=0;j<wavelength_no;j++)
                    {
                    if (slot[i+1][j].src==0) { na[nf]=j; nf+=1; }
                    if (slot[i+1][j].src==sc) ns=j;
                    if (slot[i+1][j].dst==dt) nd=j;
                    }
               }

          if (cf==0 || cs!=-1 || cd!=-1) goto next_col;

          for (j=0;j<cf;j++)
               {
               fit=0;
               if (ps!=-1)
                    if (ca[j]==ps) fit+=1; else continue;
               if (pd!=-1)
                    if (ca[j]==pd) fit+=1; else continue;
               if (ns!=-1)
                    if (ca[j]==ns) fit+=1; else continue;
               if (nd!=-1)
                    if (ca[j]==nd) fit+=1; else continue;

               if (fit>ap)
                    { as[0]=i; as[1]=ca[j]; ap=fit; }
               if (ap==4) goto assign;
               }
          next_col:
          if (i!=(T1-1))
               {
               ps=cs; pd=cd; cs=ns; cd=nd; cf=nf;
               for (j=0;j<nf;j++) ca[j]=na[j];
          }
          }

     if (ap==-1)
          return 0;
```



Process Model Report: **mm1bl_scheduler**	Fri Jan 7 03:36:13 1994	Page 7 of 7
M/M/1/B/CL Uniform Traffic System scheduler Model		

```
    else
        {
        assign:
        slot[as[0]][as[1]].src=sc; slot[as[0]][as[1]].dst=dt; slot[as[0]][as[1]].ctime=op_sim_time();
        op_pk_nfd_set(packet,"wavelength",as[1]); op pk_nfd_set(packet,"timeslot",as[0]);
        svc=op_dist_outcome(svc_dist); no_in_svc+=1; op_pk_send_delayed(packet,0,svc);
        return 1;
        }
    }
```

M/M/1/B/RL: OPNET reports for the *scheduler* and *release* processor models.



Process Model Report: **mm1br_scheduler**	Fri Jan 7 03:31:37 1994	Page 1 of 10
M/M/1/B/RL Uniform Traffic System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *26 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int         src;
    int         dst;
    int         len;
    double      ctime;
    } channel_asgn;

typedef struct
    {
    int   slot;
    int   wavelength;
    int   len;
    } other_asgn;

channel_asgn        pkt[1025];
other_asgn          other[1025][15];
int                 no_in_svc,released_pk,slot[130][10];
double              time;

#define   ARRIVAL     (op_intrpt_type()==OPC_INTRPT_STRM)
#define ENDSIM   (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

Process Model Report: ***mm1br_scheduler***	*Fri Jan 7 03:31:37 1994*	*Page 2 of 10*
M/M/1/B/RL Uniform Traffic System scheduler Model		

State Variables *31 lines*

```
Objid          \own_id;
Objid          \src_id;
int            \node_no;
int            \wavelength_no;
int            \fm_size;
int            \T1;
int            \slot_no;
int            \N;
int            \N1;
double         \svc_time;
char           \arv_arg[20];
double         \arv_rate;
Distribution   *\src_dist;
Distribution   *\dst_dist;
Distribution   *\svc_dist;
long int       \total_pk;
long int       \blocked_pk;
long int       \blocked_c_pk;
double         \cu_max;
double         \cu_prev;
double         \pb_prev;
double         \pbc_prev;
double         \load_prev;
double         \total_time;
int            \pk_count;
int            \tr_count;
int            \cu_steady;
int            \pb_steady;
int            \pbc_steady;
int            \load_steady;
int            \ready;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j;
double   ratio,temp;
```

Process Model Report: ***mm1br_scheduler***	*Fri Jan 7 03:31:38 1994*	*Page 3 of 10*
M/M/1/B/RL Uniform Traffic System scheduler Model		

State 0: init (Enter Execs) *forced, 29 lines*

```
op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)N/(double)slot_no;
N1=floor(svc_time); svc_time=(double)fm_size/(double)slot_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

total_pk=0; blocked_pk=0; blocked_c_pk=0; released_pk=0; no_in_svc=0; time=0;
cu_max=(double)(N1*slot_no)/(double)N; load_prev=arv_rate/(double)wavelength_no;
cu_steady=0; pb_steady=0; pbc_steady=0; load_steady=0; ready=0;

for (i=0;i<=N1;i++)
    {
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0;
    for (j=0; j<15; j++)
        {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;}
    }
for (i=0;i<fm_size;i++)
    for (j=0;j<wavelength_no;j++) slot[i][j]=0;
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 10 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;
temp=op_dist_outcome(svc_dist);

pkptr=op_pk_get(op_intrpt_strm());
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);

total_pk+=1; total_time+=temp;
```

State 1: pk_prepare (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 56 lines*

```
if (!load_steady && released_pk>0)
    {
    temp=slot_no*total_time/(N*op_sim_time()); ratio=temp/load_prev;
    if ( ratio > 0.999 && ratio < 1.001 )
        {
        load_steady=1, ratio=0,
        for (i=1; i<=N1; i++)
            if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
        cu_prev=(time+ratio)*slot_no/(N*op_sim_time());
        pb_prev=(double)blocked_pk/(double)total_pk; pbc_prev=(double)blocked_c_pk/(double)total_pk;
        temp=cu_prev*N1; pk_count=floor(temp); temp=(double)released_pk/(double)pk_count; tr_count=floor(temp)+20;
        }
    }

if ( load_steady && !ready && released_pk>(tr_count*pk_count) )
    {
    if (!cu_steady)
        {
        ratio=0;
        for (i=1; i<=N1; i++)
            if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
        temp=(time+ratio)*slot_no/(N*op_sim_time()); ratio=temp/cu_prev;
        if ( ratio > 0.999 && ratio < 1.001 && time<(total_pk*svc_time) && temp < cu_max ) cu_steady=1;
        else        cu_prev=temp;
        }
    if (!pb_steady)
        {
        temp=(double)blocked_pk/(double)total_pk;
        if (pb_prev!=0) ratio=temp/pb_prev;
        else        if (temp==0) ratio=1; else ratio=0;
        if (ratio > 0.999 && ratio < 1.001)             pb_steady=1; else pb_prev=temp;
        }
    if (!pbc_steady)
        {
        temp=(double)blocked_c_pk/(double)total_pk;
        if (pbc_prev!=0) ratio=temp/pbc_prev;
        else        if (temp==0) ratio=1; else ratio=0;
        if (ratio > 0.999 && ratio < 1.001)             pbc_steady=1;
        else        pbc_prev=temp;
        }
    if ( cu_steady && pb_steady && pbc_steady ) ready=1;
    if (!ready)
            {
            ratio=0;
            for (i=1; i<=N1; i++)
                if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
            temp=(time+ratio)*slot_no*N1/(N*op_sim_time()); pk_count=2*floor(temp); tr_count+=1,
            }
    }

if (ready)
    {
    temp=total_pk*slot_no*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
    if (ratio>0.995&&ratio<1.005&&time<(total_pk*svc_time)&&(time*slot_no)/(N*op_sim_time())<=cu_max&&!ENDSIM)
        op_sim_end("reaching steady state","","","");
    }
```

State 2: **idle (CET's)**

```
CET  Cond:   (ARRIVAL)
#0   Exec:   ;
     Trans:  pk_prepare
CET  Cond:   (default)
#1   Exec:   ;
     Trans:  idle
CET  Cond:   (ENDSIM)
#2   Exec:   record_stats();
     Trans:  idle
```

State 3: **schedule (Enter Execs)** *forced, 4 lines*

```
if (no_in_svc == N1)
     {blocked_pk+=1; op_pk_destroy(pkptr);}
else
     if (!find_resource(pkptr)) {blocked_pk+=1; blocked_c_pk+=1; op_pk_destroy(pkptr);}
```

State 3: **schedule (CET's)**

```
CET  Cond:   (1)
#0   Exec:   ;
     Trans:  idle
```

Function Block *230 lines*

```
record_stats()
     {
     int                  i,j;
     double               cu=0;
     extern double        time;
     extern channel_asgn  pkt[1025];

     for (i=1; i<=N1; i++)
          if (pkt[i].src!=0) time+=(op_sim_time()-pkt[i].ctime);

     cu=time*slot_no/(N*op_sim_time());
     if ( cu > cu_max )   cu=cu_max;
     op_stat_write_scalar("load",slot_no*total_time/(N*op_sim_time()));
     op_stat_write_scalar("cu",cu);
     op_stat_write_scalar("pb",(double)blocked_pk/(double)total_pk);
     op_stat_write_scalar("pbc",(double)blocked_c_pk/(double)total_pk);
     }


int find_resource(packet)
     Packet*   packet;
     {
     int                  i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next,sc,dt;
     int                  conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
     double               svc;
     extern int           no_in_svc,slot[130][10];
     extern other_asgn    other[1025][15];
     extern channel_asgn  pkt[1025];
```

```
op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst",&dt); op_pk_nfd_get(packet,"svc",&svc);

ct=0;
for (j=0, j<wavelength_no; j++) empty[j]=0;

for (i=0;i<fm_size;i++)
     {

     conflict[i][0]=-1; conflict[i][1]=-1;
     for (k=0; k<wavelength_no; k++)
          {
          if ( slot[i][k]==0 ) empty[k]+=1;
          if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
          if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
          }

     go_back=0;
     if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
          {
          for (k=0; k<wavelength_no; k++)
               {
               if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
                    {
                    if ( slot[i][k]!=0 ) j=i-empty[k]-1;
                    else      j=i-empty[k];
                    if ( j<0 ) oh=0;
                    else
                         if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                         else oh=2;
                    if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
                    insert:
                    if ( ct==0 )
                         {
                         if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k]; else store[0][0]=i-empty[k]+1;
                         store[0][1]=k;      store[0][2]=empty[k];      store[0][3]=oh;
                         }
                    else
                         for (j=(ct-1); j>=0; j--)
                              {
                              if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
                                   {
                                   store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
                                   store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
                                   }
                              if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
                                   {
                                   store_ct:
                                   if ( slot[i][k]!=0 || go_back ) store[j+1][0]=i-empty[k];
                                   else      store[j+1][0]=i-empty[k]+1;
                                   store[j+1][1]=k;      store[j+1][2]=empty[k];      store[j+1][3]=oh;
                                   break;
                                   }
                              if ( j==0 ) { j=-1; goto store_ct; }
                              }
                    if ( go_back ) goto back;
                    empty[k]=0; ct+=1;
                    }
               }
          }
```

```
    else
        {
        for ( k=0; k<wavelength no; k++)
            {
            if ( empty[k]>0 )
                {
                if ( slot[i][k]==0 ) empty[k]-=1;
                if ( empty[k]==0) continue;
                j=i-empty[k]-1;
                if ( j<0 ) oh=0;
                else
                    if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                    else oh=2;
                if ((conflict[i][0]==-1 || conflict[i][0]==k)&&(conflict[i][1]==-1 || conflict[i][1]==k)) oh=oh;
                else        if ( oh==0 ) oh=1; else oh=3;
                if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0; continue; }
                go_back=1;
                goto insert;
                back:
                empty[k]=0; ct+=1;
                }
            }
        }
    }

nd=slot_no; i_ct=0; waste=0; next=0; asgn[0]=-1; asgn[1]=0;
for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
for (i=0; i<ct; i++)
    {
    if ( store[i][3]==0 ) fit = 0;
    else if ( store[i][3]==3 ) fit = 2; else                    fit = 1;
    j=store[i][2]-fit;

    if (next)
        {
        if ( j>=nd )
            {
            k=j-nd+fit;
            if ( fit<= ap1 && k<ap )
                {
                for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                    if ( rmb[oh] ) { k=-1; break; }
                if ( k!=-1 )
                    {
                    ap=k, ap1=fit, temp[i_ct]=i,
                    for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                    k=store[i][0]-1; if ( k<0 ) k=0;
                    go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                    for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1.
                    }
                }
            }
        if ( j<nd || i==(ct-1) )
            {
            waste+=ap1; i_ct+=1;
            if ( waste<asgn[0] || asgn[0]==-1 )
                {
                asgn[0]=waste; asgn[1]=i_ct;
                for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
```

```
                if ( asgn[0]==0 ) goto assign;
                }
            if ( j<nd )
                {
                temp[0]=i; nd=slot_no-j; waste=fit; i_ct=1; next=0;
                for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                k=store[i][0]-1; if ( k<0 ) k=0;
                go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                }
            }
        continue;
        }

    k=0;
    for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
        if ( rmb[oh] ) { k=1; break; }
    if ( k==1 ) continue;
    else
        {
        temp[i_ct]=i; k=store[i][0]-1; if ( k<0 ) k=0;
        go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
        for (oh=k; oh<go_back; oh++)
            if ( j>nd ) tmp_rmb[oh]=1; else rmb[oh]=1;
        }

    if ( j<nd ) { nd-=j; waste+=fit; i_ct+=1; continue; }
    if ( j==nd )
        {
        waste+=fit; i_ct+=1;
        if ( waste<asgn[0] || asgn[0]==-1 )
            {
            asgn[0]=waste; asgn[1]=i_ct;
            for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
            if ( asgn[0]==0 ) goto assign;
            }
        nd=slot_no; i_ct=0; waste=0;
        for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
        continue;
        }
    if ( j>nd ) { ap=j-nd+fit; ap1=fit; next=1; }
    if ( next && i==(ct-1) )
        {
        waste+=ap1; i_ct+=1;
        if ( waste<asgn[0] || asgn[0]==-1 )
            {
            asgn[0]=waste; asgn[1]=i_ct;
            for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
            if ( asgn[0]==0 ) goto assign;
            }
        }
    }

if(asgn[0]==-1)
    return 0;
else
    {
    assign:
    for (k=1; k<=N1; k++)
```

Process Model Report: **mm1br_scheduler**	Fri Jan 7 03:31:40 1994	Page 10 of 10
M/M/1/B/RL Uniform Traffic System scheduler Model		

```
                {
                if (pkt[k].src!=0) continue;
                pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1]; pkt[k].ctime=op_sim_time(); nd=slot_no;
                for (i=2; i<=asgn[1]; i++)
                     {
                     oh=store[asgn[i]][0]; ct=store[asgn[i]][2]; nd-=ct;
                     if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                     else      if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                               else      if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                     for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                     other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
                     }
                i=asgn[1]+1; oh=store[asgn[i]][0];
                if ( store[asgn[i]][3]==3 ) oh+=1;
                else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
                for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
                other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
                no_in_svc+=1;
                op_pk_nfd_set(packet,"timeslot",oh); op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
                op_pk_send_delayed(packet,0,svc); break;
                }
           return 1;
           }
      }
```

Process Model Report: **mm1br_release**	Fri Jan 7 03:33:21 1994	Page 1 of 1
M/M/1/B/RL Uniform Traffic System release Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *19 lines*

```
typedef struct
     {
     int       src;
     int       dst;
     int       len;
     double    ctime;
     } channel_asgn;

typedef struct
     {
     int       slot;
     int       wavelength;
     int       len;
     } other_asgn;

extern    int             no_in_svc,slot[130][10],released_pk;
extern    double          time;
extern    channel_asgn    pkt[1025];
extern    other_asgn      other[1025][15];
```

Temporary Variables *2 lines*

```
Packet*   pkptr;
int       wavelength,timeslot,i,j,k;
```

State 0: discard (Enter Execs) *unforced, 12 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
     {
     pkptr=op_pk_get(op_intrpt_strm());
     op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
     i=slot[timeslot][wavelength]; time+=(op_sim_time()-pkt[i].ctime);
     for (j=0; j<pkt[i].len; j++)
          {
          for (k=other[i][j].slot; k<(other[i][j].slot+other[i][j].len); k++) slot[k][other[i][j].wavelength]=0;
          other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;
          }
     pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0; no_in_svc-=1; released_pk+=1; op_pk_destroy(pkptr);
     }
```

State 0: discard (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**discard**

B.3.2 Queueing System

Many methods can be used to determine the stopping time for a simulation [8].

Regenerative method [9] was used as the termination criterion for *M/M/1/Q/L+1* system. A cycle starts when the queue changes its size from zero to one. Average queue size is measured in each of the cycle, and its half length of the 90% confidence interval is calculated. The program terminates when one of the three conditions are true: the half length interval of the average queue size for 90% confidence interval reaches within 10% value of the average queue size, or the number of the cycles gone through by the simulation reaches 20,000, or the simulation has been running for 2,000,000 seconds. The simulation will also end if it is determined that its cycle time is extremely long, that is it went through less than two cycles in 99,999 seconds. The procedure outlined in [10] is used to calculate the inverse of the cdf of the normal distribution, which seems to have a relative accuracy of about 5 decimal places.

For *M/M/1/Q/CL* and *M/M/1/Q/RL* systems, a sequential batch means procedure outlined in [11] is used to determine the run length of the simulation. The following is the description of the procedure:

(1) Set $i \leftarrow 1$, $n_0 \leftarrow 600$, $n_1 \leftarrow 800$. Collect n_1 observations.
(2) Partition the n_i observations into 400 batches, each of size $n_i/400$. If the estimated serial correlation in these batches is greater than 0.4, go to (5). If it is negative, go to (4).
(3) Partition the n_i observations into 200 batches, each of size $n_i/200$. If the estimated serial correlation among these 200 batches is greater than that among the 400 batches, go to (5).
(4) Partition the n_i observations into 40 batches, each of size $n_i/40$. Construct a *nominal* p-percent confidence interval assuming the 40 batches are independent and ignoring that the batches were constructed sequentially. If the interval is acceptably small relative to the current value of $\bar{X}$ (say half-width/$\bar{X} < \gamma$), stop; otherwise go to (5).
(5) Set $i \leftarrow i+1$, $n_i \leftarrow 2n_{i-2}$. Collect $n_i - n_{i-1}$ additional observations and go to (2).

In our simulations, we used the 90% confidence interval, and the termination criteria is γ from step (4) equals to 0.1. Also the observation was made on the average queue size after the measured load comes to within 1% of the offered load obtained in Section 4.1. Again the simulation is terminated either when the above criteria is met or after it has been running for 2,000,000 seconds.

The seed used for each simulation has value *NWL*, where *N* is the number of users, *W* the wavelength channels, and *L* the throughput requirement. So for *N* equals to forty, *W* equals to eight and *L* equals to one, the seed is 4081.

M/M/1/Q/L+1: Opnet reports for the *scheduler* and *release* processor models.

Process Model Report: **mm1ql1_scheduler**	Fri Jan 7 05:10:45 1994	Page 1 of 7

M/M/1/Q/L+1 Uniform Traffic System
scheduler Model

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *26 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int       src;
    int       dst;
    double    ctime;
    } channel_asgn;

typedef struct
    {
    double    a;
    double    q;
    } cycle_stat;

channel_asgn       slot[130][10];
int                no_in_svc;
double             time;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define ENDSIM   (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define EMPTYQ  (op_subq_empty(0))
#define NEXTQ    (op_intrpt_type()==OPC_INTRPT_REMOTE && \
                  !EMPTYQ)
```

Process Model Report: **mm1ql1_scheduler**	Fri Jan 7 05:10:45 1994	Page 2 of 7

M/M/1/Q/L+1 Uniform Traffic System
scheduler Model

State Variables *33 lines*

```
Objid           \own_id;
Objid           \src_id;
int             \node_no;
int             \wavelength_no;
int             \fm_size;
int             \T1;
int             \slot_no;
int             \N;
int             \N1;
double          \svc_time;
char            \arv_arg[20];
double          \arv_rate;
Distribution    *\src_dist;
Distribution    *\dst_dist;
Distribution    *\svc_dist;
cycle_stat      \cstat[20000];
long int        \total_pk;
long int        \processed_pk;
long int        \cycle_ct;
long int        \next_cycle;
double          \total_time;
double          \load_prev;
double          \delay;
double          \alpha;
double          \queue;
double          \temp_alpha;
double          \temp_queue;
double          \tstart;
double          \qsize;
double          \z;
int             \cycle_inc;
int             \end_sim;
int             \load_steady;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j,k;
double   a1,a2,a3,a4,a5,r,s11,s12,s22,s,temp,ratio;
```

Process Model Report: **mm1ql1_scheduler**	Fri Jan 7 05:10:46 1994	Page 3 of 7
M/M/1/Q/L+1 Uniform Traffic System scheduler Model		

State 0: init (Enter Execs) *forced, 27 lines*

```
f_inverse(0.1,&z);        /* 90% confidence interval */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)fm_size/(double)(slot_no+1); T1=floor(svc_time);
N1=T1*wavelength_no; svc_time=(double)fm_size/(double)slot_no; load_prev=arv_rate/(double)wavelength_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

total_pk=0; processed_pk=0; no_in_svc=0; cycle_ct=0; end_sim=0; delay=0; time=0; total_time=0;
load_steady=0; alpha=0; queue=0; temp_alpha=0; temp_queue=0; tstart=0; op_subq_flush(0);

for (i=0;i<T1;i++)
      for (j=0;j<wavelength_no;j++)
            {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0;}
for (i=0;i<20000;i++)
      {cstat[i].a=0; cstat[i].q=0;}
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 10 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;
temp=op_dist_outcome(svc_dist);

pkptr=op_pk_get(op_intrpt_strm());
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);

total_pk+=1; total_time+=temp;
```

State 1: pk_prepare (CET's)

CET #0	*Cond:*	(1)
	Exec:	;

Process Model Report: **mm1ql1_scheduler**	Fri Jan 7 05:10:46 1994	Page 4 of 7
M/M/1/Q/L+1 Uniform Traffic System scheduler Model		

	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 30 lines*

```
if (!load_steady) { next_cycle=1000; cycle_inc=1000;}

if ( cycle_ct>next_cycle && load_steady)
      {
      a1=0; a2=0; a3=0; a4=0; a5=0;
      for ( j=0; j<cycle_ct && j<20000; j++ )
            {a1+=cstat[j].a; a2+=cstat[j].q; a3+=cstat[j].a*cstat[j].a; a4+=cstat[j].q*cstat[j].q; a5+=cstat[j].a*cstat[j].q;}
      r=a2/a1; j=cycle_ct; s11=(a4-a2*a2/j)/(j-1); s22=(a3-a1*a1/j)/(j-1); s12=(a5-a1*a2/j)/(j-1);
      s=s11-2*r*s12+r*r*s22; s=z*sqrt(s*j)/a1;
      if (s==0.0) end_sim=1;
      else
            {
            a1=s/(0.1*r);
            if ( a1<1.0 ) end_sim=1;
            else
                  {
                  a2=j*a1*a1; j=floor(a2)-j;
                  if ( j>cycle_inc ) next_cycle+=cycle_inc; else next_cycle+=j;
                  }
            }
      }

if ( end_sim && !ENDSIM )
      op_sim_end("reaching steady state","","","");

if (cycle_ct==20000 && !ENDSIM)
      op_sim_end("20000 cycle","","","").

if ( op_sim_time()>99999 && cycle_ct<2 && !ENDSIM )
      op_sim_end("cycle too long","","","");
```

State 2: idle (CET's)

CET #0	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**
CET #1	*Cond:*	(ENDSIM)
	Exec:	record_stats();
	Trans:	**idle**
CET #2	*Cond:*	(NEXTQ && !ARRIVAL)
	Exec:	;
	Trans:	**schedule**
CET #3	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**

Function Block *87 lines*

```
record_stats()
    {
    int             i,j,k;
    double          a=0,a1,a2,a3,a4,a5,s=0,s11,s12,s22,r,temp;
    extern double           time;
    extern channel_asgn     slot[130][10];

    a1=0; a2=0; a3=0; a4=0; a5=0;
    for ( j=0; j<cycle_ct && j<20000; j++ )
        {
        a1+=(double)cstat[j].a; a2+=(double)cstat[j].q; a3+=(double)cstat[j].a*(double)cstat[j].a;
        a4+=(double)cstat[j].q*(double)cstat[j].q; a5+=(double)cstat[j].a*(double)cstat[j].q;
        }
    if (cycle_ct==0) { a1=alpha+temp_alpha; a2=queue+temp_queue; }
    if (a2==0.0 && a1==0.0) a1=op_sim_time();
    r=a2/a1;
    if (cycle_ct>1)
        {
        a=a1/cycle_ct; j=cycle_ct; s11=(a4-a2*a2/j)/(j-1); s22=(a3-a1*a1/j)/(j-1); s12=(a5-a1*a2/j)/(j-1);
        s=s11-2*r*s12+r*r*s22; s=z*sqrt(s*j)/a1;
        }

    for (i=0; i<T1; i++)
        for (j=0; j<wavelength_no; j++)
            if (slot[i][j].src!=0) time+=(op_sim_time()-slot[i][j].ctime);

    op_stat_write_scalar("load",total_time*slot_no/(N*op_sim_time()));
    op_stat_write_scalar("cu",time*slot_no/(N*op_sim_time()));
    op_stat_write_scalar("Q",r);
    op_stat_write_scalar("qdev",s);
    op_stat_write_scalar("D",delay/(double)processed_pk);
    op_stat_write_scalar("C1",a);
    op_stat_write_scalar("Cc",cycle_ct);
    }



f_inverse (x,z1)
    double    x,*z1;
    {
    double    y,p,p0,p1,p2,p3,p4,q0,q1,q2,q3,q4;

    x=1-x/2;
    p0=-0.322232431088e0; p1=-1.0; p2=-0.342242088547e0; p3=-0.0204231210245e0; p4=-0.453642210148e-4;
    q0=0.099348462606e0; q1=0.588581570495e0; q2=0.531103462366e0; q3=0.10353775285e0; q4=0.38560700634e-2;

    p=x;
    if (p>0.5) p=1.0-p;
    y=sqrt(-log(p*p));
    *z1=y+(p0+y*(p1+y*(p2+y*(p3+y*p4))))/(q0+y*(q1+y*(q2+y*(q3+y*q4))));
    if (x<0.5) *z1=-*z1;
    }


int find_resource(packet)
    Packet*   packet;
    {
    int                         i,j,k,ap,sc,dt;
```

State 3: **schedule (Enter Execs)** *forced, 50 lines*

```
if ( ARRIVAL )
    {
    it (no_in_svc == N1)
        {
      if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
            { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
        else
            {
            ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
            if (ratio==1.0)
                {
                if (load_steady) {alpha+=(temp-tstart); cstat[cycle_ct].a=alpha; cstat[cycle_ct].q=queue, cycle_ct+=1,}
                if (!load_steady) { temp_alpha=alpha; temp_queue=queue; }
                tstart=temp; qsize=1; alpha=0; queue=0; load_steady=1;
                }
            else
                {
                alpha+=(temp-tstart); queue+=qsize*(temp-tstart); tstart=temp; qsize=ratio;
                if (tstart==0.0) printf("error1");
                }
            }
        }
    else
        if (!find_resource(pkptr))
            {
          if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
            else
                {
                ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
                alpha+=(temp-tstart); queue+=qsize*(temp-tstart); tstart=temp; qsize=ratio;
                }
            }
    }

if ( NEXTQ )
        {
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if (no_in_svc < N1)
                {
                pkptr=op_subq_pk_remove(0,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                }
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize && load_steady)
            {alpha+=(temp-tstart); queue+=qsize*(temp-tstart); tstart=temp; qsize=ratio;}
        }
```

State 3:		**schedule (CET's)**
CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**idle**

Process Model Report: **mm1ql1_release**	Fri Jan 7 05:13:51 1994	Page 1 of 1
M/M/1/Q/L+1 Uniform Traffic System release Model		

Summary

Number of States	Header Block	State Variables	Temporary Variables	Function Block
1	**Yes**	**No**	**Yes**	**No**

Header Block *9 lines*

```
typedef struct
    {
    int       src;
    int       dst;
    double    ctime;
    } channel_asgn;
extern   int              no_in_svc;
extern   double           time;
extern   channel_asgn     slot[130][10];
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      wavelength,timeslot;
Objid    own_id,src_id;
```

State 0: discard (Enter Execs) *unforced, 9 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm());
    op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
    time+=(op_sim_time()-slot[timeslot][wavelength].ctime);
    slot[timeslot][wavelength].src=0; slot[timeslot][wavelength].dst=0; slot[timeslot][wavelength].ctime=0;
    no_in_svc-=1; op_pk_destroy(pkptr); own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
    op_intrpt_schedule_remote(op_sim_time(),1,src_id);
    }
```

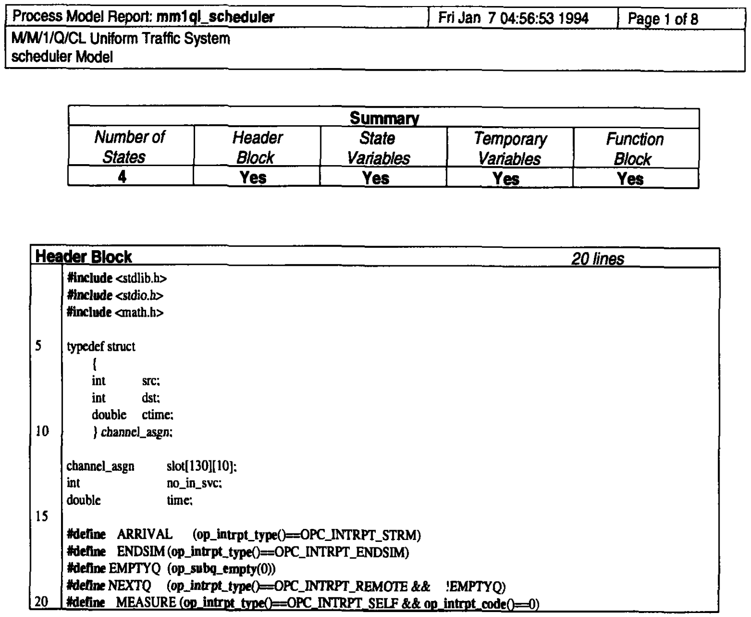
State 0: discard (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	discard

Process Model Report: **mm1ql1_scheduler**	Fri Jan 7 05:10:47 1994	Page 7 of 7
M/M/1/Q/L+1 Uniform Traffic System scheduler Model		

```
double                  svc;
extern int              no_in_svc;
extern channel_asgn     slot[130][10];

op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst",&dt); op_pk_nfd_get(packet,"svc",&svc);

ap=0,
for (i=0;i<T1;i++)              /* for each column */
    {
    k=-1;
    for (j=0;j<wavelength_no;j++)
        {
        if (slot[i][j].src==0 ) k=j;
        if (slot[i][j].src==sc || slot[i][j].dst==dt) goto next_col;
        }
if (k!=-1)              { ap=1; goto assign; }
    next_col:       ap=0;
    }

if (!ap) return 0;
else
    {
    assign:
    slot[i][k].src=sc; slot[i][k].dst=dt; slot[i][k].ctime=op_sim_time();
    op_pk_nfd_set(packet,"wavelength",k); op_pk_nfd_set(packet,"timeslot",i);
    no_in_svc+=1; processed_pk+=1; delay+=op_sim_time()-op_pk_creation_time_get(packet);
    op_pk_send_delayed(packet,0,svc); return 1;
    }
}
```

M/M/1/Q/CL: OPNET reports for the *scheduler* processor model. The *release* model is identical to that in M/M/1/Q/C+1.

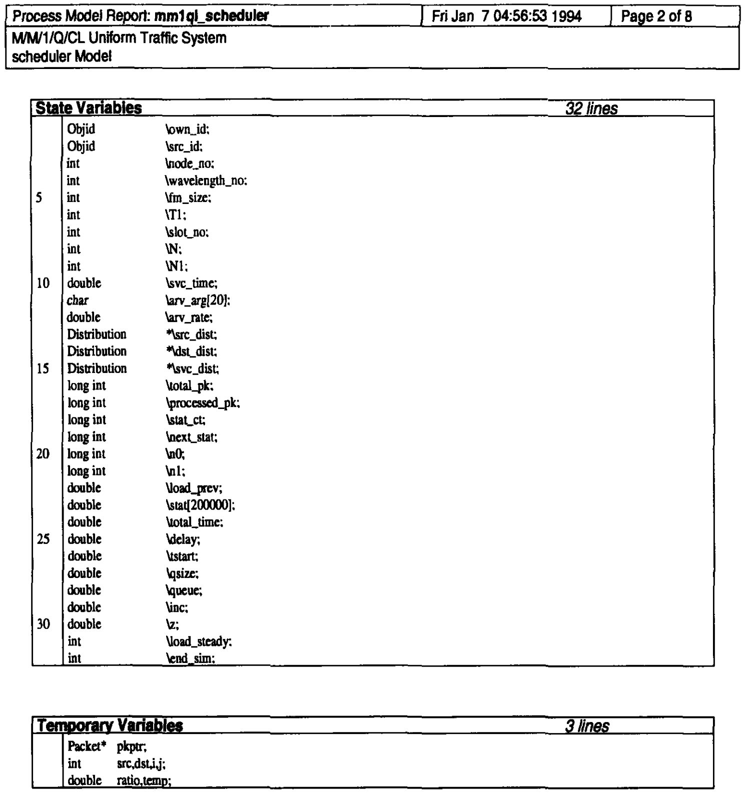


Process Model Report: **mm1ql_scheduler**	Fri Jan 7 04:56:53 1994	Page 1 of 8
M/M/1/Q/CL Uniform Traffic System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *20 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int         src;
    int         dst;
    double      ctime;
    } channel_asgn;

channel_asgn        slot[130][10];
int                 no_in_svc;
double              time;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define  ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define EMPTYQ  (op_subq_empty(0))
#define NEXTQ   (op_intrpt_type()==OPC_INTRPT_REMOTE &&   !EMPTYQ)
#define  MEASURE (op_intrpt_type()==OPC_INTRPT_SELF && op_intrpt_code()==0)
```



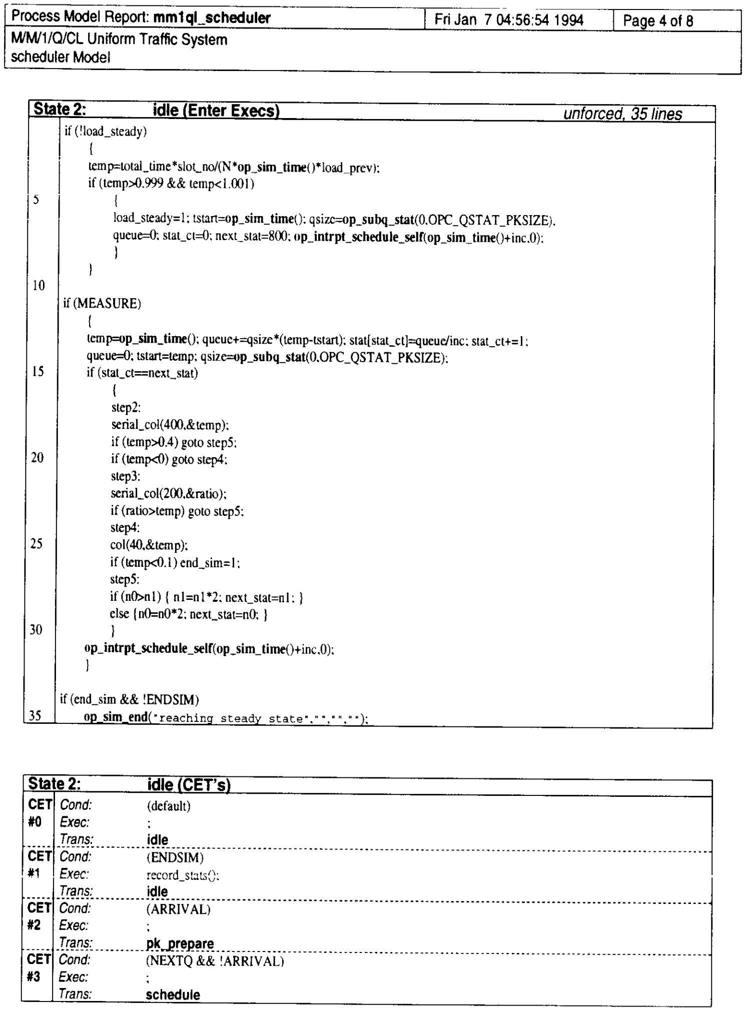
Process Model Report: **mm1ql_scheduler**	Fri Jan 7 04:56:53 1994	Page 2 of 8
M/M/1/Q/CL Uniform Traffic System scheduler Model		

State Variables *32 lines*

```
Objid           \own_id;
Objid           \src_id;
int             \node_no;
int             \wavelength_no;
int             \fm_size;
int             \T1;
int             \slot_no;
int             \N;
int             \N1;
double          \svc_time;
char            \arv_arg[20];
double          \arv_rate;
Distribution    *\src_dist;
Distribution    *\dst_dist;
Distribution    *\svc_dist;
long int        \total_pk;
long int        \processed_pk;
long int        \stat_ct;
long int        \next_stat;
long int        \n0;
long int        \n1;
double          \load_prev;
double          \stat[200000];
double          \total_time;
double          \delay;
double          \start;
double          \qsize;
double          \queue;
double          \inc;
double          \z;
int             \load_steady;
int             \end_sim;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j;
double   ratio,temp;
```



Process Model Report: **mm1ql_scheduler**	Fri Jan 7 04:56:54 1994	Page 4 of 8

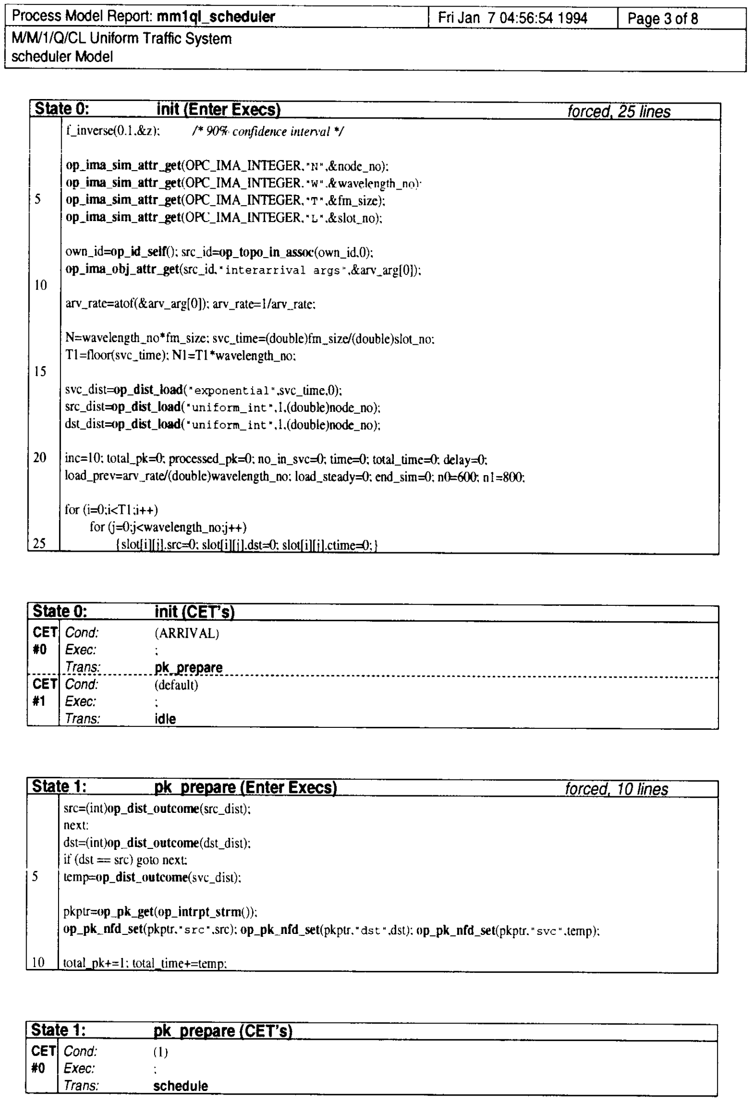
M/M/1/Q/CL Uniform Traffic System
scheduler Model

State 2: **idle (Enter Execs)** *unforced, 35 lines*

```
if (!load_steady)
    {
    temp=total_time*slot_no/(N*op_sim_time()*load_prev);
    if (temp>0.999 && temp<1.001)
        {
        load_steady=1; tstart=op_sim_time(); qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE).
        queue=0; stat_ct=0; next_stat=800; op_intrpt_schedule_self(op_sim_time()+inc,0);
        }
    }

if (MEASURE)
    {
    temp=op_sim_time(); queue+=qsize*(temp-tstart); stat[stat_ct]=queue/inc; stat_ct+=1;
    queue=0; tstart=temp; qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
    if (stat_ct==next_stat)
        {
        step2:
        serial_col(400,&temp);
        if (temp>0.4) goto step5;
        if (temp<0) goto step4;
        step3:
        serial_col(200,&ratio);
        if (ratio>temp) goto step5;
        step4:
        col(40,&temp);
        if (temp<0.1) end_sim=1;
        step5:
        if (n0>n1) { n1=n1*2; next_stat=n1; }
        else {n0=n0*2; next_stat=n0; }
        }
    op_intrpt_schedule_self(op_sim_time()+inc,0);
    }

if (end_sim && !ENDSIM)
    op_sim_end("reaching steady state","","","");
```

State 2: **idle (CET's)**

CET #0	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**
CET #1	*Cond:*	(ENDSIM)
	Exec:	record_stats();
	Trans:	**idle**
CET #2	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #3	*Cond:*	(NEXTQ && !ARRIVAL)
	Exec:	;
	Trans:	**schedule**



Process Model Report: **mm1ql_scheduler**	Fri Jan 7 04:56:54 1994	Page 3 of 8

M/M/1/Q/CL Uniform Traffic System
scheduler Model

State 0: **init (Enter Execs)** *forced, 25 lines*

```
f_inverse(0.1,&z);      /* 90% confidence interval */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no)·
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)fm_size/(double)slot_no;
T1=floor(svc_time); N1=T1*wavelength_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

inc=10; total_pk=0; processed_pk=0; no_in_svc=0; time=0; total_time=0; delay=0;
load_prev=arv_rate/(double)wavelength_no; load_steady=0; end_sim=0; n0=600; n1=800;

for (i=0;i<T1;i++)
    for (j=0;j<wavelength_no;j++)
        {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0;}
```

State 0: **init (CET's)**

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: **pk_prepare (Enter Execs)** *forced, 10 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;
temp=op_dist_outcome(svc_dist);

pkptr=op_pk_get(op_intrpt_strm());
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);

total_pk+=1; total_time+=temp;
```

State 1: **pk_prepare (CET's)**

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**schedule**

State 3: **schedule (Enter Execs)** *forced, 39 lines*

```
if ( ARRIVAL )
     {
     if (no_in_svc == N1)
          {
       if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
               { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
          elsc      if (load_steady)
                    {
                    temp=op_sim_time(); queue+=qsize*(temp-tstart); tstart=temp;
                    qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
                    }
          }
     else
          if (!find_resource(pkptr))
               {
            if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                    { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
               else      if (load_steady)
                         {
                         temp=op_sim_time(); queue+=qsize*(temp-tstart); tstart=temp;
                         qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
                         }
               }
     }

if ( NEXTQ )
          {
          ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
          for (i=0;i<ratio;i++)
               if (no_in_svc < N1)
                    {
                    pkptr=op_subq_pk_remove(0,0);
                    if (!find_resource(pkptr))
                       if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                              { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                    }
          ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
          if (ratio!=qsize && load_steady) {queue+=qsize*(temp-tstart); tstart=temp; qsize=ratio;}
          }
```

State 3: **schedule (CET's)**

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *181 lines*

```
record_stats()
     {
     int                    i,j,b;
     double                 x,s,y[40];
     extern double          time;
     extern channel_asgn    slot[130][10];

     s=(double)stat_ct/(double)40;
```

```
     b=floor(s);
     x=0;
     for (i=0;i<40;i++)
          {
          y[i]=0;
          for (j=0;j<b;j++)          y[i]+=stat[i*b+j];
          y[i]=y[i]/b; x+=y[i];
          }
     x=x/40; s=0;
     for (i=0;i<40;i++) { j=y[i]-x; s+=j*j; }
     s=s/39;
     if (x!=0) s=z*sqrt(s/40)/x;

     for (i=0; i<T1; i++)
          for (j=0; j<wavelength_no; j++)
               if (slot[i][j].src!=0) time+=(op_sim_time()-slot[i][j].ctime);

     op_stat_write_scalar("load",total_time*slot_no/(N*op_sim_time()));
     op_stat_write_scalar("cu",time*slot_no/(N*op_sim_time()));
     op_stat_write_scalar("Q",x);
     op_stat_write_scalar("D",delay/(double)processed_pk);
     op_stat_write_scalar("qd",s);
     op_stat_write_scalar("c",stat_ct);
     printf("%d\tQ: %f\tD: %f\tdev: %f\n",stat_ct,x,(delay/processed_pk),s);
     }


f_inverse (x,z1)
     double    x,*z1;
     {
     double    y,p,p0,p1,p2,p3,p4,q0,q1,q2,q3,q4;

     x=1-x/2;
     p0=-0.322232431088e0; p1=-1.0; p2=-0.342242088547e0; p3=-0.0204231210245e0; p4=-0.453642210148e-4;
     q0=0.099348462606e0; q1=0.588581570495e0; q2=0.531103462366e0; q3=0.10353775285e0; q4=0.38560700634e-2;

     p=x;
     if (p>0.5) p=1.0-p;
     y=sqrt(-log(p*p));
     *z1=y+(p0+y*(p1+y*(p2+y*(p3+y*p4))))/(q0+y*(q1+y*(q2+y*(q3+y*q4))));
     if (x<0.5) *z1=-*z1;
     }


serial_col(n,p1)
     int       n;
     double    *p1;
     {
     int  i,j,k,b;
     double    p,x,x1,x2,c,c1,c2,s,s1,s2,y[400];

     b=next_stat/n; k=n/2; x=0; x1=0; x2=0;
     for (i=0;i<n;i++)
          {
          y[i]=0;
          for (j=0;j<b;j++)          y[i]+=stat[i*b+j];
          y[i]=y[i]/b; x+=y[i];
          if (i<k) x1+=y[i]; else x2+=y[i];
          }
```

```
    x=x/n; x1=x1/k; x2=x2/k; s=0; s1=0; s2=0; c=0; c1=0; c2=0;
    for (i=0;i<n;i++)
        {
        j=y[i]-x;
        s+=j*j;
        if (i<(n-1)) c+=j*(y[i+1]-x);
        if (i<k)
            {
            j=y[i]-x1; s1+=j*j;
            if (i<(k-1)) c1+=j*(y[i+1]-x1);
            }
        else
            {
            j=y[i]-x2; s2+=j*j;
            if (i<(n-1)) c2+=j*(y[i+1]-x2);
            }
        }
    if (s==0) *p1=0; else *p1=2*c/s-(c1/s1+c2/s2)/2;
    }


col(n,p1)
    int         n;
    double      *p1;
    {
    int   i,j,b;
    double      p,x,s,y[400];

    b=next_stat/n; x=0;
    for (i=0;i<n;i++)
        {
        y[i]=0;
        for (j=0;j<b;j++)           y[i]+=stat[i*b+j];
        y[i]=y[i]/b; x+=y[i];
        }
    x=x/n; s=0;
    for (i=0;i<n;i++) { j=y[i]-x; s+=j*j; }
    s=s/(n-1); s=sqrt(s/n);
    if (s==0) *p1=0; else *p1=z*s/x;
    printf("%d: %f\t%f\n",stat_ct,x,*p1);
    }


int find_resource(packet)
    Packet*   packet;
    {
    int                     i,j,ap,fit,ps,pd,cf,cs,cd,nf,ns,nd,ca[8],na[8],as[2],sc,dt,
    double                  svc;
    extern int              no_in_svc;
    extern channel_asgn     slot[130][10];

    op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst",&dt); op_pk_nfd_get(packet,"svc",&svc);

    ap=-1; ps=-1; pd=-1;
    for (i=0;i<T1;i++)              /* for each column */
        {
        if (i==0)
            {
            cf=0; cs=-1; cd=-1;
```

```
            for (j=0;j<wavelength_no;j++)
                {
                if (slot[i][j].src==0) { ca[cf]=j; cf+=1; }
                if (slot[i][j].src==sc) cs=j;
                if (slot[i][j].dst==dt) cd=j;
                }
            }

        if (i==(T1-1)) { ns=-1; nd=-1; }
        else
            {
            nf=0; ns=-1; nd=-1;
            for (j=0;j<wavelength_no;j++)
                {
                if (slot[i+1][j].src==0) { na[nf]=j; nf+=1; }
                if (slot[i+1][j].src==sc) ns=j;
                if (slot[i+1][j].dst==dt) nd=j;
                }
            }

        if (cf==0 || cs!=-1 || cd!=-1) goto next_col;

        for (j=0;j<cf;j++)
            {
            fit=0;
            if (ps!=-1)
                if (ca[j]==ps) fit+=1; else continue;
            if (pd!=-1)
                if (ca[j]==pd) fit+=1; else continue;
            if (ns!=-1)
                if (ca[j]==ns) fit+=1; else continue;
            if (nd!=-1)
                if (ca[j]==nd) fit+=1; else continue;

            if (fit>ap) {as[0]=i; as[1]=ca[j]; ap=fit;}
            if (ap==4) goto assign;
            }
        next_col:
        if (i!=(T1-1))
            {
            ps=cs; pd=cd; cs=ns; cd=nd; cf=nf;
            for (j=0;j<nf;j++) ca[j]=na[j];
        }
        }

    if (ap==-1) return 0;
    else
        {
        assign:
        slot[as[0]][as[1]].src=sc; slot[as[0]][as[1]].dst=dt; slot[as[0]][as[1]].ctime=op_sim_time();
        op_pk_nfd_set(packet,"wavelength",as[1]); op_pk_nfd_set(packet,"timeslot",as[0]);
        no_in_svc+=1; processed_pk+=1; delay+=op_sim_time()-op_pk_creation_time_get(packet);
        op_pk_send_delayed(packet,0,svc); return 1;
        }
    }
```

M/M/1/Q/RL: OPNET reports for the *scheduler* and *release* processor models.

Process Model Report: **mm1qr_scheduler**	Fri Jan 7 04:26:56 1994	Page 1 of 11
M/M/1/Q/RL Uniform Traffic System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *29 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
	{
	int		src;
	int		dst;
	int		len;
	double	ctime;
	} channel_asgn;

typedef struct
	{
	int	slot;
	int	wavelength;
	int	len;
	} other_asgn;

channel_asgn		pkt[1025];
other_asgn		other[1025][15];
int			no_in_svc,released_pk,slot[130][10];
double			time;

#define	ARRIVAL		(op_intrpt_type()==OPC_INTRPT_STRM)
#define	ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define EMPTYQ	(op_subq_empty(0))
#define NEXTQ	(op_intrpt_type()==OPC_INTRPT_REMOTE &&	!EMPTYQ)
#define	MEASURE (op_intrpt_type()==OPC_INTRPT_SELF && op_intrpt_code()==0)
```

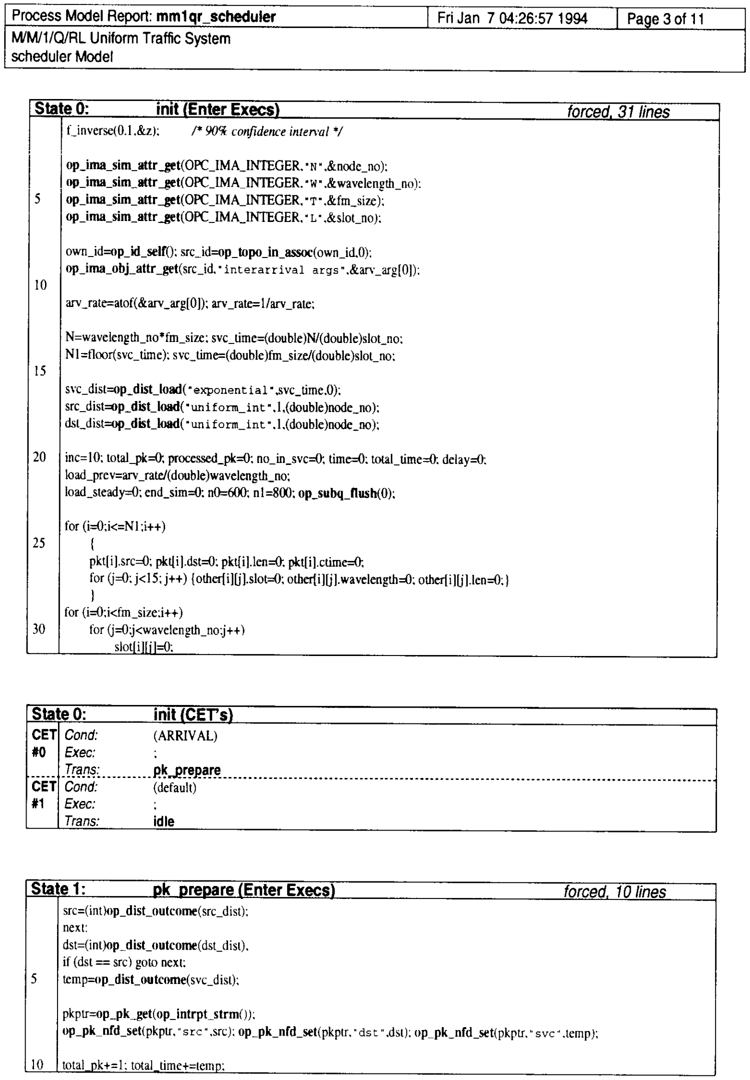
Process Model Report: **mm1qr_scheduler**	Fri Jan 7 04:26:56 1994	Page 2 of 11
M/M/1/Q/RL Uniform Traffic System scheduler Model		

State Variables *32 lines*

```
Objid			\own_id;
Objid			\src_id;
int			\node_no;
int			\wavelength_no;
int			\fm_size;
int			\T1;
int			\slot_no;
int			\N;
int			\N1;
double			\svc_time;
char			\arv_arg[20];
double			\arv_rate;
Distribution		*\src_dist;
Distribution		*\dst_dist;
Distribution		*\svc_dist;
long int		\total_pk;
long int		\processed_pk;
long int		\stat_ct;
long int		\next_stat;
long int		\n0;
long int		\n1;
double			\load_prev;
double			\stat[200000];
double			\total_time;
double			\delay;
double			\tstart;
double			\qsize;
double			\queue;
double			\inc;
double			\z;
int			\load_steady;
int			\end_sim;
```

Temporary Variables *3 lines*

```
Packet*	pkptr;
int	src,dst,i,j;
double	ratio,temp;
```



Process Model Report: **mm1qr_scheduler**	Fri Jan 7 04:26:57 1994	Page 3 of 11
M/M/1/Q/RL Uniform Traffic System scheduler Model		

State 0: init (Enter Execs) *forced, 31 lines*

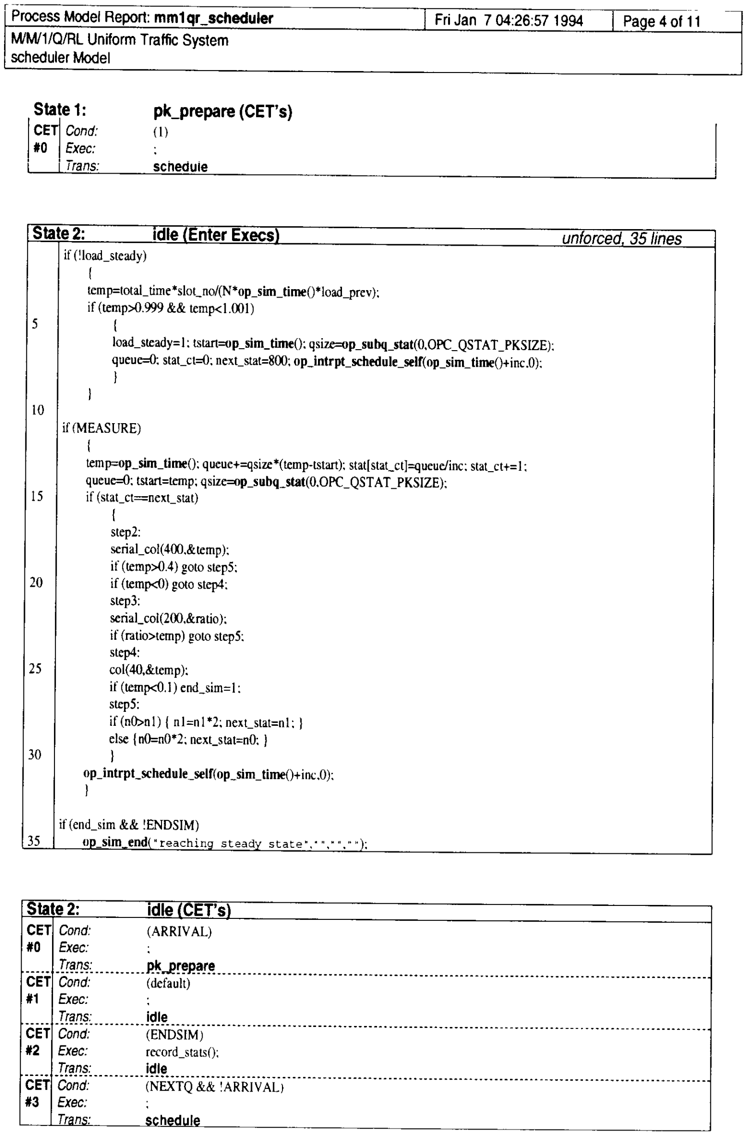
```
f_inverse(0.1,&z);        /* 90% confidence interval */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)N/(double)slot_no;
N1=floor(svc_time); svc_time=(double)fm_size/(double)slot_no;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);

inc=10; total_pk=0; processed_pk=0; no_in_svc=0; time=0; total_time=0; delay=0;
load_prev=arv_rate/(double)wavelength_no;
load_steady=0; end_sim=0; n0=600; n1=800; op_subq_flush(0);

for (i=0;i<=N1;i++)
    {
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0;
    for (j=0; j<15; j++) {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;}
    }
for (i=0;i<fm_size;i++)
    for (j=0;j<wavelength_no;j++)
        slot[i][j]=0;
```

State 0: init (CET's)

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 10 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist),
if (dst == src) goto next;
temp=op_dist_outcome(svc_dist);

pkptr=op_pk_get(op_intrpt_strm());
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);

total_pk+=1; total_time+=temp;
```



Process Model Report: **mm1qr_scheduler**	Fri Jan 7 04:26:57 1994	Page 4 of 11
M/M/1/Q/RL Uniform Traffic System scheduler Model		

State 1: pk_prepare (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 35 lines*

```
if (!load_steady)
    {
    temp=total_time*slot_no/(N*op_sim_time()*load_prev);
    if (temp>0.999 && temp<1.001)
        {
        load_steady=1; tstart=op_sim_time(); qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
        queue=0; stat_ct=0; next_stat=800; op_intrpt_schedule_self(op_sim_time()+inc,0);
        }
    }

if (MEASURE)
    {
    temp=op_sim_time(); queue+=qsize*(temp-tstart); stat[stat_ct]=queue/inc; stat_ct+=1;
    queue=0; tstart=temp; qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
    if (stat_ct==next_stat)
        {
        step2:
        serial_col(400,&temp);
        if (temp>0.4) goto step5;
        if (temp<0) goto step4;
        step3:
        serial_col(200,&ratio);
        if (ratio>temp) goto step5;
        step4:
        col(40,&temp);
        if (temp<0.1) end_sim=1;
        step5:
        if (n0>n1) { n1=n1*2; next_stat=n1; }
        else {n0=n0*2; next_stat=n0; }
        }
    op_intrpt_schedule_self(op_sim_time()+inc,0);
    }

if (end_sim && !ENDSIM)
    op_sim_end("reaching steady state","","","");
```

State 2: idle (CET's)

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#2	*Exec:*	record_stats();
	Trans:	**idle**
CET	*Cond:*	(NEXTQ && !ARRIVAL)
#3	*Exec:*	;
	Trans:	**schedule**

State 3: schedule (Enter Execs) *forced, 39 lines*

```
if ( ARRIVAL )
    {
    if (no_in_svc == N1)
        {
      if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
            { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
        else        if (load_steady)
                    {
                    temp=op_sim_time(); queue+=qsize*(temp-tstart); tstart=temp;
                    qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
                    }
        }
    else
        if (!find_resource(pkptr))
            {
          if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
            else        if (load_steady)
                        {
                        temp=op_sim_time(); queue+=qsize*(temp-tstart); tstart=temp;
                        qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
                        }
            }
    }

if ( NEXTQ )
        {
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if (no_in_svc < N1)
                {
                pkptr=op_subq_pk_remove(0,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                }
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize && load_steady) {queue+=qsize*(temp-tstart); tstart=temp; qsize=ratio;}
        }
```

State 3: schedule (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *320 lines*

```
record_stats()
    {
    int             i,j,b;
    double          x,s,y[40];
    extern int      slot[130][10];
```

```
    extern double           time;
    extern channel_asgn     pkt[1025];

    s=(double)stat_ct/(double)40; b=floor(s); x=0;
    for (i=0;i<40;i++)
        {
        y[i]=0;
        for (j=0;j<b;j++)           y[i]+=stat[i*b+j];
        y[i]=y[i]/b; x+=y[i];
        }
    x=x/40; s=0;
    for (i=0;i<40;i++) { j=y[i]-x; s+=j*j; }
    s=s/39;
    if (x!=0) s=z*sqrt(s/40)/x;

    for (i=1; i<=N1; i++)
        if (pkt[i].src!=0) time+=(op_sim_time()-pkt[i].ctime);

    op_stat_write_scalar("load",total_time*slot_no/(N*op_sim_time()));
    op_stat_write_scalar("cu",time*slot_no/(N*op_sim_time()));
    op_stat_write_scalar("Q",x);
    op_stat_write_scalar("D",delay/(double)processed_pk);
    op_stat_write_scalar("qd",s);
    op_stat_write_scalar("c",stat_ct);
    printf("%d\tQ: %f\tD: %f\tdev: %f\n",stat_ct,x,(delay/processed_pk),s);
    }


f_inverse (x,z1)
    double    x,*z1;
    {
    double    y,p,p0,p1,p2,p3,p4,q0,q1,q2,q3,q4;

    x=1-x/2;
    p0=-0.322232431088e0; p1=-1.0; p2=-0.342242088547e0; p3=-0.0204231210245e0; p4=-0.453642210148e-4;
    q0=0.099348462606e0; q1=0.588581570495e0; q2=0.531103462366e0; q3=0.10353775285e0; q4=0.38560700634e-2;

    p=x;
    if (p>0.5) p=1.0-p;
    y=sqrt(-log(p*p));
    *z1=y+(p0+y*(p1+y*(p2+y*(p3+y*p4))))/(q0+y*(q1+y*(q2+y*(q3+y*q4))));
    if (x<0.5) *z1=-*z1;
    }


serial_col(n,p1)
    int         n;
    double      *p1;
    {
    int    i,j,k,b;
    double      p,x,x1,x2,c,c1,c2,s,s1,s2,y[400];

    b=next_stat/n; k=n/2; x=0; x1=0; x2=0;
    for (i=0;i<n;i++)
        {
        y[i]=0;
        for (j=0;j<b;j++)           y[i]+=stat[i*b+j];
        y[i]=y[i]/b; x+=y[i];
        if (i<k) x1+=y[i]; else x2+=y[i];
```

```
            }
        x=x/n; x1=x1/k; x2=x2/k; s=0; s1=0; s2=0; c=0; c1=0; c2=0;
        for (i=0;i<n;i++)
            {
            j=y[i]-x; s+=j*j;
            if (i<(n-1)) c+=j*(y[i+1]-x);
            if (i<k)
                {
                j=y[i]-x1; s1+=j*j;
                if (i<(k-1)) c1+=j*(y[i+1]-x1);
                }
            else
                {
                j=y[i]-x2; s2+=j*j;
                if (i<(n-1)) c2+=j*(y[i+1]-x2);
                }
            }
        if (s==0) *p1=0;
        else        *p1=2*c/s-(c1/s1+c2/s2)/2;
        }


col(n,p1)
        int         n;
        double      *p1;
        {
        int   i,j,b;
        double     p,x,s,y[400];

        b=next_stat/n;
        x=0;
        for (i=0;i<n;i++)
            {
            y[i]=0;
            for (j=0;j<b;j++)           y[i]+=stat[i*b+j];
            y[i]=y[i]/b; x+=y[i];
            }
        x=x/n; s=0;
        for (i=0;i<n;i++)
            { j=y[i]-x; s+=j*j; }
        s=s/(n-1); s=sqrt(s/n);
        if (s==0) *p1=0; else *p1=z*s/x;
        printf("%d:  %f\t%f\n",stat_ct,x,*p1);
        }


int find_resource(packet)
        Packet*    packet;
        {
        int                      i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next,sc,dt;
        int                      conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
        double                   svc;
        extern int               no_in_svc,slot[130][10];
        extern other_asgn        other[1025][15];
        extern channel_asgn      pkt[1025];

        op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst',&dt). op_pk_nfd_get(packet,"svc",&svc);

        ct=0;
```

```
        for (j=0; j<wavelength_no; j++) empty[j]=0;

        for (i=0;i<fm_size;i++)
            {

            conflict[i][0]=-1; conflict[i][1]=-1;
            for (k=0; k<wavelength_no; k++)
                {
                if ( slot[i][k]==0 ) empty[k]+=1;
                if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
                if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
                }

            go_back=0;
            if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
                {
                for (k=0; k<wavelength_no; k++)
                    {
                    if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
                        {
                        if ( slot[i][k]!=0 ) j=i-empty[k]-1; else j=i-empty[k];
                        if ( j<0 ) oh=0;
                        else
                            if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                            else oh=2;
                        if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
                        insert:
                        if ( ct==0 )
                            {
                            if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k];
                            else        store[0][0]=i-empty[k]+1;
                            store[0][1]=k;        store[0][2]=empty[k];          store[0][3]=oh.
                            }
                        else
                            for (j=(ct-1); j>=0; j--)
                                {
                                if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
                                    {
                                    store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
                                    store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
                                    }
                                if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
                                    {
                                    store_ct:
                                    if ( slot[i][k]!=0 || go_back ) store[j+1][0]=i-empty[k];
                                    else        store[j+1][0]=i-empty[k]+1;
                                    store[j+1][1]=k; store[j+1][2]=empty[k]; store[j+1][3]=oh; break;
                                    }
                                if ( j==0 ) { j=-1; goto store_ct; }
                                }
                        if ( go_back ) goto back;
                        empty[k]=0; ct+=1;
                        }
                    }
                }
            else
                {
                for ( k=0; k<wavelength_no; k++)
                    {
```

```
                    if ( empty[k]>0 )
                        {
                        if ( slot[i][k]==0 ) empty[k]-=1;
                        if ( empty[k]==0) continue;
                        j=i-empty[k]-1;
                        if ( j<0 ) oh=0;
                        else
                            if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                            else oh=2;
                        if ((conflict[i][0]==-1 || conflict[i][0]==k)&&(conflict[i][1]==-1 || conflict[i][1]==k)) oh=oh;
                        else      if ( oh==0 ) oh=1; else oh=3;
                        if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0; continue; }
                        go_back=1;
                        goto insert;
                        back:
                        empty[k]=0; ct+=1;
                        }
                    }
                }
            }

    nd=slot_no; i_ct=0; waste=0; next=0; asgn[0]=-1; asgn[1]=0;
    for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
    for (i=0; i<ct; i++)
        {
        if ( store[i][3]==0 ) fit = 0;
        else if ( store[i][3]==3 ) fit = 2; else fit = 1;
        j=store[i][2]-fit;

        if (next)
            {
            if ( j>=nd )
                {
                k=j-nd+fit;
                if ( fit<= ap1 && k<ap )
                    {
                    for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                        if ( rmb[oh] ) { k=-1; break; }
                    if ( k!=-1 )
                        {
                        ap=k; ap1=fit; temp[i_ct]=i;
                        for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                        k=store[i][0]-1; if ( k<0 ) k=0;
                        go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                        for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1;
                        }
                    }
                }
            if ( j<nd || i==(ct-1) )
                {
                waste+=ap1; i_ct+=1;
                if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
                if ( j<nd )
                    {
```

```
                    temp[0]=i; nd=slot_no-j; waste=fit; i_ct=1; next=0;
                    for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                    k=store[i][0]-1, if ( k<0 ) k=0;
                    go_back=store[i][0]+store[i][2]+1, if ( go_back>fm_size ) go_back=fm_size;
                    for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                    }
                }
            continue;
            }

        k=0;
        for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
            if ( rmb[oh] ) { k=1; break; }
        if ( k==1 ) continue;
        else
            {
            temp[i_ct]=i;
            k=store[i][0]-1; if ( k<0 ) k=0;
            go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
            for (oh=k; oh<go_back; oh++)
                if ( j>nd ) tmp_rmb[oh]=1; else rmb[oh]=1;
            }

        if ( j<nd ) { nd-=j; waste+=fit; i_ct+=1; continue; }
        if ( j==nd )
            {
            waste+=fit; i_ct+=1;
            if ( waste<asgn[0] || asgn[0]==-1 )
                {
                asgn[0]=waste; asgn[1]=i_ct;
                for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                if ( asgn[0]==0 ) goto assign;
                }
            nd=slot_no; i_ct=0; waste=0;
            for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
            continue;
            }
        if ( j>nd ) { ap=j-nd+fit; ap1=fit; next=1; }
        if ( next && i==(ct-1) )
            {
            waste+=ap1; i_ct+=1;
            if ( waste<asgn[0] || asgn[0]==-1 )
                {
                asgn[0]=waste; asgn[1]=i_ct;
                for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                if ( asgn[0]==0 ) goto assign;
                }
            }
        }

    if(asgn[0]==-1) return 0;
    else
        {
        assign:
        for (k=1; k<=N1; k++)
            {
            if (pkt[k].src!=0) continue;
            pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1]; pkt[k].ctime=op_sim_time(); nd=slot_no;
            for (i=2; i<=asgn[1]; i++)
```

Process Model Report: **mm1qr_scheduler** | Fri Jan 7 04:26:59 1994 | Page 11 of 11

M/M/1/Q/RL Uniform Traffic System
scheduler Model

```
                        {
                        oh=store[asgn[i]][0]; ct=store[asgn[i]][2]; nd-=ct;
                        if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                        else        if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                                    else        if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                        for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                        other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
                        }
                i=asgn[1]+1; oh=store[asgn[i]][0];
                if ( store[asgn[i]][3]==3 ) oh+=1;
                else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
                for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
                other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
                no_in_svc+=1; processed_pk+=1; delay+=op_sim_time()-op_pk_creation_time_get(packet);
                op_pk_nfd_set(packet,"timeslot",oh); op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
                op_pk_send_delayed(packet,0,svc); break;
                }
        return 1;
        }
    }
```

Process Model Report: **mm1qr_release** | Fri Jan 7 04:35:29 1994 | Page 1 of 1

M/M/1/Q/RL Uniform Traffic System
release Model

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *17 lines*

```
typedef struct
    {
    int         src;
    int         dst;
    int         len;
    double      ctime;
    } channel_asgn;
typedef struct
    {
    int         slot;
    int         wavelength;
    int         len;
    } other_asgn;
extern    int               no_in_svc,slot[130][10];
extern    double            time;
extern    channel_asgn      pkt[1025];
extern    other_asgn        other[1025][15];
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       wavelength,timeslot,i,j,k;
Objid     own_id,src_id;
```

State 0: **discard (Enter Execs)** *unforced, 13 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm());
    op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
    i=slot[timeslot][wavelength]; time+=(op_sim_time()-pkt[i].ctime);
    for (j=0; j<pkt[i].len; j++)
        {
        for (k=other[i][j].slot; k<(other[i][j].slot+other[i][j].len); k++) slot[k][other[i][j] wavelength]=0;
        other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;
        }
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0; no_in_svc-=1; op_pk_destroy(pkptr);
    own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0); op_intrpt_schedule_remote(op_sim_time(),1,src_id);
    }
```

State 0: **discard (CET's)**

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**discard**

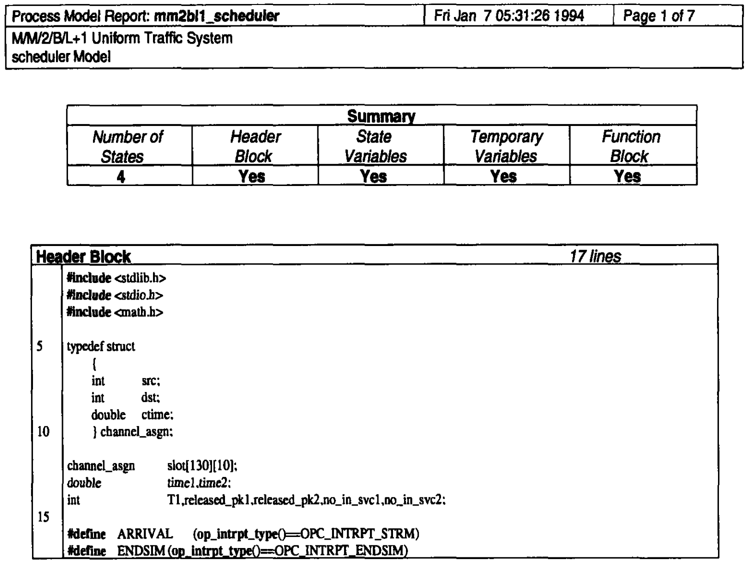
B.4 Two Class, Uniform Traffic System

The termination criterion used for *M/M/2/B/L+1*, *M/M/2/B/CL*, and *M/M/2/B/RL* systems is to end the program when either 99999 seconds have elapsed or the "steady state" condition has been reached. The program starts to monitor periodically the channel utilization (*cu*), the blocking probability for the system and for each type of the sessions (*pb*, pb_1, pb_2) once the measured load is within 1% of the offered load derived in Section 4.1. If the newly measured *cu, pb*, pb_1, and pb_2 are within 1% of the previous measured values and the total number of sessions processed by the system is more than 20,000, we say that "steady state" condition is reached. The program will be ready for termination once the measured load is within 5% of the offered load.

The seed used for each simulation has value *NWs*, where *N* is the number of users, *W* the wavelength channels, and *s* equals to one if the offered load is normalized against sessions with smaller of the two throughput requirements and zero otherwise (refer to Section 4.1). So for *N* equals to forty, *W* equals to eight and *s* equals to one, the seed is 4081.

B.4.1 Blocking System

M/M/2/B/L+1: OPNET reports for the *scheduler* and *release* processor models.



Process Model Report: **mm2bl1_scheduler**	Fri Jan 7 05:31:26 1994	Page 1 of 7
M/M/2/B/L+1 Uniform Traffic System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *17 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
	{
	int		src;
	int		dst;
	double	ctime;
	} channel_asgn;

channel_asgn		slot[130][10];
double			time1,time2;
int			T1,released_pk1,released_pk2,no_in_svc1,no_in_svc2;

#define	ARRIVAL	(op_intrpt_type()==OPC_INTRPT_STRM)
#define	ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

Process Model Report: **mm2bl1_scheduler** | Fri Jan 7 05:31:26 1994 | Page 2 of 7

M/M/2/B/L+1 Uniform Traffic System
scheduler Model

State Variables *40 lines*

```
Objid          \own_id;
Objid          \src_id;
int            \node_no;
int            \wavelength_no;
int            \fm_size;
int            \T2;
int            \slot_no;
int            \slot_no1;
int            \slot_no2;
int            \N;
int            \N1;
int            \N2;
int            \s;
double         \a;
double         \svc_time;
char           \arv_arg[20];
double         \arv_rate;
Distribution   *\src_dist;
Distribution   *\dst_dist;
Distribution   *\svc_dist;
Distribution   *\cls_dist;
long int       \blocked_pk1;
long int       \blocked_pk2;
long int       \total_pk1;
long int       \total_pk2;
double         \cu_prev;
double         \pb_prev;
double         \pb1_prev;
double         \pb2_prev;
double         \load_prev;
int            \pk_count1;
int            \pk_count2;
int            \tr_count1;
int            \tr_count2;
int            \cu_steady;
int            \pb_steady;
int            \pb1_steady;
int            \pb2_steady;
int            \load_steady;
int            \ready;
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       i,j,src,dst,type;
double    temp,ratio;
```

Process Model Report: **mm2bl1_scheduler** | Fri Jan 7 05:31:26 1994 | Page 3 of 7

M/M/2/B/L+1 Uniform Traffic System
scheduler Model

State 0: init (Enter Execs) *forced, 36 lines*

```
/* assume L1 <= L2 */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L1",&slot_no1);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L2",&slot_no2);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"s",&s);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"a",&a);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=fm_size/((1+slot_no2)+(1+slot_no1)*a/(1-a)); T2=floor(svc_time);
N2=T2*wavelength_no; svc_time=(double)(fm_size-T2*(1+slot_no2))/(double)(1+slot_no1); T1=floor(svc_time);
N1=T1*wavelength_no;

if (s==0) s=-1;
if ( (slot_no1-slot_no2)*s >=0) svc_time=(double)fm_size/(double)slot_no2;
else svc_time=(double)fm_size/(double)slot_no1;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);
cls_dist=op_dist_load("bernoulli",a,0);
load_prev=(a*slot_no1+(1-a)*slot_no2)*arv_rate*svc_time/(double)N;

blocked_pk1=0; blocked_pk2=0; total_pk1=0; total_pk2=0; released_pk1=0; released_pk2=0;
no_in_svc1=0; no_in_svc2=0; time1=0; time2=0;
cu_steady=0; pb_steady=0; pb1_steady=0; pb2_steady=0; load_steady=0; ready=0;

for (i=0;i<(T1+T2);i++)
      for (j=0;j<wavelength_no;j++)
            {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0;}
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 8 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

type=(int)op_dist_outcome(cls_dist);
if (type==1) total_pk1+=1; else total_pk2+=1;
pkptr=op_pk_get(op_intrpt_strm());
```

State 1:		**pk_prepare (CET's)**
CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**schedule**

State 2: **idle (Enter Execs)** *unforced, 80 lines*

```
if (!load_steady && released_pk1>0 && released_pk2>0)
     {
     ratio=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()*load_prev);
     if ( ratio>0.999 && ratio<1.001 )
          {
          load_steady=1; ratio=0; temp=0;
          for (i=0; i<T1; i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
          for (i=T1; i<(T1+T2); i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
          cu_prev=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time());
          pb_prev=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2);
          pb1_prev=(double)blocked_pk1/(double)total_pk1; pb2_prev=(double)blocked_pk2/(double)total_pk2;
          temp=cu_prev*N1; pk_count1=floor(temp); temp=(double)released_pk1/(double)pk_count1; tr_count1=floor(temp)+20;
          temp=cu_prev*N2; pk_count2=floor(temp); temp=(double)released_pk2/(double)pk_count2; tr_count2=floor(temp)+20;
          }
     }

if ( load_steady && !ready && released_pk1>(tr_count1*pk_count1) && released_pk2>(tr_count2*pk_count2) )
     {
     if (!cu_steady)
          {
          ratio=0; temp=0;
          for (i=0; i<T1; i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
          for (i=T1; i<(T1+T2); i++)
               for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
          temp=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time()); ratio=temp/cu_prev;
          if (ratio>0.999 && ratio<1.001 && temp<(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()))
               cu_steady=1;
          else
               cu_prev=temp;
          }
     if (!pb_steady)
          {
          temp=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2);
          if (pb_prev!=0) ratio=temp/pb_prev;
          else      if (temp==0) ratio=1; else ratio=0;
          if (ratio > 0.999 && ratio < 1.001)          pb_steady=1; else pb_prev=temp;
          }
     if (!pb1_steady)
          {
          temp=(double)blocked_pk1/(double)total_pk1;
          if (pb1_prev!=0) ratio=temp/pb1_prev;
```

```
          else      if (temp==0) ratio=1; else ratio=0;
          if (ratio > 0.999 && ratio < 1.001)          pb1_steady=1; else pb1_prev=temp;
          }
     if (!pb2_steady)
          {
          temp=(double)blocked_pk2/(double)total_pk2;
          if (pb2_prev!=0) ratio=temp/pb2_prev;
          else      if (temp==0) ratio=1; else ratio=0;
          if (ratio > 0.999 && ratio < 1 001)          pb2_steady=1; else pb2_prev=temp;
          }
     if ( cu_steady && pb_steady && pb1_steady && pb2_steady ) ready=1;
     if (!ready)
               {
               ratio=0; temp=0;
               for (i=0; i<T1; i++)
                    for (j=0; j<wavelength_no; j++)
                         if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
               for (i=T1; i<(T1+T2); i++)
                    for (j=0; j<wavelength_no; j++)
                         if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
               ratio=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time());
               temp=ratio*N1; pk_count1=2*floor(temp); tr_count1+=1;
               temp=ratio*N2; pk_count2=2*floor(temp); tr_count2+=1;
               }
     }

if (ready && (total_pk1+total_pk2)>20000)
     {
     temp=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
     if ( ratio > 0.995 && ratio < 1.005 && !ENDSIM)
          op_sim_end("reaching steady state","","","");
     }
```

State 2:		**idle (CET's)**
CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**
CET #2	*Cond:*	(ENDSIM)
	Exec:	record_stats();
	Trans:	**idle**

State 3: schedule (Enter Execs) *forced, 11 lines*

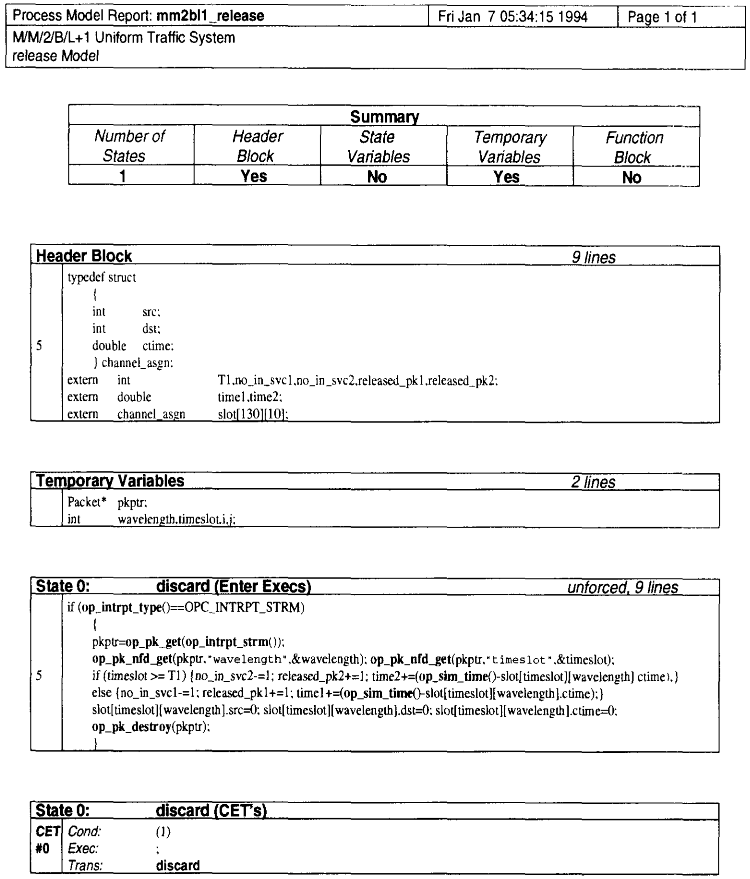
```
if (type==1 && no_in_svc1==N1)
     {blocked pk1+=1; op_pk_destroy(pkptr);}
else
     if (type==0 && no_in_svc2==N2)
          {blocked_pk2+=1; op_pk_destroy(pkptr);}
     else
          if (!find_resource(src,dst,pkptr,type))
               {
               op_pk_destroy(pkptr);
               if (type==1) blocked_pk1+=1; else blocked_pk2+=1;
               }
```

State 3: schedule (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *54 lines*

```
record_stats()
     {
     int                    i,j;
     extern double          time1,time2;
     extern channel_asgn    slot[130][10];

     for (i=0; i<T1; i++)
          for (j=0; j<wavelength_no; j++)
               if (slot[i][j].src!=0) time1+=(op_sim_time()-slot[i][j].ctime);
     for (i=T1; i<(T1+T2); i++)
          for (j=0; j<wavelength_no; j++)
               if (slot[i][j].src!=0) time2+=(op_sim_time()-slot[i][j].ctime);
     op_stat_write_scalar("load",(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()));
   op_stat_write_scalar("cu",(slot_no1*time1+slot_no2*time2)/(N*op_sim_time()));
     op_stat_write_scalar("pb",(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2));
     op_stat_write_scalar("pb1",(double)blocked_pk1/(double)total_pk1);
     op_stat_write_scalar("pb2",(double)blocked_pk2/(double)total_pk2);
     }


int find_resource(sc,dt,packet,tp)
     int        sc,dt,tp;
     Packet*    packet;
     {
     int                    i,j,k,ap,high,low;
     double                 svc;
     extern int             T1,no_in_svc1,no_in_svc2;
     extern channel_asgn    slot[130][10];

     ap=0;
     if (tp==1) { low=0; high=T1; } else { low=T1; high=T1+T2; }

   for (i=low; i<high; i++)                    /* for each column */
          {
          k=-1;
          for (j=0;j<wavelength_no;j++)
               {
               if (slot[i][j].src==0 ) k=j;
               if (slot[i][j].src==sc || slot[i][j].dst==dt) goto next_col;
               }
     if (k!=-1)             { ap=1; goto assign; }
          next_col:         ap=0;
          }

     if (!ap) return 0;
     else
          {
          assign:
          slot[i][k].src=sc; slot[i][k].dst=dt; slot[i][k].ctime=op_sim_time();
          op_pk_nfd_set(packet,"wavelength",k); op_pk_nfd_set(packet,"timeslot",i); svc=op_dist_outcome(svc_dist);
          if (tp==1) no_in_svc1+=1; else no_in_svc2+=1; op_pk_send_delayed(packet,0,svc);
          return 1;
          }
     }
```



Process Model Report: **mm2bl1_release** | Fri Jan 7 05:34:15 1994 | Page 1 of 1

M/M/2/B/L+1 Uniform Traffic System
release Model

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *9 lines*

```
typedef struct
    {
    int       src;
    int       dst;
    double    ctime;
    } channel_asgn;
extern  int            T1,no_in_svc1,no_in_svc2,released_pk1,released_pk2;
extern  double         time1,time2;
extern  channel_asgn   slot[130][10];
```

Temporary Variables *2 lines*

```
Packet*  pkptr;
int      wavelength,timeslot,i,j;
```

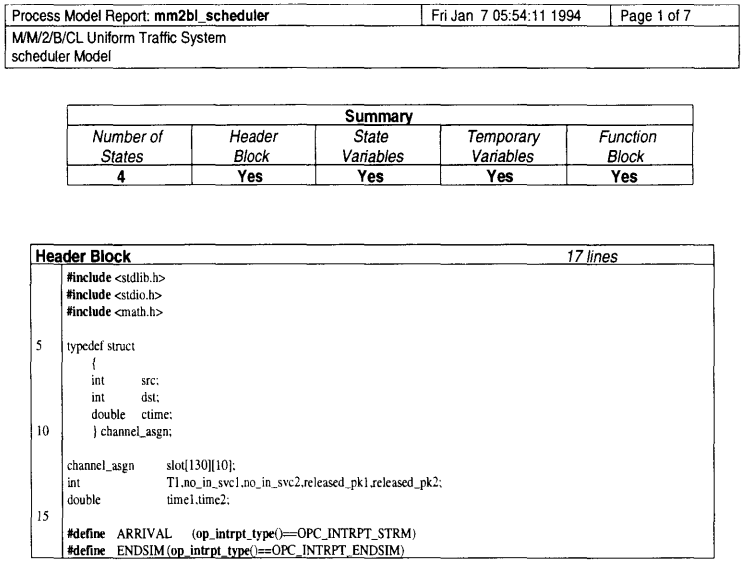
State 0: discard (Enter Execs) *unforced, 9 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm());
    op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
    if (timeslot >= T1) {no_in_svc2-=1; released_pk2+=1; time2+=(op_sim_time()-slot[timeslot][wavelength] ctime),}
    else {no_in_svc1-=1; released_pk1+=1; time1+=(op_sim_time()-slot[timeslot][wavelength].ctime);}
    slot[timeslot][wavelength].src=0; slot[timeslot][wavelength].dst=0; slot[timeslot][wavelength].ctime=0;
    op_pk_destroy(pkptr);
    }
```

State 0: discard (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**discard**

<u>*M/M/2/B/CL:*</u> OPNET report for the *scheduler* processor model. The *release* model is identical to that in M/M/2/B/L+1.



Process Model Report: **mm2bl_scheduler** | Fri Jan 7 05:54:11 1994 | Page 1 of 7

M/M/2/B/CL Uniform Traffic System
scheduler Model

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *17 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int       src;
    int       dst;
    double    ctime;
    } channel_asgn;

channel_asgn    slot[130][10];
int             T1,no_in_svc1,no_in_svc2,released_pk1,released_pk2;
double          time1,time2;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define  ENDSIM(op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

State 0: init (Enter Execs) *forced, 34 lines*

```
/* assume L1 <= L2 */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L1",&slot_no1);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L2",&slot_no2);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"s",&s);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"a",&a);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=fm_size/(slot_no2+slot_no1*a/(1-a)); T2=floor(svc_time); N2=T2*wavelength_no;
svc_time=(double)(fm_size-T2*slot_no2)/(double)slot_no1; T1=floor(svc_time); N1=T1*wavelength_no;

if (s==0) s=-1;
if ( (slot_no1-slot_no2)*s >=0) svc_time=(double)fm_size/(double)slot_no2;
else svc_time=(double)fm_size/(double)slot_no1;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);
cls_dist=op_dist_load("bernoulli",a,0);
load_prev=(a*slot_no1+(1-a)*slot_no2)*arv_rate*svc_time/(double)N;

blocked_pk1=0; blocked_pk2=0; total_pk1=0; total_pk2=0; released_pk1=0; released_pk2=0; time1=0; time2=0;
no_in_svc1=0; no_in_svc2=0; cu_steady=0; pb_steady=0; pb1_steady=0; pb2_steady=0; load_steady=0; ready=0;

for (i=0;i<(T1+T2);i++)
      for (j=0;j<wavelength_no;j++)
            {slot[i][j].src=0; slot[i][j].dst=0; slot[i][j].ctime=0;}
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 8 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

type=(int)op_dist_outcome(cls_dist);
if (type==1) total_pk1+=1; else total_pk2+=1;
pkptr=op_pk_get(op_intrpt_strm());
```

State Variables *41 lines*

```
Objid          \own_id;
Objid          \src_id;
int            \node_no;
int            \wavelength_no;
int            \fm_size;
int            \T2;
int            \slot_no;
int            \slot_no1;
int            \slot_no2;
int            \N;
int            \N1;
int            \N2;
int            \s;
double         \a;
double         \svc_time;
char           \arv_arg[20];
double         \arv_rate;
Distribution   *\src_dist;
Distribution   *\dst_dist;
Distribution   *\svc_dist;
Distribution   *\cls_dist;
long int       \blocked_pk1;
long int       \blocked_pk2;
long int       \total_pk1;
long int       \total_pk2;
double         \cu_prev;
double         \pb_prev;
double         \pb1_prev;
double         \pb2_prev;
double         \load_prev;
int            \pk_count1;
int            \pk_count2;
int            \tr_count1;
int            \tr_count2;
int            \cu_steady;
int            \pb_steady;
int            \pb1_steady;
int            \pb2_steady;
int            \load_steady;
int            \ready;
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       i,j,src,dst,type;
double    temp,ratio;
```

State 1:		**pk_prepare (CET's)**
CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: **idle (Enter Execs)** *unforced, 80 lines*

```
if (!load_steady && released_pk1>0 && released_pk2>0)
    {
    temp=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
    if ( ratio>0.999 && ratio<1.001 )
        {
        load_steady=1; ratio=0; temp=0;
        for (i=0; i<T1; i++)
            for (j=0; j<wavelength_no; j++)
                if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
        for (i=T1; i<(T1+T2); i++)
            for (j=0; j<wavelength_no; j++)
                if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
        cu_prev=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time());
        pb_prev=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2);
        pb1_prev=(double)blocked_pk1/(double)total_pk1; pb2_prev=(double)blocked_pk2/(double)total_pk2;
        temp=cu_prev*N1; pk_count1=floor(temp); temp=(double)released_pk1/(double)pk_count1; tr_count1=floor(temp)+20;
        temp=cu_prev*N2; pk_count2=floor(temp); temp=(double)released_pk2/(double)pk_count2; tr_count2=floor(temp)+20;
        }
    }

if ( load_steady && !ready && released_pk1>(tr_count1*pk_count1) && released_pk2>(tr_count2*pk_count2) )
    {
    if (!cu_steady)
        {
        ratio=0; temp=0;
        for (i=0; i<T1; i++)
            for (j=0; j<wavelength_no; j++)
                if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
        for (i=T1; i<(T1+T2); i++)
            for (j=0; j<wavelength_no; j++)
                if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
        temp=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time()); ratio=temp/cu_prev;
        if (ratio>0.999 && ratio<1.001 && temp<(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()))
            cu_steady=1;
        else
            cu_prev=temp;
        }
    if (!pb_steady)
        {
        temp=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2),
        if (pb_prev!=0) ratio=temp/pb_prev;
        else      if (temp==0) ratio=1; else ratio=0;
        if (ratio > 0.999 && ratio < 1.001) pb_steady=1; else pb_prev=temp;
        }
    if (!pb1_steady)
        {
        temp=(double)blocked_pk1/(double)total_pk1;
        if (pb1_prev!=0) ratio=temp/pb1_prev;
        else      if (temp==0) ratio=1; else ratio=0;
        if (ratio > 0.999 && ratio < 1.001) pb1_steady=1; else pb1_prev=temp;
        }
```

```
    if (!pb2_steady)
        {
        temp=(double)blocked_pk2/(double)total_pk2;
        if (pb2_prev!=0) ratio=temp/pb2_prev;
        else      if (temp==0) ratio=1; else ratio=0;
        if (ratio > 0.999 && ratio < 1.001) pb2_steady=1; else pb2_prev=temp;
        }
    if ( cu_steady && pb_steady && pb1_steady && pb2_steady ) ready=1;
    if (!ready)
            {
            ratio=0; temp=0;
            for (i=0; i<T1; i++)
                for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) temp+=(op_sim_time()-slot[i][j].ctime);
            for (i=T1; i<(T1+T2); i++)
                for (j=0; j<wavelength_no; j++)
                    if (slot[i][j].src!=0) ratio+=(op_sim_time()-slot[i][j].ctime);
            ratio=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time());
            temp=ratio*N1; pk_count1=2*floor(temp); tr_count1+=1;
            temp=ratio*N2; pk_count2=2*floor(temp); tr_count2+=1;
            }
    }

if (ready && (total_pk1+total_pk2)>20000)
    {
    temp=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
    if ( ratio > 0.995 && ratio < 1.005 && !ENDSIM)
        op_sim_end("reaching steady state","","","");
    }
```

State 2:		**idle (CET's)**
CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#2	*Exec:*	record_stats();
	Trans:	**idle**

State 3: **schedule (Enter Execs)** *forced, 11 lines*

```
if (type==1 && no_in_svc1==N1)
    {blocked_pk1+=1; op_pk_destroy(pkptr);}
else
    if (type==0 && no_in_svc2==N2)
        {blocked_pk2+=1; op_pk_destroy(pkptr);}
    else
        if (!find_resource(src,dst,pkptr,type))
            {
            op_pk_destroy(pkptr);
            if (type==1) blocked_pk1+=1; else blocked_pk2+=1;
            }
```

State 3: **schedule (CET's)**

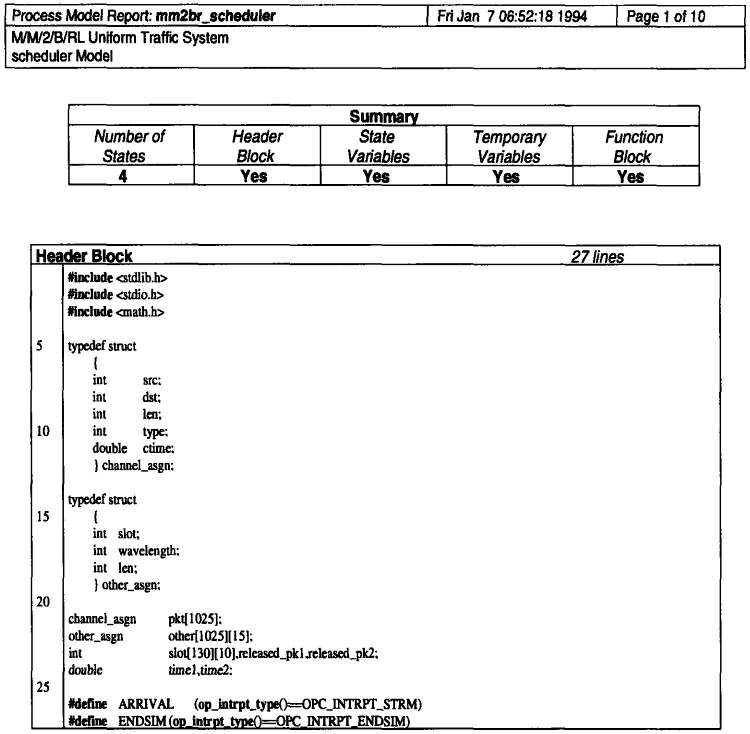
CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *103 lines*

```
record_stats()
    {
    int                     i,j;
    extern double           time1,time2;
    extern channel_asgn     slot[130][10];

    for (i=0; i<T1; i++)
        for (j=0; j<wavelength_no; j++)
            if (slot[i][j].src!=0) time1+=(op_sim_time()-slot[i][j].ctime);
    for (i=T1; i<(T1+T2); i++)
        for (j=0; j<wavelength_no; j++)
            if (slot[i][j].src!=0) time2+=(op_sim_time()-slot[i][j].ctime);
    op_stat_write_scalar("load",(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()));
  op_stat_write_scalar("cu",(slot_no1*time1+slot_no2*time2)/(N*op_sim_time()));
    op_stat_write_scalar("pb",(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2));
    op_stat_write_scalar("pb1",(double)blocked_pk1/(double)total_pk1);
    op_stat_write_scalar("pb2",(double)blocked_pk2/(double)total_pk2);
    }


int find_resource(sc,dt,packet,tp)
    int       sc,dt,tp;
    Packet*   packet;
    {
    int                     i,j,ap,fit,ps,pd,cf,cs,cd,nf,ns,nd,ca[8],na[8],as[2],high,low;
    double                  svc;
    extern int              no_in_svc1,no_in_svc2;
    extern channel_asgn     slot[130][10];

    ap=-1;
    if (tp==1)
        {low=0; high=T1; ps=-1; pd=-1;}
    else
        {
        low=T1; high=T1+T2; ps=-1; pd=-1;
        for (j=0;j<wavelength_no;j++)
            {
            if (slot[low-1][j].src==sc) ps=j;
            if (slot[low-1][j].dst==dt) pd=j;
            }
        }

    for (i=low; i<high; i++)                /* for each column */
        {
        if (i==low)
            {
            cf=0; cs=-1; cd=-1;
            for (j=0;j<wavelength_no;j++)
                {
                if (slot[i][j].src==0) { ca[cf]=j; cf+=1; }
                if (slot[i][j].src==sc) cs=j;
```

```
                if (slot[i][j].dst==dt) cd=j;
                }
            }

        if (i==(high-1) && tp==0) { ns=-1; nd=-1; }
        else
            {
            nf=0; ns=-1; nd=-1;
            for (j=0;j<wavelength_no;j++)
                {
                if (slot[i+1][j].src==0) { na[nf]=j; nf+=1; }
                if (slot[i+1][j].src==sc) ns=j;
                if (slot[i+1][j].dst==dt) nd=j;
                }
            }

        if (cf==0 || cs!=-1 || cd!=-1) goto next_col;

        for (j=0;j<cf;j++)
            {
            fit=0;
            if (ps!=-1)
                if (ca[j]==ps) fit+=1; else continue;
            if (pd!=-1)
                if (ca[j]==pd) fit+=1; else continue;
            if (ns!=-1)
                if (ca[j]==ns) fit+=1; else continue;
            if (nd!=-1)
                if (ca[j]==nd) fit+=1; else continue;
            if (fit>ap) {as[0]=i; as[1]=ca[j]; ap=fit;}
            if (ap==4) goto assign;
            }

        next_col:
        if (i!=(high-1))
            {
            ps=cs; pd=cd; cs=ns; cd=nd; cf=nf;
            for (j=0;j<nf;j++) ca[j]=na[j];
        }
        }

    if (ap==-1)
        return 0;
    else
        {
        assign:
        slot[as[0]][as[1]].src=sc; slot[as[0]][as[1]].dst=dt; slot[as[0]][as[1]].ctime=op_sim_time();
        op_pk_nfd_set(packet,"wavelength",as[1]); op_pk_nfd_set(packet,"timeslot",as[0]); svc=op_dist_outcome(svc_dist
        if (tp==1) no_in_svc1+=1; else no_in_svc2+=1;
        op_pk_send_delayed(packet,0,svc); return 1;
        }
    }
```

M/M/2/B/RL: OPNET reports for the *scheduler* and *release* processor models.



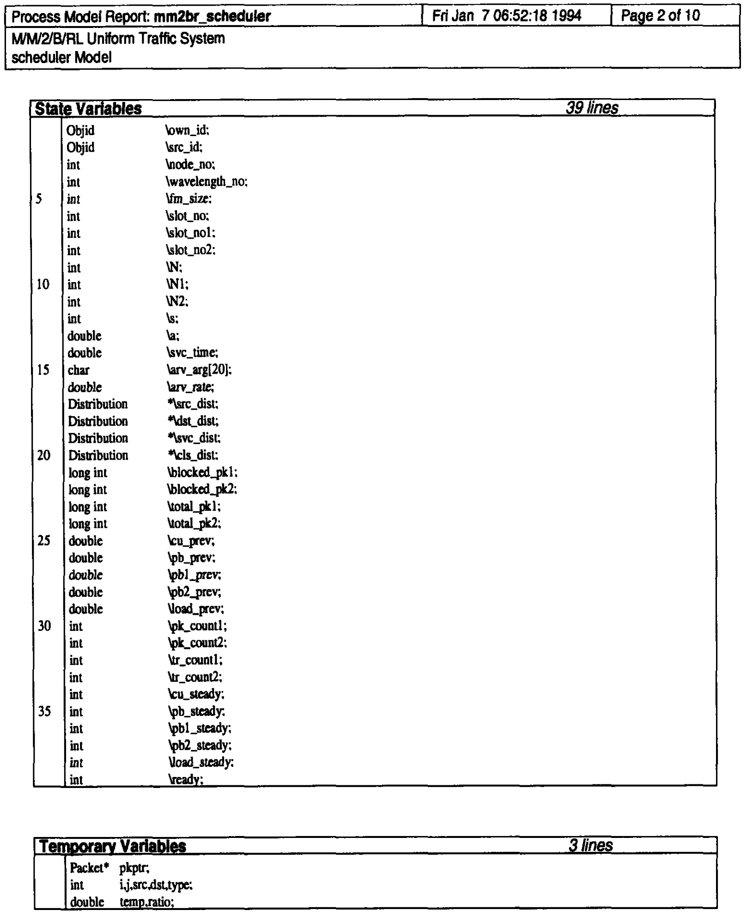
Process Model Report: **mm2br_scheduler**	Fri Jan 7 06:52:18 1994	Page 1 of 10
M/M/2/B/RL Uniform Traffic System scheduler Model		

Summary

Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *27 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
	{
	int	src;
	int	dst;
	int	len;
	int	type;
	double	ctime;
	} channel_asgn;

typedef struct
	{
	int slot;
	int wavelength;
	int len;
	} other_asgn;

channel_asgn		pkt[1025];
other_asgn		other[1025][15];
int			slot[130][10],released_pk1,released_pk2;
double			time1,time2;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define  ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```



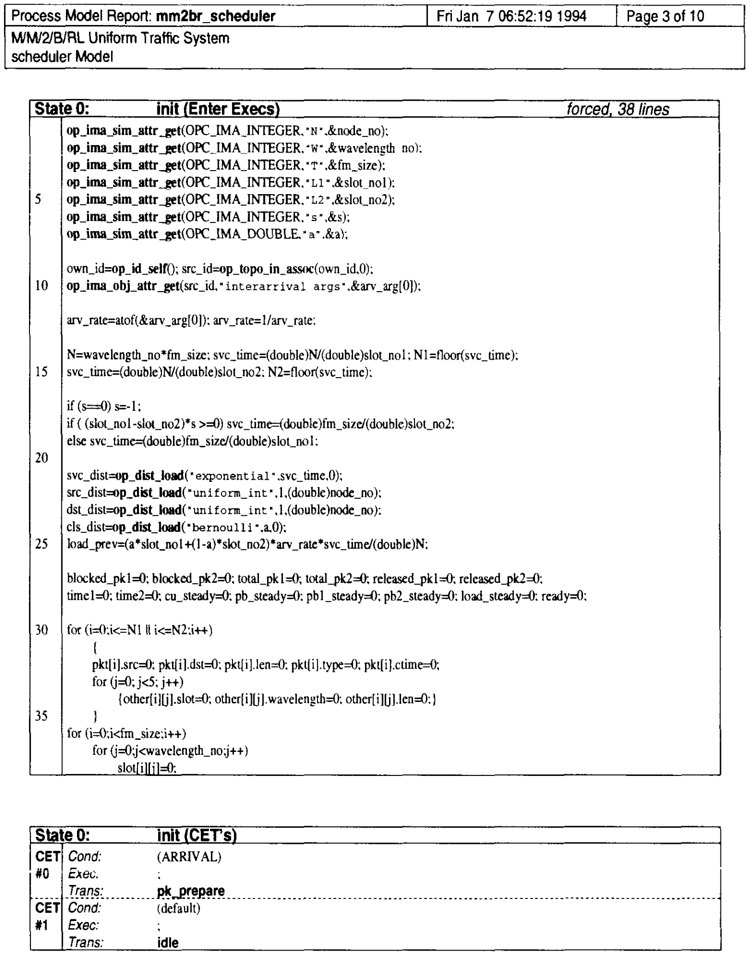
Process Model Report: **mm2br_scheduler**	Fri Jan 7 06:52:18 1994	Page 2 of 10
M/M/2/B/RL Uniform Traffic System scheduler Model		

State Variables *39 lines*

```
Objid			\own_id;
Objid			\src_id;
int			\node_no;
int			\wavelength_no;
int			\fm_size;
int			\slot_no;
int			\slot_no1;
int			\slot_no2;
int			\N;
int			\N1;
int			\N2;
int			\s;
double			\a;
double			\svc_time;
char			\arv_arg[20];
double			\arv_rate;
Distribution		*\src_dist;
Distribution		*\dst_dist;
Distribution		*\svc_dist;
Distribution		*\cls_dist;
long int		\blocked_pk1;
long int		\blocked_pk2;
long int		\total_pk1;
long int		\total_pk2;
double			\cu_prev;
double			\pb_prev;
double			\pb1_prev;
double			\pb2_prev;
double			\load_prev;
int			\pk_count1;
int			\pk_count2;
int			\tr_count1;
int			\tr_count2;
int			\cu_steady;
int			\pb_steady;
int			\pb1_steady;
int			\pb2_steady;
int			\load_steady;
int			\ready;
```

Temporary Variables *3 lines*

```
Packet*	pkptr;
int	i,j,src,dst,type;
double	temp,ratio;
```



Process Model Report: **mm2br_scheduler**	Fri Jan 7 06:52:19 1994	Page 3 of 10

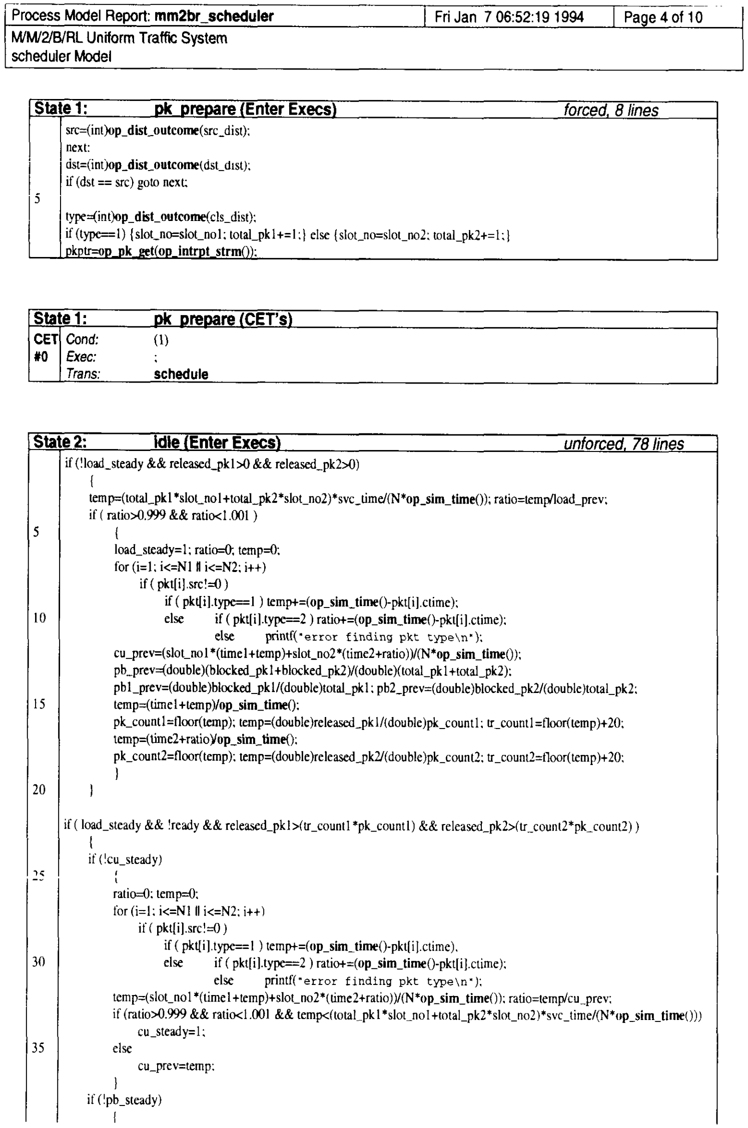
M/M/2/B/RL Uniform Traffic System
scheduler Model

State 0: **init (Enter Execs)** *forced, 38 lines*

```
op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L1",&slot_no1);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L2",&slot_no2);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"s",&s);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"a",&a);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)N/(double)slot_no1; N1=floor(svc_time);
svc_time=(double)N/(double)slot_no2; N2=floor(svc_time);

if (s==0) s=-1;
if ( (slot_no1-slot_no2)*s >=0) svc_time=(double)fm_size/(double)slot_no2;
else svc_time=(double)fm_size/(double)slot_no1;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);
cls_dist=op_dist_load("bernoulli",a,0);
load_prev=(a*slot_no1+(1-a)*slot_no2)*arv_rate*svc_time/(double)N;

blocked_pk1=0; blocked_pk2=0; total_pk1=0; total_pk2=0; released_pk1=0; released_pk2=0;
time1=0; time2=0; cu_steady=0; pb_steady=0; pb1_steady=0; pb2_steady=0; load_steady=0; ready=0;

for (i=0;i<=N1 || i<=N2;i++)
     {
     pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].type=0; pkt[i].ctime=0;
     for (j=0; j<5; j++)
          {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;}
     }
for (i=0;i<fm_size;i++)
     for (j=0;j<wavelength_no;j++)
          slot[i][j]=0;
```

State 0: **init (CET's)**

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**



Process Model Report: **mm2br_scheduler**	Fri Jan 7 06:52:19 1994	Page 4 of 10

M/M/2/B/RL Uniform Traffic System
scheduler Model

State 1: **pk_prepare (Enter Execs)** *forced, 8 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

type=(int)op_dist_outcome(cls_dist);
if (type==1) {slot_no=slot_no1; total_pk1+=1;} else {slot_no=slot_no2; total_pk2+=1;}
pkptr=op_pk_get(op_intrpt_strm());
```

State 1: **pk_prepare (CET's)**

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: **idle (Enter Execs)** *unforced, 78 lines*

```
if (!load_steady && released_pk1>0 && released_pk2>0)
     {
     temp=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
     if ( ratio>0.999 && ratio<1.001 )
          {
          load_steady=1; ratio=0; temp=0;
          for (i=1; i<=N1 || i<=N2; i++)
               if ( pkt[i].src!=0 )
                    if ( pkt[i].type==1 ) temp+=(op_sim_time()-pkt[i].ctime);
                    else      if ( pkt[i].type==2 ) ratio+=(op_sim_time()-pkt[i].ctime);
                              else      printf("error finding pkt type\n");
          cu_prev=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time());
          pb_prev=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2);
          pb1_prev=(double)blocked_pk1/(double)total_pk1; pb2_prev=(double)blocked_pk2/(double)total_pk2;
          temp=(time1+temp)/op_sim_time();
          pk_count1=floor(temp); temp=(double)released_pk1/(double)pk_count1; tr_count1=floor(temp)+20;
          temp=(time2+ratio)/op_sim_time();
          pk_count2=floor(temp); temp=(double)released_pk2/(double)pk_count2; tr_count2=floor(temp)+20;
          }
     }

if ( load_steady && !ready && released_pk1>(tr_count1*pk_count1) && released_pk2>(tr_count2*pk_count2) )
     {
     if (!cu_steady)
          {
          ratio=0; temp=0;
          for (i=1; i<=N1 || i<=N2; i++)
               if ( pkt[i].src!=0 )
                    if ( pkt[i].type==1 ) temp+=(op_sim_time()-pkt[i].ctime).
                    else      if ( pkt[i].type==2 ) ratio+=(op_sim_time()-pkt[i].ctime);
                              else      printf("error finding pkt type\n");
          temp=(slot_no1*(time1+temp)+slot_no2*(time2+ratio))/(N*op_sim_time()); ratio=temp/cu_prev;
          if (ratio>0.999 && ratio<1.001 && temp<(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()))
               cu_steady=1;
          else
               cu_prev=temp;
          }
     if (!pb_steady)
          {
```

State 3: schedule (Enter Execs) *forced, 5 lines*

```
if (!find_resource(src,dst,pkptr))
    {
    op_pk_destroy(pkptr);
    if (type==1) blocked_pk1+=1; else blocked_pk2+=1;
    }
```

State 3: schedule (CET's)

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**idle**

Function Block *231 lines*

```
record_stats()
    {
    int                     i,c1=0,c2=0;
    extern double           time1,time2;
    extern channel_asgn     pkt[1025];

    for (i=1; i<=N1 || i<=N2; i++)
        if ( pkt[i].src!=0 )
            if ( pkt[i].type==1 )           { c1+=1; time1+=(op_sim_time()-pkt[i].ctime);}
            else
                if ( pkt[i].type==2 ) { c2+=1; time2+=(op_sim_time()-pkt[i].ctime);}
                else printf("error in pkt type\n");
    op_stat_write_scalar("load",(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()));
   op_stat_write_scalar("cu",(slot_no1*time1+slot_no2*time2)/(N*op_sim_time()));
    op_stat_write_scalar("pb",(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2));
    op_stat_write_scalar("pb1",(double)blocked_pk1/(double)total_pk1);
    op_stat_write_scalar("pb2",(double)blocked_pk2/(double)total_pk2);
    }


int find_resource(sc,dt,packet)
    int         sc,dt;
    Packet*     packet;
    {
    int                     i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next;
    int                     conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
    double                  svc;
    extern int              slot[130][10];
    extern other_asgn       other[1025][15];
    extern channel_asgn     pkt[1025];

    ct=0;
    for (j=0; j<wavelength_no; j++) empty[j]=0;

    for (i=0;i<fm_size;i++)
        {

        conflict[i][0]=-1; conflict[i][1]=-1;
        for (k=0; k<wavelength_no; k++)
            {
            if ( slot[i][k]==0 ) empty[k]+=1;
            if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
```

```
            temp=(double)(blocked_pk1+blocked_pk2)/(double)(total_pk1+total_pk2);
            if (pb_prev!=0) ratio=temp/pb_prev;
            else        if (temp==0) ratio=1; else ratio=0;
            if (ratio > 0.999 && ratio < 1.001) pb_steady=1; else pb_prev=temp;
            }
        if (!pb1_steady)
            {
            temp=(double)blocked_pk1/(double)total_pk1;
            if (pb1_prev!=0) ratio=temp/pb1_prev;
            else        if (temp==0) ratio=1; else ratio=0;
            if (ratio > 0.999 && ratio < 1.001) pb1_steady=1; else pb1_prev=temp;
            }
        if (!pb2_steady)
            {
            temp=(double)blocked_pk2/(double)total_pk2;
            if (pb2_prev!=0) ratio=temp/pb2_prev;
            else        if (temp==0) ratio=1; else ratio=0;
            if (ratio > 0.999 && ratio < 1.001) pb2_steady=1; else pb2_prev=temp;
            }
        if ( cu_steady && pb_steady && pb1_steady && pb2_steady ) ready=1.
        if (!ready)
                {
                ratio=0; temp=0;
                for (i=1; i<=N1 || i<=N2; i++)
                    if ( pkt[i].src!=0 )
                        if ( pkt[i].type==1 ) temp+=(op_sim_time()-pkt[i].ctime);
                        else        if ( pkt[i].type==2 ) ratio+=(op_sim_time()-pkt[i].ctime);
                                    else        printf("error finding pkt type\n");
                temp=(time1+temp)/op_sim_time(); pk_count1=2*floor(temp); tr_count1+=1;
                temp=(time2+ratio)/op_sim_time(); pk_count2=2*floor(temp); tr_count2+=1;
                }
        }

    if (ready && (total_pk1+total_pk2)>20000)
        {
        temp=(total_pk1*slot_no1+total_pk2*slot_no2)*svc_time/(N*op_sim_time()); ratio=temp/load_prev;
        if ( ratio > 0.995 && ratio < 1.005 && !ENDSIM)
            op_sim_end("reaching steady state","","","");
        }
```

State 2: idle (CET's)

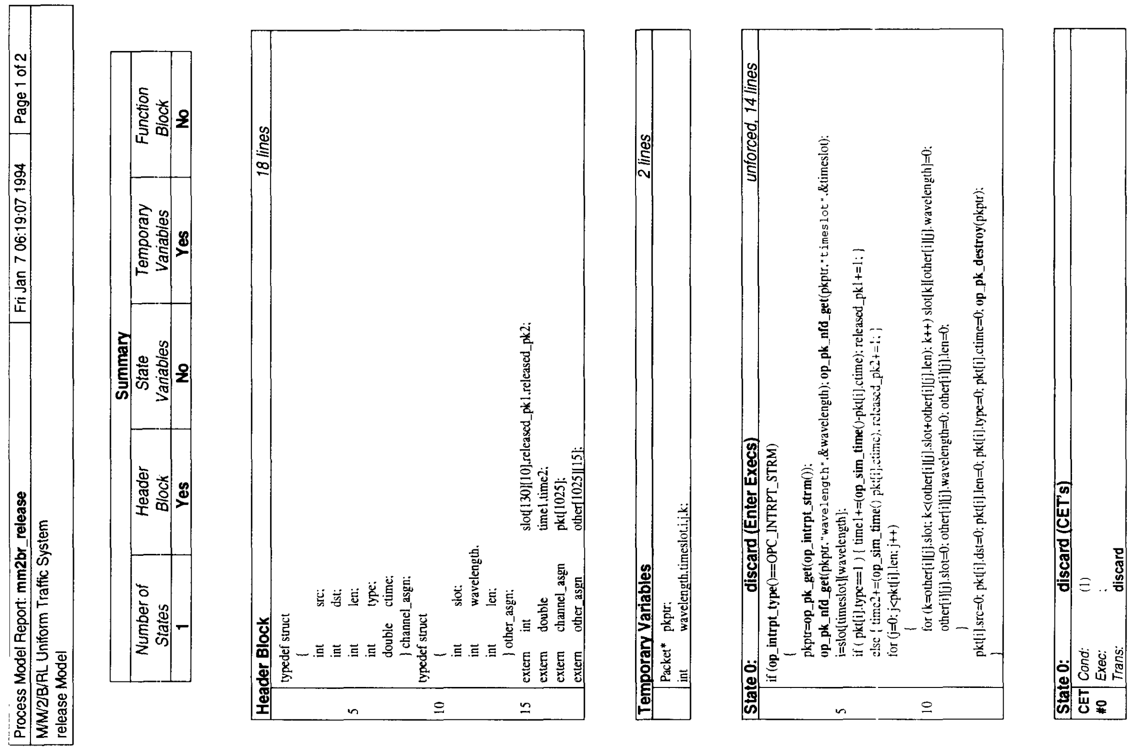
CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**
CET #2	*Cond:*	(ENDSIM)
	Exec:	record_stats();
	Trans:	**idle**

```
    if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
    }

go_back=0;
if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
    {
    for (k=0; k<wavelength_no; k++)
        {
        if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
            {
            if ( slot[i][k]!=0 ) j=i-empty[k]-1; else j=i-empty[k];
            if ( j<0 ) oh=0;
            else
                if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                else oh=2;
            if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
            insert:
            if ( ct==0 )
                {
                if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k]; else store[0][0]=i-empty[k]+1;
                store[0][1]=k; store[0][2]=empty[k]; store[0][3]=oh;
                }
            else
                for (j=(ct-1); j>=0; j--)
                    {
                    if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
                        {
                        store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
                        store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
                        }
                    if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
                        {
                        store_ct:
                        if ( slot[i][k]!=0 || go_back ) store[j+1][0]=i-empty[k]; else store[j+1][0]=i-empty[k]+1;
                        store[j+1][1]=k; store[j+1][2]=empty[k]; store[j+1][3]=oh; break;
                        }
                    if ( j==0 ) { j=-1; goto store_ct; }
                    }
            if ( go_back ) goto back;
            empty[k]=0; ct+=1;
            }
        }
    }
else
    {
    for ( k=0; k<wavelength_no; k++)
        {
        if ( empty[k]>0 )
            {
            if ( slot[i][k]==0 ) empty[k]-=1;
            if ( empty[k]==0) continue;
            j=i-empty[k]-1;
            it ( j<0 ) oh=0;
            else
                if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                else oh=2;
            if ((conflict[i][0]==-1 || conflict[i][0]==k)&&(conflict[i][1]==-1 || conflict[i][1]==k)) oh=oh.
            else        if ( oh==0 ) oh=1; else oh=3;
            if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0. continue; }
```

```
                go_back=1;
                goto insert;
                back:
                empty[k]=0, ct+=1,
                }
            }
        }
    }

nd=slot_no; i_ct=0; waste=0; next=0; asgn[0]=-1; asgn[1]=0;
for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
for (i=0; i<ct; i++)
    {
    if ( store[i][3]==0 ) fit = 0;
    else if ( store[i][3]==3 ) fit = 2; else fit = 1;
    j=store[i][2]-fit;

    if (next)
        {
        if ( j>=nd )
            {
            k=j-nd+fit;
            if ( fit<= ap1 && k<ap )
                {
                for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                    if ( rmb[oh] ) { k=-1; break; }
                if ( k!=-1 )
                    {
                    ap=k; ap1=fit; temp[i_ct]=i;
                    for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                    k=store[i][0]-1; if ( k<0 ) k=0;
                    go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                    for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1;
                    }
                }
            }
        if ( j<nd || i==(ct-1) )
            {
            waste+=ap1; i_ct+=1;
            if ( waste<asgn[0] || asgn[0]==-1 )
                {
                asgn[0]=waste; asgn[1]=i_ct;
                for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                if ( asgn[0]==0 ) goto assign;
                }
            if ( j<nd )
                {
                temp[0]=i; nd=slot_no-j; waste=fit; i_ct=1; next=0;
                for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                k=store[i][0]-1; if ( k<0 ) k=0;
                go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                }
            }
        continue;
        }

    k=0;
    for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
```

```
                if ( rmb[oh] ) { k=1; break; }
            if ( k==1 )
                continue;
            else
                {
                temp[i_ct]=i;
                k=store[i][0]-1; if ( k<0 ) k=0;
                go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                for (oh=k; oh<go_back; oh++)
                    if ( j>nd ) tmp_rmb[oh]=1; else rmb[oh]=1;
                }

            it ( j<nd ) { nd-=j; waste+=fit; i_ct+=1; continue; }
            it ( j==nd )
                {
                waste+=fit; i_ct+=1;
                if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
                nd=slot_no; i_ct=0; waste=0;
                for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                continue;
                }
            if ( j>nd ) { ap=j-nd+fit; ap1=fit; next=1; }
            if ( next && i==(ct-1) )
                {
                waste+=ap1; i_ct+=1;
                if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
                }
            }

        if(asgn[0]==-1) return 0;
        else
            {
            assign:
            for (k=1; k<=N1 || k<=N2; k++)
                {
                if (pkt[k].src!=0) continue;
                pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1];
                if (slot_no==slot_no1) pkt[k].type=1;
                else
                    if (slot_no==slot_no2 ) pkt[k].type=2; else printf("error in assign pkt type\n");
                pkt[k].ctime=op_sim_time(); nd=slot_no;
                for (i=2; i<=asgn[1]; i++)
                    {
                    oh=store[asgn[i]][0]; ct=store[asgn[i]][2]; nd-=ct;
                    if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                    else        if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                                else        if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                    for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                    other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
```

```
                        }
                    i=asgn[1]+1; oh=store[asgn[i]][0];
                    if ( store[asgn[i]][3]==3 ) oh+=1;
                    else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
                    for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
                    other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
                    op_pk_nfd_set(packet,"timeslot",oh); op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
                    svc=op_dist_outcome(svc_dist); op_pk_send_delayed(packet,0,svc); break;
                    }
                return 1;
                }
            }
```



Process Model Report: **mm2br_release**	Fri Jan 7 06:19:07 1994	Page 1 of 2
M/M/2/B/RL Uniform Traffic System release Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *18 lines*

```
typedef struct
    {
    int        src;
    int        dst;
    int        len;
    int        type;
    double     ctime;
    } channel_asgn;
typedef struct
    {
    int        slot;
    int        wavelength.
    int        len;
    } other_asgn;
extern   int              slot[130][10],released_pk1,released_pk2;
extern   double           time1,time2;
extern   channel_asgn     pkt[1025];
extern   other_asgn       other[1025][15];
```

Temporary Variables *2 lines*

```
Packet*   pkptr;
int       wavelength,timeslot,i,j,k;
```

State 0: **discard (Enter Execs)** *unforced, 14 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm());
    op_pk_nfd_get(pkptr,"wavelength",&wavelength); op_pk_nfd_get(pkptr,"timeslot",&timeslot);
    i=slot[timeslot][wavelength];
    if ( pkt[i].type==1 ) { time1+=(op_sim_time()-pkt[i].ctime); released_pk1+=1; }
    else { time2+=(op_sim_time()-pkt[i].ctime); released_pk2+=1; }
    for (j=0; j<pkt[i].len; j++)
        {
        for (k=other[i][j].slot; k<(other[i][j].slot+other[i][j].len); k++) slot[k][other[i][j].wavelength]=0;
        other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;
        }
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].type=0; pkt[i].ctime=0; op_pk_destroy(pkptr);
    }
```

State 0:		**discard (CET's)**
CET #0	*Cond:* *Exec:* *Trans:*	(1) ; **discard**

B.4.2 Queueing System

The sequential batch procedure described in Section B.3.2 is used as termination criteria for the *M/M/2/Q/RL* simulation. The observation was made on the average queue size for both types of sessions (Q_1 & Q_2) after the measured load comes to within 1% of the offered load obtained in Section 4.1. When the 90% confidence interval for Q_1 & Q_2 comes to within 10% of the value of Q_1 & Q_2 respectively, the simulation will end. The simulation will also be terminated after it has been running for 2,000,000 seconds.

The seed used for each simulation has value *NWs*, where *N* is the number of users, *W* the wavelength channels, and *s* equals to one if the offered load is normalized against sessions with smaller of the two throughput requirements and zero otherwise (refer to Section 4.1). So for *N* equals to forty, *W* equals to eight and *s* equals to one, the seed is 4081.

M/M/2/Q/RL: OPNET Reports for the *scheduler* and *release* processor models.

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *32 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
      {
      int        src;
      int        dst;
      int        len;
      int        type;
      double     ctime;
      } channel_asgn;

typedef struct
      {
      int   slot;
      int   wavelength;
      int   len;
      } other_asgn;

channel_asgn        pkt[1025];
other_asgn          other[1025][15];
int                 no_in_svc,slot_no1,slot_no2,slot[130][10];
double              time[2];

#define   ARRIVAL     (op_intrpt_type()==OPC_INTRPT_STRM)
#define   ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define EMPTYQ  (op_subq_empty(0) && op_subq_empty(1))
#define NEXTQ    (op_intrpt_type()==OPC_INTRPT_REMOTE && !EMPTYQ)
#define NEXTQ1   (op_intrpt_type()==OPC_INTRPT_REMOTE && op_intrpt_code()==1 && !op_subq_empty(1))
#define NEXTQ2   (op_intrpt_type()==OPC_INTRPT_REMOTE && op_intrpt_code()==2 && !op_subq_empty(0))
#define   MEASURE (op_intrpt_type()==OPC_INTRPT_SELF && op_intrpt_code()==0)
```

State Variables *34 lines*

```
Objid            \own_id;
Objid            \src_id;
int              \node_no;
int              \wavelength_no;
int              \fm_size;
int              \N;
int              \N1;
int              \N2;
int              \sn;
double           \a;
double           \svc_time;
char             \arv_arg[20];
double           \arv_rate;
Distribution     *\src_dist;
Distribution     *\dst_dist;
Distribution     *\svc_dist;
Distribution     *\cls_dist;
long int         \total_pk[2];
long int         \processed_pk[2];
long int         \stat_ct;
long int         \next_stat;
long int         \n0;
long int         \n1;
double           \load_prev;
double           \stat[200000][2];
double           \total_time[2];
double           \delay[2];
double           \tstart[2];
double           \qsize[2];
double           \queue[2];
double           \inc;
double           \z;
int              \load_steady;
int              \end_sim;
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       i,j,src,dst,type;
double    temp,ratio;
```

Process Model Report: **mm2qr_scheduler**	Fri Jan 7 06:50:30 1994	Page 3 of 13
M/M/2/Q/RL Uniform Traffic System scheduler Model		

State 0: **init (Enter Execs)** *forced, 40 lines*

```
f_inverse(0.1,&z);        /* 90% confidence interval */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L1",&slot_no1).
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L2",&slot_no2);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"s",&sn);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"a",&a);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)N/(double)slot_no1; N1=floor(svc_time);
svc_time=(double)N/(double)slot_no2; N2=floor(svc_time);

if (sn==0) sn=-1;
if ( (slot_no1-slot_no2)*sn >=0) svc_time=(double)fm_size/(double)slot_no2;
else svc_time=(double)fm_size/(double)slot_no1;

svc_dist=op_dist_load("exponential",svc_time,0);
src_dist=op_dist_load("uniform_int",1,(double)node_no);
dst_dist=op_dist_load("uniform_int",1,(double)node_no);
cls_dist=op_dist_load("bernoulli",a,0);

inc=10; no_in_svc=0; load_prev=(a*slot_no1+(1-a)*slot_no2)*arv_rate*svc_time/(double)N;
load_steady=0; end_sim=0; n0=600; n1=800; op_subq_flush(0); op_subq_flush(1);

for (i=0; i<2; i++) {total_pk[i]=0; processed_pk[i]=0; time[i]=0; total_time[i]=0; delay[i]=0;}
for (i=0;i<=N1 || i<=N2;i++)
    {
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].type=0; pkt[i].ctime=0;
    for (j=0; j<15; j++)
        {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;}
    }
for (i=0;i<fm_size;i++)
    for (j=0;j<wavelength_no;j++)
        slot[i][j]=0;
```

State 0: **init (CET's)**

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**

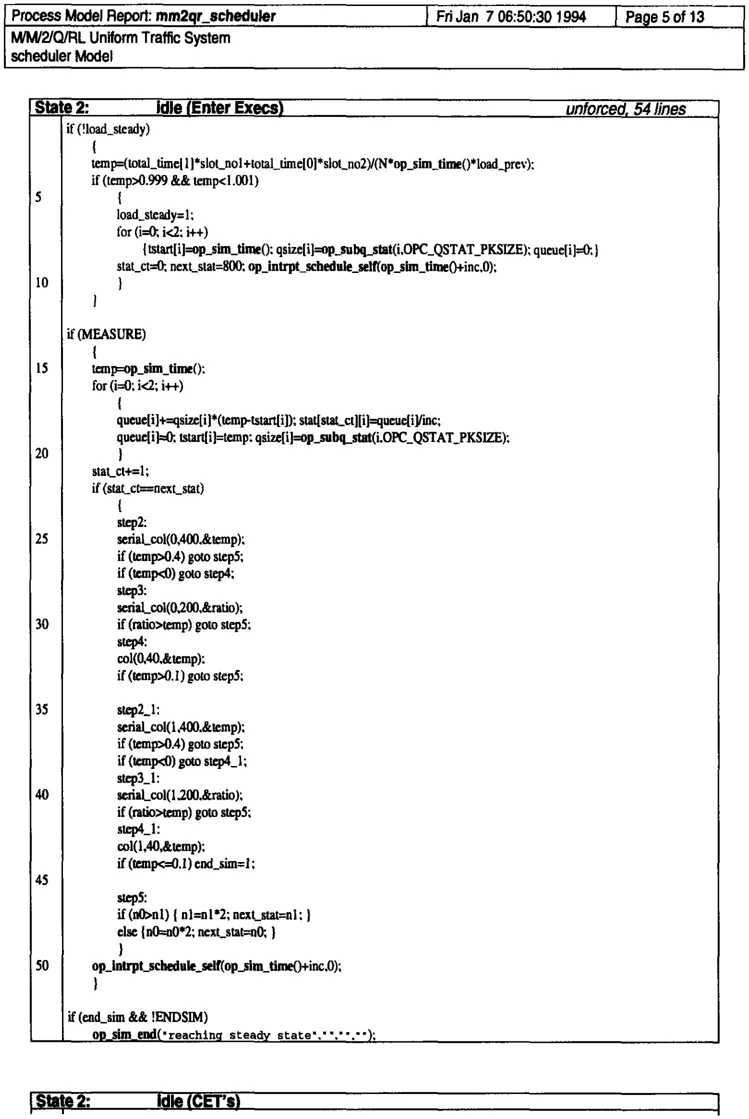
Process Model Report: **mm2qr_scheduler**	Fri Jan 7 06:50:30 1994	Page 4 of 13
M/M/2/Q/RL Uniform Traffic System scheduler Model		

State 1: **pk_prepare (Enter Execs)** *forced, 10 lines*

```
src=(int)op_dist_outcome(src_dist);
next:
dst=(int)op_dist_outcome(dst_dist);
if (dst == src) goto next;

type=(int)op_dist_outcome(cls_dist); temp=op_dist_outcome(svc_dist);
if (type==1) {total_pk[1]+=1; total_time[1]+=temp;} else {total_pk[0]+=1; total_time[0]+=temp.}

pkptr=op_pk_get(op_intrpt_strm()); op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst);
op_pk_nfd_set(pkptr,"svc",temp); op_pk_nfd_set(pkptr,"type",type);
```

State 1: **pk_prepare (CET's)**

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

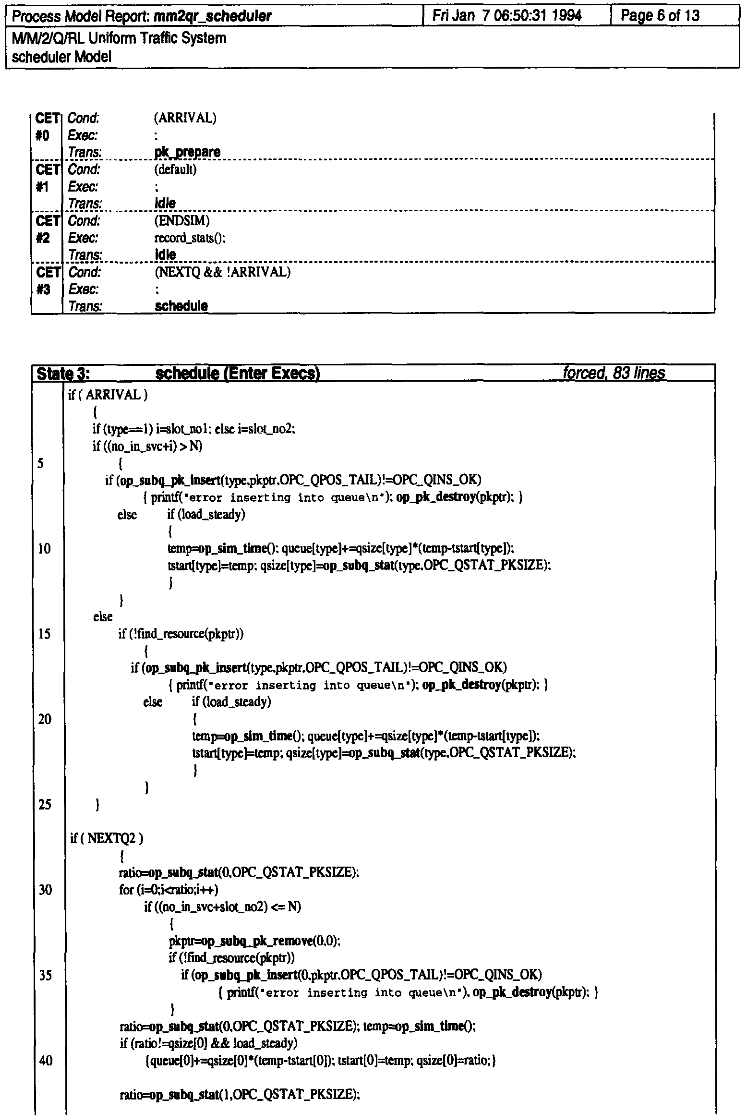


Process Model Report: **mm2qr_scheduler**	Fri Jan 7 06:50:30 1994	Page 5 of 13
M/M/2/Q/RL Uniform Traffic System scheduler Model		

State 2: **idle (Enter Execs)** *unforced, 54 lines*

```
if (!load_steady)
    {
    temp=(total_time[1]*slot_no1+total_time[0]*slot_no2)/(N*op_sim_time()*load_prev);
    if (temp>0.999 && temp<1.001)
        {
        load_steady=1;
        for (i=0; i<2; i++)
            {tstart[i]=op_sim_time(); qsize[i]=op_subq_stat(i,OPC_QSTAT_PKSIZE); queue[i]=0;}
        stat_ct=0; next_stat=800; op_intrpt_schedule_self(op_sim_time()+inc,0);
        }
    }

if (MEASURE)
    {
    temp=op_sim_time();
    for (i=0; i<2; i++)
        {
        queue[i]+=qsize[i]*(temp-tstart[i]); stat[stat_ct][i]=queue[i]/inc;
        queue[i]=0; tstart[i]=temp; qsize[i]=op_subq_stat(i,OPC_QSTAT_PKSIZE);
        }
    stat_ct+=1;
    if (stat_ct==next_stat)
        {
        step2:
        serial_col(0,400,&temp);
        if (temp>0.4) goto step5;
        if (temp<0) goto step4;
        step3:
        serial_col(0,200,&ratio);
        if (ratio>temp) goto step5;
        step4:
        col(0,40,&temp);
        if (temp>0.1) goto step5;

        step2_1:
        serial_col(1,400,&temp);
        if (temp>0.4) goto step5;
        if (temp<0) goto step4_1;
        step3_1:
        serial_col(1,200,&ratio);
        if (ratio>temp) goto step5;
        step4_1:
        col(1,40,&temp);
        if (temp<=0.1) end_sim=1;

        step5:
        if (n0>n1) { n1=n1*2; next_stat=n1; }
        else {n0=n0*2; next_stat=n0; }
        }
    op_intrpt_schedule_self(op_sim_time()+inc,0);
    }

if (end_sim && !ENDSIM)
    op_sim_end("reaching steady state","","","");
```

State 2: **idle (CET's)**



Process Model Report: **mm2qr_scheduler**	Fri Jan 7 06:50:31 1994	Page 6 of 13
M/M/2/Q/RL Uniform Traffic System scheduler Model		

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#2	*Exec:*	record_stats();
	Trans:	**idle**
CET	*Cond:*	(NEXTQ && !ARRIVAL)
#3	*Exec:*	;
	Trans:	**schedule**

State 3: **schedule (Enter Execs)** *forced, 83 lines*

```
if ( ARRIVAL )
    {
    if (type==1) i=slot_no1; else i=slot_no2;
    if ((no_in_svc+i) > N)
        {
      if (op_subq_pk_insert(type,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
            { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
        else     if (load_steady)
                {
                temp=op_sim_time(); queue[type]+=qsize[type]*(temp-tstart[type]);
                tstart[type]=temp; qsize[type]=op_subq_stat(type,OPC_QSTAT_PKSIZE);
                }
        }
    else
        if (!find_resource(pkptr))
            {
          if (op_subq_pk_insert(type,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
            else     if (load_steady)
                    {
                    temp=op_sim_time(); queue[type]+=qsize[type]*(temp-tstart[type]);
                    tstart[type]=temp; qsize[type]=op_subq_stat(type,OPC_QSTAT_PKSIZE);
                    }
            }
    }

if ( NEXTQ2 )
        {
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if ((no_in_svc+slot_no2) <= N)
                {
                pkptr=op_subq_pk_remove(0,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                }
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize[0] && load_steady)
            {queue[0]+=qsize[0]*(temp-tstart[0]); tstart[0]=temp; qsize[0]=ratio;}

        ratio=op_subq_stat(1,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if ((no_in_svc+slot_no1) <= N)
                {
                pkptr=op_subq_pk_remove(1,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(1,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr), }
                }
        ratio=op_subq_stat(1,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize[1] && load_steady)
            {queue[1]+=qsize[1]*(temp-tstart[1]); tstart[1]=temp; qsize[1]=ratio;}
        }

if ( NEXTQ1 )
        {
        ratio=op_subq_stat(1,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if ((no_in_svc+slot_no1) <= N)
                {
                pkptr=op_subq_pk_remove(1,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(1,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                }
        ratio=op_subq_stat(1,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize[1] && load_steady)
            {queue[1]+=qsize[1]*(temp-tstart[1]); tstart[1]=temp; qsize[1]=ratio;}

        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
        for (i=0;i<ratio;i++)
            if ((no_in_svc+slot_no2) <= N)
                {
                pkptr=op_subq_pk_remove(0,0);
                if (!find_resource(pkptr))
                  if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                        { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                }
        ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE); temp=op_sim_time();
        if (ratio!=qsize[0] && load_steady)
            {queue[0]+=qsize[0]*(temp-tstart[0]); tstart[0]=temp; qsize[0]=ratio;}
        }
```

State 3:		**schedule (CET's)**
CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *338 lines*

```
record_stats()
    {
    int                 i,j,b,t;
    double              x,s,y[40],xx[2],ss[2];
    extern int          slot_no1,slot_no2,slot[130][10];
    extern double       time[2];
    extern channel_asgn pkt[1025];
```

```
    for (t=0; t<2; t++)
        {
        s=(double)stat_ct/(double)40; b=floor(s), x=0,
        for (i=0;i<40;i++)
            {
            y[i]=0;
            for (j=0;j<b;j++)          y[i]+=stat[i*b+j][t];
            y[i]=y[i]/b; x+=y[i];
            }
        x=x/40; s=0;
        for (i=0;i<40;i++) { j=y[i]-x; s+=j*j; }
        s=s/39;
        if (x!=0) s=z*sqrt(s/40)/x;
        xx[t]=x, ss[t]=s;
        }

    for (i=1; i<=N1 || i<=N2; i++)
        if ( pkt[i].src!=0 )
            if ( pkt[i].type==1 )          time[1]+=(op_sim_time()-pkt[i].ctime);
            else        time[0]+=(op_sim_time()-pkt[i].ctime);

    op_stat_write_scalar("load",(total_time[1]*slot_no1+total_time[0]*slot_no2)/(N*op_sim_time()));
    op_stat_write_scalar("cu",(time[1]*slot_no1+time[0]*slot_no2)/(N*op_sim_time()));
    op_stat_write_scalar("a",(double)total_pk[1]/(double)(total_pk[0]+total_pk[1]));
    op_stat_write_scalar("Q1",xx[1]);
    op_stat_write_scalar("Q2",xx[0]);
    op_stat_write_scalar("D1",delay[1]/(double)processed_pk[1]);
    op_stat_write_scalar("D2",delay[0]/(double)processed_pk[0]);
    op_stat_write_scalar("qd1",ss[1]);
    op_stat_write_scalar("qd2",ss[0]);
    op_stat_write_scalar("c",stat_ct);
    }



f_inverse (x,z1)
    double    x,*z1;
    {
    double    y,p,p0,p1,p2,p3,p4,q0,q1,q2,q3,q4;

    x=1-x/2;
    p0=-0.322232431088e0; p1=-1.0; p2=-0.342242088547e0; p3=-0.0204231210245e0; p4=-0.453642210148e-4;
    q0=0.099348462606e0; q1=0.588581570495e0; q2=0.531103462366e0; q3=0.10353775285e0; q4=0.38560700634e-2;

    p=x;
    if (p>0.5) p=1.0-p;
    y=sqrt(-log(p*p));
    *z1=y+(p0+y*(p1+y*(p2+y*(p3+y*p4))))/(q0+y*(q1+y*(q2+y*(q3+y*q4))));
    if (x<0.5) *z1=-*z1;
    }


serial_col(t,n,p1)
    int         t,n;
    double      *p1,
    {
    int   i,j,k,b;
    double    p,x,x1,x2,c,c1,c2,s,s1,s2,y[400];
```

```
	b=next_stat/n; k=n/2; x=0; x1=0; x2=0;
	for (i=0;i<n;i++)
		{
		y[i]=0;
		for (j=0;j<b;j++)		y[i]+=stat[i*b+j][t];
		y[i]=y[i]/b; x+=y[i];
		if (i<k) x1+=y[i]; else x2+=y[i];
		}
	x=x/n; x1=x1/k; x2=x2/k; s=0; s1=0; s2=0; c=0; c1=0; c2=0;
	for (i=0;i<n;i++)
		{
		j=y[i]-x; s+=j*j;
		if (i<(n-1)) c+=j*(y[i+1]-x);
		if (i<k)
			{
			j=y[i]-x1; s1+=j*j;
			if (i<(k-1)) c1+=j*(y[i+1]-x1);
			}
		else
			{
			j=y[i]-x2; s2+=j*j;
			if (i<(n-1)) c2+=j*(y[i+1]-x2);
			}
		}
	if (s==0) *p1=0; else *p1=2*c/s-(c1/s1+c2/s2)/2;
	}


col(t,n,p1)
	int		t,n;
	double	*p1;
	{
	int	i,j,b;
	double	p,x,s,y[400];

	b=next_stat/n; x=0;
	for (i=0;i<n;i++)
		{
		y[i]=0;
		for (j=0;j<b;j++)		y[i]+=stat[i*b+j][t];
		y[i]=y[i]/b; x+=y[i];
		}
	x=x/n; s=0;
	for (i=0;i<n;i++) { j=y[i]-x; s+=j*j; }
	s=s/(n-1); s=sqrt(s/n);
	if (s==0) *p1=0; else *p1=z*s/x;
	printf("\n%d(%d): %f\t%f",stat_ct,t,x,*p1);
	}


int find_resource(packet)
	Packet*	packet;
	{
	int				i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next,sc,dt,tp;
	int				conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
	double			svc;
	extern int		no_in_svc,slot_no1,slot_no2,slot[130][10];
	extern other_asgn	other[1025][15];
```

```
	extern channel_asgn	pkt[1025];

	op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst",&dt);
	op_pk_nfd_get(packet,"svc",&svc); op_pk_nfd_get(packet,"type",&tp);

	ct=0;
	for (j=0; j<wavelength_no; j++) empty[j]=0;

	for (i=0;i<fm_size;i++)
		{

		conflict[i][0]=-1; conflict[i][1]=-1;
		for (k=0; k<wavelength_no; k++)
			{
			if ( slot[i][k]==0 ) empty[k]+=1;
			if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
			if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
			}

		go_back=0;
		if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
			{
			for (k=0; k<wavelength_no; k++)
				{
				if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
					{
					if ( slot[i][k]!=0 ) j=i-empty[k]-1; else j=i-empty[k];
					if ( j<0 ) oh=0;
					else
						if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
						else oh=2;
					if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
					insert:
					if ( ct==0 )
						{
						if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k]; else store[0][0]=i-empty[k]+1;
						store[0][1]=k; store[0][2]=empty[k]; store[0][3]=oh;
						}
					else
						for (j=(ct-1); j>=0; j--)
							{
							if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
								{
								store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
								store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
								}
							if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
								{
								store_ct:
								if (slot[i][k]!=0 || go_back) store[j+1][0]=i-empty[k]; else store[j+1][0]=i-empty[k]+1;
								store[j+1][1]=k;  store[j+1][2]=empty[k];  store[j+1][3]=oh;
								break;
								}
							if ( j==0 ) { j=-1; goto store_ct; }
							}
					if ( go_back ) goto back;
					empty[k]=0; ct+=1;
					}
				}
```

```
                }
            else
                {
                for ( k=0; k<wavelength_no; k++)
                    {
                    if ( empty[k]>0 )
                        {
                        if ( slot[i][k]==0 ) empty[k]-=1;
                        if ( empty[k]==0) continue;
                        j=i-empty[k]-1;
                        if ( j<0 ) oh=0;
                        else
                            if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                            else oh=2;
                        if ((conflict[i][0]==-1 || conflict[i][0]==k)&&(conflict[i][1]==-1 || conflict[i][1]==k)) oh=oh;
                        else        if ( oh==0 ) oh=1; else oh=3;
                        if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0; continue; }
                        go_back=1;
                        goto insert;
                        back:
                        empty[k]=0; ct+=1;
                        }
                    }
                }
            }

        if (tp==1) nd=slot_no1; else nd=slot_no2;
        i_ct=0; waste=0; next=0; asgn[0]=-1; asgn[1]=0;
        for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
        for (i=0; i<ct; i++)
            {
            if ( store[i][3]==0 ) fit = 0;
            else if ( store[i][3]==3 ) fit = 2; else fit = 1;
            j=store[i][2]-fit;

            if (next)
                {
                if ( j>=nd )
                    {
                    k=j-nd+fit;
                    if ( fit<= ap1 && k<ap )
                        {
                        for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                            if ( rmb[oh] ) { k=-1; break; }
                        if ( k!=-1 )
                            {
                            ap=k; ap1=fit; temp[i_ct]=i;
                            for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                            k=store[i][0]-1; if ( k<0 ) k=0;
                            go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                            for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1;
                            }
                        }
                    }
                if ( j<nd || i==(ct-1) )
                    {
                    waste+=ap1; i_ct+=1;
                    if ( waste<asgn[0] || asgn[0]==-1 )
                        {
```

```
                        asgn[0]=waste; asgn[1]=i_ct;
                        for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                        if ( asgn[0]==0 ) goto assign;
                        }
                    if ( j<nd )
                        {
                        temp[0]=i; waste=fit; i_ct=1; next=0;
                        if (tp==1) nd=slot_no1-j; else nd=slot_no2-j;
                        for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                        k=store[i][0]-1; if ( k<0 ) k=0;
                        go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                        for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                        }
                    }
                continue;
                }

            k=0;
            for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                if ( rmb[oh] ) { k=1; break; }
            if ( k==1 ) continue;
            else
                {
                temp[i_ct]=i;
                k=store[i][0]-1; if ( k<0 ) k=0;
                go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                for (oh=k; oh<go_back; oh++)
                    if ( j>nd ) tmp_rmb[oh]=1; else rmb[oh]=1;
                }

            if ( j<nd ) { nd-=j; waste+=fit; i_ct+=1; continue; }
            if ( j==nd )
                {
                waste+=fit; i_ct+=1;
                if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
                i_ct=0; waste=0; if (tp==1) nd=slot_no1; else nd=slot_no2;
                for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                continue;
                }
            if ( j>nd ) { ap=j-nd+fit; ap1=fit; next=1; }
            if ( next && i==(ct-1) )
                {
                waste+=ap1; i_ct+=1;
                if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
                }
            }

        if(asgn[0]==-1) return 0;
        else
```

Process Model Report: **mm2qr_release**	Fri Jan 7 06:28:22 1994	Page 1 of 2
M/M/2/Q/RL Uniform Traffic System release Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
1	**Yes**	**No**	**Yes**	**No**

Header Block *18 lines*

```
typedef struct
    {
    int        src;
    int        dst;
    int        len;
    int        type;
    double     ctime;
    } channel_asgn;
typedef struct
    {
    int        slot;
    int        wavelength;
    int        len;
    } other_asgn;
extern    int             no_in_svc,slot_no1,slot_no2,slot[130][10];
extern    double          time[2];
extern    channel_asgn    pkt[1025];
extern    other_asgn      other[1025][15];
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       wavelength,timeslot,i,j,k,type;
Objid     own_id,src_id;
```

State 0: discard (Enter Execs) *unforced, 13 lines*

```
if (op_intrpt_type()==OPC_INTRPT_STRM)
    {
    pkptr=op_pk_get(op_intrpt_strm()); op_pk_nfd_get(pkptr,"wavelength",&wavelength);
    op_pk_nfd_get(pkptr,"timeslot",&timeslot); i=slot[timeslot][wavelength];
    if ( pkt[i].type==1 ) { time[1]+=(op_sim_time()-pkt[i].ctime); no_in_svc-=slot_no1; type=1; }
    else { time[0]+=(op_sim_time()-pkt[i].ctime); no_in_svc-=slot_no2; type=2; }
    for (j=0; j<pkt[i].len; j++)
        { for (k=other[i][j].slot; k<(other[i][j].slot+other[i][j].len); k++) slot[k][other[i][j].wavelength]=0;
        other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0; }
    pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].type=0; pkt[i].ctime=0; op_pk_destroy(pkptr);
    own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
    op_intrpt_schedule_remote(op_sim_time(),type,src_id);
    }
```

State 0: discard (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**discard**

Process Model Report: **mm2qr_scheduler**	Fri Jan 7 06:50:34 1994	Page 13 of 13
M/M/2/Q/RL Uniform Traffic System scheduler Model		

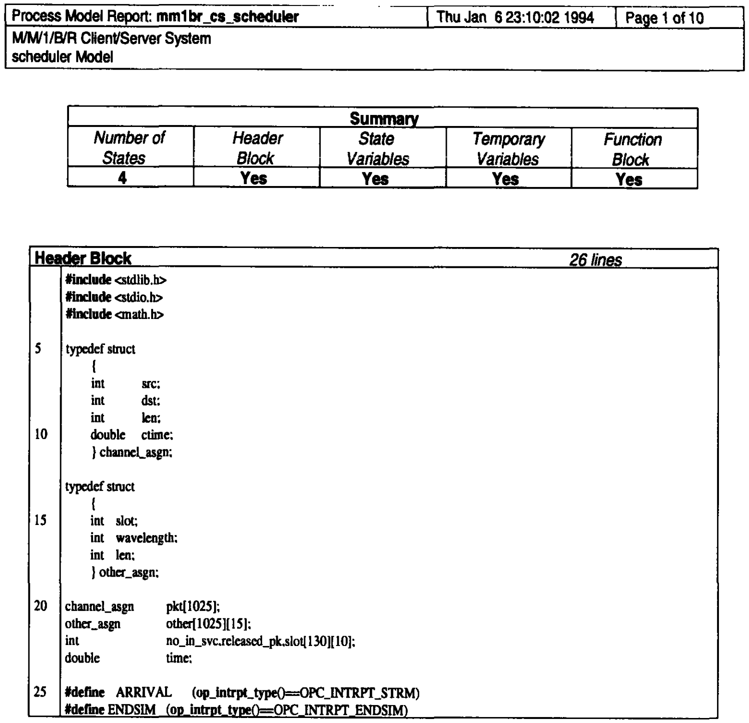
```
        {
        assign:
        for (k=1; k<=N1 || k<=N2; k++)
            {
            if (pkt[k].src!=0) continue;
            pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1]; pkt[k].type=tp; pkt[k].ctime=op_sim_time();
            if (tp==1)
                {
                nd=slot_no1; no_in_svc+=slot_no1; processed_pk[1]+=1;
                delay[1]+=op_sim_time()-op_pk_creation_time_get(packet);
                }
            else
                {
                nd=slot_no2; no_in_svc+=slot_no2; processed_pk[0]+=1;
                delay[0]+=op_sim_time()-op_pk_creation_time_get(packet);
                }
            for (i=2; i<=asgn[1]; i++)
                {
                oh=store[asgn[i]][0]; ct=store[asgn[i]][2]. nd-=ct;
                if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                else        if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                            else        if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
                }
            i=asgn[1]+1; oh=store[asgn[i]][0];
            if ( store[asgn[i]][3]==3 ) oh+=1;
            else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
            for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
            other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
            op_pk_nfd_set(packet,"timeslot",oh); op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
            op_pk_send_delayed(packet,0,svc); break;
            }
        return 1;
        }
    }
```

B.5 Single Class, Client/Server System

B.5.1 Blocking System

The termination criteria and the seed used here is the same as that in Section B.3.1.

<u>M/M/1/B/RL:</u> OPNET report for the *scheduler* processor model. The *release* model is identical to that in M/M/1/B/RL uniform traffic system.



Process Model Report: **mm1br_cs_scheduler**	Thu Jan 6 23:10:02 1994	Page 1 of 10
M/M/1/B/R Client/Server System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *26 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int         src;
    int         dst;
    int         len;
    double      ctime;
    } channel_asgn;

typedef struct
    {
    int   slot;
    int   wavelength;
    int   len;
    } other_asgn;

channel_asgn        pkt[1025];
other_asgn          other[1025][15];
int                 no_in_svc.released_pk.slot[130][10];
double              time;

#define  ARRIVAL    (op_intrpt_type()==OPC_INTRPT_STRM)
#define ENDSIM  (op_intrpt_type()==OPC_INTRPT_ENDSIM)
```

M/M/1/B/R Client/Server System
scheduler Model

State Variables *38 lines*

```
Objid           \own_id;
Objid           \src_id;
int             \node_no;
int             \server_no;
int             \wavelength_no;
int             \fm_size;
int             \T1;
int             \slot_no;
int             \N;
int             \N1;
double          \traffic;
double          \svc_time;
char            \arv_arg[20];
double          \arv_rate;
Distribution    *\tfc_dist;
Distribution    *\srv_dist;
Distribution    *\src_dist;
Distribution    *\dst_dist;
Distribution    *\svc_dist;
long int        \total_pks;
long int        \total_pku;
long int        \blocked_pks;
long int        \blocked_pku;
long int        \blocked_c_pks;
long int        \blocked_c_pku;
double          \cu_max;
double          \cu_prev;
double          \pb_prev;
double          \pbc_prev;
double          \load_prev;
double          \total_time;
int             \pk_count;
int             \tr_count;
int             \cu_steady;
int             \pb_steady;
int             \pbc_steady;
int             \load_steady;
int             \ready;
```

Temporary Variables *3 lines*

```
Packet*  pkptr;
int      src,dst,i,j;
double   ratio,temp;
```

M/M/1/B/R Client/Server System
scheduler Model

State 0: **init (Enter Execs)** *forced, 41 lines*

```
op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"Ns",&server_no);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"tp",&traffic);

own_id=op_id_self();
src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]);
arv_rate=1/arv_rate;

N=wavelength_no*fm_size;
svc_time=(double)N/(double)slot_no;
N1=floor(svc_time);
svc_time=(double)fm_size/(double)slot_no;

svc_dist=op_dist_load("exponential",svc_time,0);
tfc_dist=op_dist_load("bernoulli",(double)(2*traffic*server_no),0);
srv_dist=op_dist_load("uniform_int",1,(double)(2*server_no));
src_dist=op_dist_load("uniform_int",(double)(1+server_no),(double)node_no);
dst_dist=op_dist_load("uniform_int",(double)(1+server_no),(double)node_no);

total_pks=0;          total_pku=0;
blocked_pks=0;        blocked_pku=0;
blocked_c_pks=0;      blocked_c_pku=0;
released_pk=0;        no_in_svc=0;       time=0;
cu_max=(double)(N1*slot_no)/(double)N;
load_prev=arv_rate/(double)wavelength_no;
cu_steady=0; pb_steady=0; pbc_steady=0; load_steady=0; ready=0;

for (i=0;i<=N1;i++)
     {
     pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0;
     for (j=0; j<15; j++)
          {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0; }
     }
for (i=0;i<fm_size;i++)
     for (j=0;j<wavelength_no;j++) slot[i][j]=0;
```

State 0: **init (CET's)**

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 1: pk_prepare (Enter Execs) *forced, 19 lines*

```
temp=(int)op_dist_outcome(tfc_dist);
if (temp)        /* client-server traffic */
      {
      ratio=(int)op_dist_outcome(srv_dist);
      if (ratio<=server_no)
            {src=ratio; dst=(int)op_dist_outcome(dst_dist); total_pks+=1;}
      else
            {dst=ratio-server_no; src=(int)op_dist_outcome(src_dist); total_pks+=1;}
      }
else
      {
      src=(int)op_dist_outcome(src_dist);
      next:
      dst=(int)op_dist_outcome(dst_dist);
      if (dst == src) goto next;
      total_pku+=1;
      }
temp=op_dist_outcome(svc_dist); total_time+=temp; pkptr=op_pk_get(op_intrpt_strm());
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);
```

State 1: pk_prepare (CET's)

CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**schedule**

State 2: idle (Enter Execs) *unforced, 64 lines*

```
if (!load_steady && released_pk>0)
      {
      temp=slot_no*total_time/(N*op_sim_time());
      ratio=temp/load_prev;
      temp=(double)total_pks/(double)((total_pks+total_pku)*2*server_no*traffic);
      if ( ratio > 0.999 && ratio < 1.001 && temp > 0.999 && temp < 1.001)
            {
            load_steady=1; ratio=0;
            for (i=1; i<=N1; i++)
                  if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
            cu_prev=(time+ratio)*slot_no/(N*op_sim_time());
            pb_prev=(double)(blocked_pks+blocked_pku)/(double)(total_pks+total_pku);
            pbc_prev=(double)(blocked_c_pks+blocked_c_pku)/(double)(total_pks+total_pku);
            temp=cu_prev*N1; pk_count=floor(temp);
            temp=(double)released_pk/(double)pk_count; tr_count=floor(temp)+20;
            }
      }

if ( load_steady && !ready && released_pk>(tr_count*pk_count) )
      {
      if (!cu_steady)
            {
            ratio=0;
            for (i=1; i<=N1; i++)
                  if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
            temp=(time+ratio)*slot_no/(N*op_sim_time()); ratio=temp/cu_prev;
            if ( ratio > 0.999 && ratio < 1.001 )              cu_steady=1;
            else       cu_prev=temp;
```

```
            }
      if (!pb_steady)
            {
            temp=(double)(blocked_pks+blocked_pku)/(double)(total_pks+total_pku).
            if (pb_prev!=0) ratio=temp/pb_prev;
            else        if (temp==0) ratio=1.
                        else       ratio=0;
            if (ratio > 0.999 && ratio < 1.001)            pb_steady=1;
            else        pb_prev=temp;
            }
      if (!pbc_steady)
            {
            temp=(double)(blocked_c_pks+blocked_c_pku)/(double)(total_pks+total_pku);
            if (pbc_prev!=0) ratio=temp/pbc_prev;
            else        if (temp==0) ratio=1;
                        else       ratio=0;
            if (ratio > 0.999 && ratio < 1.001)            pbc_steady=1;
            else        pbc_prev=temp;
            }
      if ( cu_steady && pb_steady && pbc_steady ) ready=1;
      if (!ready)
                  {
                  ratio=0;
                  for (i=1; i<=N1; i++)
                        if (pkt[i].src!=0) ratio+=(op_sim_time()-pkt[i].ctime);
                  temp=(time+ratio)*slot_no*N1/(N*op_sim_time());
                  pk_count=2*floor(temp);      tr_count+=1;
                  }
      }

if (ready)
      {
      temp=slot_no*total_time/(N*op_sim_time()); ratio=temp/load_prev;
      if ( ratio > 0.995 && ratio < 1.005 && !ENDSIM)
            op_sim_end("reaching steady state","","","");
      }
```

State 2: idle (CET's)

CET	*Cond:*	(ARRIVAL)
#0	*Exec:*	;
	Trans:	**pk_prepare**
CET	*Cond:*	(default)
#1	*Exec:*	;
	Trans:	**idle**
CET	*Cond:*	(ENDSIM)
#2	*Exec:*	record_stats();
	Trans:	**idle**

State 3: **schedule (Enter Execs)** *forced, 13 lines*

```
if (no_in_svc == N1)
      {
      if (src<=server_no || dst<=server_no) blocked_pks+=1;
      else blocked_pku+=1;
   op_pk_destroy(pkptr);
      }
else
      if (!find_resource(pkptr))
            {
            if (src<=server_no || dst<=server_no) { blocked_pks+=1; blocked_c_pks+=1; }
            else { blocked_pku+=1; blocked_c_pku+=1; }
            op_pk_destroy(pkptr);
            }
```

State 3: **schedule (CET's)**

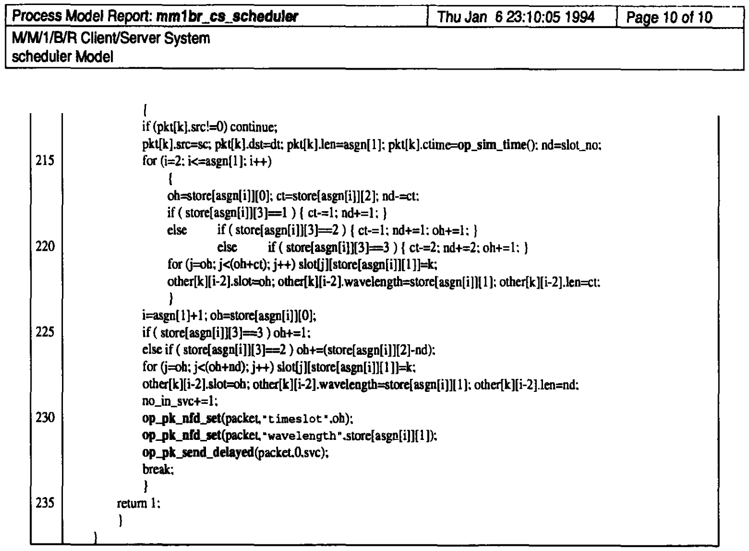
CET	*Cond:*	(1)
#0	*Exec:*	;
	Trans:	**idle**

Function Block *237 lines*

```
record_stats()
      {
      int                     i,j;
      double                  cu=0;
      extern double           time;
      extern channel_asgn     pkt[1025];

      for (i=1; i<=N1; i++)
            if (pkt[i].src!=0) time+=(op_sim_time()-pkt[i].ctime);

      cu=time*slot_no/(N*op_sim_time());
      if ( cu > cu_max )      cu=cu_max;
      op_stat_write_scalar("load",slot_no*total_time/(N*op_sim_time()));
      op_stat_write_scalar("cu",cu);
      op_stat_write_scalar("r",(double)total_pks/(double)(total_pks+total_pku));
      op_stat_write_scalar("pb",(double)(blocked_pks+blocked_pku)/(double)(total_pks+total_pku));
      op_stat_write_scalar("pbc",(double)(blocked_c_pks+blocked_c_pku)/(double)(total_pks+total_pku));
      op_stat_write_scalar("pbs",(double)blocked_pks/(double)total_pks);
      op_stat_write_scalar("pbcs",(double)blocked_c_pks/(double)total_pks);
      op_stat_write_scalar("pbu",(double)blocked_pku/(double)total_pku);
      op_stat_write_scalar("pbcu",(double)blocked_c_pku/(double)total_pku);
      }


int find_resource(packet)
      Packet*   packet;
      {
      int                     i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next,sc,dt;
      int                     conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
      double                  svc;
      extern int              no_in_svc,slot[130][10];
      extern other_asgn       other[1025][15];
      extern channel_asgn     pkt[1025];
```

```
      op_pk_nfd_get(packet,"src",&sc); op_pk_nfd_get(packet,"dst",&dt), op_pk_nfd_get(packet,"svc",&svc);

      ct=0;
      for (j=0; j<wavelength_no; j++) empty[j]=0;

      for (i=0;i<fm_size;i++)
            {

            conflict[i][0]=-1; conflict[i][1]=-1;
            for (k=0; k<wavelength_no; k++)
                  {
                  if ( slot[i][k]==0 ) empty[k]+=1;
                  if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
                  if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
                  }

            go_back=0;
            if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
                  {
                  for (k=0; k<wavelength_no; k++)
                        {
                        if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
                              {
                              if ( slot[i][k]!=0 ) j=i-empty[k]-1; else j=i-empty[k];
                              if ( j<0 ) oh=0;
                              else
                                    if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                                    else oh=2;
                              if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
                              insert:
                              if ( ct==0 )
                                    {
                                    if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k]; else store[0][0]=i-empty[k]+1;
                                    store[0][1]=k;      store[0][2]=empty[k];       store[0][3]=oh;
                                    }
                              else
                                    for (j=(ct-1); j>=0; j--)
                                          {
                                          if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
                                                {
                                                store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
                                                store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
                                                }
                                          if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
                                                {
                                                store_ct:
                                                if ( slot[i][k]!=0 || go_back ) store[j+1][0]=i-empty[k]; else store[j+1][0]=i-empty[k]+1;
                                                store[j+1][1]=k; store[j+1][2]=empty[k]; store[j+1][3]=oh; break;
                                                }
                                          if ( j==0 ) { j=-1; goto store_ct; }
                                          }
                              if ( go_back ) goto back;
                              empty[k]=0; ct+=1;
                              }
                        }
                  }
            else
                  {
                  for ( k=0; k<wavelength_no; k++)
            {
            if ( empty[k]>0 )
                {
                if ( slot[i][k]==0 ) empty[k]-=1;
                if ( empty[k]==0) continue;
                j=i-empty[k]-1;
                if ( j<0 ) oh=0;
                else
                    if ((conflict[j][0]==-1 || conflict[j][0]==k)&&(conflict[j][1]==-1 || conflict[j][1]==k)) oh=0;
                    else oh=2;
                if ((conflict[i][0]==-1 || conflict[i][0]==k)&&(conflict[i][1]==-1 || conflict[i][1]==k)) oh=oh;
                else        if ( oh==0 ) oh=1; else oh=3;
                if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0; continue; }
                go_back=1; goto insert;
                back:
                empty[k]=0; ct+=1;
                }
            }
        }
    }

nd=slot_no; i_ct=0; waste=0; next=0; asgn[0]=-1; asgn[1]=0;
for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
for (i=0; i<ct; i++)
    {
    if ( store[i][3]==0 ) fit = 0;
    else if ( store[i][3]==3 ) fit = 2; else fit = 1;
    j=store[i][2]-fit;

    if (next)
        {
        if ( j>=nd )
            {
            k=j-nd+fit;
            if ( fit<= ap1 && k<ap )
                {
                for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                    if ( rmb[oh] ) { k=-1; break; }
                if ( k!=-1 )
                    {
                    ap=k; ap1=fit; temp[i_ct]=i;
                    for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                    k=store[i][0]-1; if ( k<0 ) k=0;
                    go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size.
                    for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1;
                    }
                }
            }
        if ( j<nd || i==(ct-1) )
            {
            waste+=ap1; i_ct+=1;
            if ( waste<asgn[0] || asgn[0]==-1 )
                {
                asgn[0]=waste; asgn[1]=i_ct;
                for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                if ( asgn[0]==0 ) goto assign;
                }
            if ( j<nd )
                {
```

```
                temp[0]=i; nd=slot_no-j; waste=fit; i_ct=1; next=0;
                for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                k=store[i][0]-1; if ( k<0 ) k=0;
                go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                }
            }
        continue;
        }

    k=0;
    for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
        if ( rmb[oh] ) { k=1; break; }
    if ( k==1 ) continue;
    else
        {
        temp[i_ct]=i;
        k=store[i][0]-1; if ( k<0 ) k=0;
        go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
        for (oh=k; oh<go_back; oh++)
            if ( j>nd ) tmp_rmb[oh]=1;
            else        rmb[oh]=1;
        }

    if ( j<nd )
        { nd-=j; waste+=fit; i_ct+=1; continue; }
    if ( j==nd )
        {
        waste+=fit; i_ct+=1;
        if ( waste<asgn[0] || asgn[0]==-1 )
            {
            asgn[0]=waste; asgn[1]=i_ct;
            for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
            if ( asgn[0]==0 ) goto assign;
            }
        nd=slot_no; i_ct=0; waste=0;
        for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
        continue;
        }
    if ( j>nd )
        { ap=j-nd+fit; ap1=fit; next=1; }
    if ( next && i==(ct-1) )
        {
        waste+=ap1; i_ct+=1;
        if ( waste<asgn[0] || asgn[0]==-1 )
            {
            asgn[0]=waste; asgn[1]=i_ct;
            for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2].
            if ( asgn[0]==0 ) goto assign;
            }
        }
    }

if(asgn[0]==-1)
    return 0;
else
    {
    assign:
    for (k=1; k<=N1; k++)
```



Process Model Report: **mm1br_cs_scheduler** | Thu Jan 6 23:10:05 1994 | Page 10 of 10

M/M/1/B/R Client/Server System
scheduler Model

```
            {
            if (pkt[k].src!=0) continue;
            pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1]; pkt[k].ctime=op_sim_time(); nd=slot_no;
            for (i=2; i<=asgn[1]; i++)
                {
                oh=store[asgn[i]][0]; ct=store[asgn[i]][2]; nd-=ct;
                if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                else        if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                            else        if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
                }
            i=asgn[1]+1; oh=store[asgn[i]][0];
            if ( store[asgn[i]][3]==3 ) oh+=1;
            else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
            for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
            other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
            no_in_svc+=1;
            op_pk_nfd_set(packet,"timeslot",oh);
            op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
            op_pk_send_delayed(packet,0,svc);
            break;
            }
        return 1;
        }
    }
```

B.5.2 Queueing System

The termination criteria and the seed used here is the same as that in Section B.3.2.

<u>M/M/1/Q/RL:</u> OPNET report for the *scheduler* processor model. The *release* model is identical to that in M/M/1/Q/RL uniform traffic system.

Process Model Report: **mm1qr_cs_scheduler**	Fri Jan 7 02:35:18 1994	Page 1 of 12
M/M/1/Q/R Client/Server System scheduler Model		

Summary				
Number of States	*Header Block*	*State Variables*	*Temporary Variables*	*Function Block*
4	**Yes**	**Yes**	**Yes**	**Yes**

Header Block *29 lines*

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

typedef struct
    {
    int         src;
    int         dst;
    int         len;
    double      ctime;
    } channel_asgn;

typedef struct
    {
    int   slot;
    int   wavelength;
    int   len;
    } other_asgn;

channel_asgn        pkt[1025];
other_asgn          other[1025][15];
int                 no_in_svc,released_pk,slot[130][10];
double              time;

#define   ARRIVAL     (op_intrpt_type()==OPC_INTRPT_STRM)
#define   ENDSIM (op_intrpt_type()==OPC_INTRPT_ENDSIM)
#define EMPTYQ  (op_subq_empty(0))
#define NEXTQ     (op_intrpt_type()==OPC_INTRPT_REMOTE &&    !EMPTYQ)
#define   MEASURE (op_intrpt_type()==OPC_INTRPT_SELF && op_intrpt_code()==0)
```

Process Model Report: **mm1qr_cs_scheduler**	Fri Jan 7 02:35:19 1994	Page 2 of 12
M/M/1/Q/R Client/Server System scheduler Model		

State Variables *43 lines*

```
Objid           \own_id;
Objid           \src_id;
int             \node_no;
int             \server_no;
int             \wavelength_no;
int             \fm_size;
int             \T1;
int             \slot_no;
int             \N;
int             \N1;
double          \traffic;
double          \svc_time;
char            \arv_arg[20];
double          \arv_rate;
Distribution    *\tfc_dist;
Distribution    *\srv_dist;
Distribution    *\src_dist;
Distribution    *\dst_dist;
Distribution    *\svc_dist;
long int        \total_pks;
long int        \total_pku;
long int        \processed_pks;
long int        \processed_pku;
long int        \stat_ct;
long int        \next_stat;
long int        \n0;
long int        \n1;
double          \load_prev;
double          \stats[200000];
double          \statu[200000];
double          \total_time;
double          \delays;
double          \delayu;
double          \tstarts;
double          \tstartu;
double          \qsizes;
double          \qsizeu;
double          \queues;
double          \queueu;
double          \inc;
double          \z;
int             \load_steady;
int             \end_sim;
```

Temporary Variables *3 lines*

```
Packet*   pkptr;
int       src,dst,i,j;
double    ratio,temp;
```

Process Model Report: **mm1qr_cs_scheduler**	Fri Jan 7 02:35:19 1994	Page 4 of 12

M/M/1/Q/R Client/Server System
scheduler Model

State 1: pk_prepare (Enter Execs) *forced, 22 lines*

```
pkptr=op_pk_get(op_intrpt_strm());

temp=(int)op_dist_outcome(tfc_dist);
if (temp)       /* client-server traffic */
      {
      ratio=(int)op_dist_outcome(srv_dist);
      if (ratio<=server_no)
            {src=ratio; dst=(int)op_dist_outcome(dst_dist); total_pks+=1; op_pk_nfd_set(pkptr,"type",1);}
      else
            {dst=ratio-server_no; src=(int)op_dist_outcome(src_dist); total_pks+=1; op_pk_nfd_set(pkptr,"type",1);}
      }
else
      {
      src=(int)op_dist_outcome(src_dist);
      next:
      dst=(int)op_dist_outcome(dst_dist);
      if (dst == src) goto next;
      total_pku+=1; op_pk_nfd_set(pkptr,"type",0);
      }

temp=op_dist_outcome(svc_dist); total_time+=temp;
op_pk_nfd_set(pkptr,"src",src); op_pk_nfd_set(pkptr,"dst",dst); op_pk_nfd_set(pkptr,"svc",temp);
```

State 1: pk_prepare (CET's)

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**schedule**

Process Model Report: **mm1qr_cs_scheduler**	Fri Jan 7 02:35:19 1994	Page 3 of 12

M/M/1/Q/R Client/Server System
scheduler Model

State 0: init (Enter Execs) *forced, 35 lines*

```
f_inverse(0.1,&z);       /* 90% confidence interval */

op_ima_sim_attr_get(OPC_IMA_INTEGER,"N",&node_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"W",&wavelength_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"T",&fm_size);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"L",&slot_no);
op_ima_sim_attr_get(OPC_IMA_INTEGER,"Ns",&server_no);
op_ima_sim_attr_get(OPC_IMA_DOUBLE,"tp",&traffic);

own_id=op_id_self(); src_id=op_topo_in_assoc(own_id,0);
op_ima_obj_attr_get(src_id,"interarrival args",&arv_arg[0]);

arv_rate=atof(&arv_arg[0]); arv_rate=1/arv_rate;

N=wavelength_no*fm_size; svc_time=(double)N/(double)slot_no;
N1=floor(svc_time); svc_time=(double)fm_size/(double)slot_no;

svc_dist=op_dist_load("exponential",svc_time,0);
tfc_dist=op_dist_load("bernoulli",(double)(2*traffic*server_no),0);
srv_dist=op_dist_load("uniform_int",1,(double)(2*server_no));
src_dist=op_dist_load("uniform_int",(double)(1+server_no),(double)node_no);
dst_dist=op_dist_load("uniform_int",(double)(1+server_no),(double)node_no);

total_pks=0; total_pku=0; processed_pks=0; processed_pku=0;
inc=10; no_in_svc=0; time=0; total_time=0; delays=0; delayu=0;
load_prev=arv_rate/(double)wavelength_no;
load_steady=0; end_sim=0; n0=600; n1=800; op_subq_flush(0);

for (i=0;i<=N1;i++)
      {
      pkt[i].src=0; pkt[i].dst=0; pkt[i].len=0; pkt[i].ctime=0;
      for (j=0; j<15; j++) {other[i][j].slot=0; other[i][j].wavelength=0; other[i][j].len=0;}
      }
for (i=0;i<fm_size;i++)
      for (j=0;j<wavelength_no;j++) slot[i][j]=0;
```

State 0: init (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**

State 2: idle (Enter Execs) *unforced, 37 lines*

```
if (!load_steady)
     {
     temp=total_time*slot_no/(N*op_sim_time()*load_prev);
     if (temp>0.999 && temp<1.001)
          {
          tstarts=op_sim_time(); tstartu=op_sim_time(); qsizes=find_qsize(1). qsizeu=find_qsize(0).
          queues=0; queueu=0; load_steady=1; stat_ct=0; next_stat=800;
          op_intrpt_schedule_self(op_sim_time()+inc,0);
          }
     }

if (MEASURE)
     {
     temp=op_sim_time(); queues+=qsizes*(temp-tstarts); queueu+=qsizeu*(temp-tstartu);
     stats[stat_ct]=queues/inc; statu[stat_ct]=queueu/inc; queues=0; queueu=0;
     stat_ct+=1; tstarts=temp; tstartu=temp; qsizes=find_qsize(1); qsizeu=find_qsize(0);
     if (stat_ct==next_stat)
          {
          step2:
          serial_col(400,&temp);
          if (temp>0.4) goto step5;
          if (temp<0) goto step4;
          step3:
          serial_col(200,&ratio);
          if (ratio>temp) goto step5;
          step4:
          col(40,&temp);
          if (temp<0.1) end_sim=1;
          step5:
          if (n0>n1) { n1=n1*2; next_stat=n1; }
          else {n0=n0*2; next_stat=n0; }
          }
     op_intrpt_schedule_self(op_sim_time()+inc,0);
     }

if (end_sim && !ENDSIM)
     op_sim_end("reaching steady state","","","");
```

State 2: idle (CET's)

CET #0	*Cond:*	(ARRIVAL)
	Exec:	;
	Trans:	**pk_prepare**
CET #1	*Cond:*	(default)
	Exec:	;
	Trans:	**idle**
CET #2	*Cond:*	(ENDSIM)
	Exec:	record_stats();
	Trans:	**idle**
CET #3	*Cond:*	(NEXTQ && !ARRIVAL)
	Exec:	;
	Trans:	**schedule**

State 3: schedule (Enter Execs) *forced, 47 lines*

```
if ( ARRIVAL )
     {
     if (no_in_svc == Ni)
          {
       if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
               { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
          else      if (load_steady)
                    {
                    temp=op_sim_time();
                    if (src<=server_no || dst<=server_no)
                         {queues+=qsizes*(temp-tstarts); tstarts=temp; qsizes=find_qsize(1);}
                    else
                         {queueu+=qsizeu*(temp-tstartu); tstartu=temp; qsizeu=find_qsize(0);}
                    }
          }
     else
          if (!find_resource(pkptr))
               {
            if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                    { printf("error inserting into queue\n"); op_pk_destroy(pkptr); }
               else      if (load_steady)
                         {
                         temp=op_sim_time();
                         if (src<=server_no || dst<=server_no)
                              {queues+=qsizes*(temp-tstarts); tstarts=temp; qsizes=find_qsize(1);}
                         else
                              {queueu+=qsizeu*(temp-tstartu); tstartu=temp; qsizeu=find_qsize(0);}
                         }
               }
     }

if ( NEXTQ )
          {
          ratio=op_subq_stat(0,OPC_QSTAT_PKSIZE);
          for (i=0;i<ratio;i++)
               if (no_in_svc < N1)
                    {
                    pkptr=op_subq_pk_remove(0,0);
                    if (!find_resource(pkptr))
                       if (op_subq_pk_insert(0,pkptr,OPC_QPOS_TAIL)!=OPC_QINS_OK)
                              { printf("error inserting into queue\n"), op_pk_destroy(pkptr); }
                    }
          temp=op_sim_time(); ratio=find_qsize(1);
          if (ratio!=qsizes && load_steady) {queues+=qsizes*(temp-tstarts); tstarts=temp; qsizes=ratio;}
          ratio=find_qsize(0);
          if (ratio!=qsizeu && load_steady) {queueu+=qsizeu*(temp-tstartu); tstartu=temp; qsizeu=ratio;}
          }
```

State 3: schedule (CET's)

CET #0	*Cond:*	(1)
	Exec:	;
	Trans:	**idle**

Function Block *352 lines*

```
record_stats()
     {
     int                    i,j,b;
     double                 x,xs,xu,s,ss,su,y[40],ys[40],yu[40];
     extern int             slot[130][10];
     extern double          time;
     extern channel_asgn    pkt[1025];

     s=(double)stat_ct/(double)40;
     b=floor(s);
     x=0; xs=0; xu=0;
     for (i=0;i<40;i++)
          {
          y[i]=0; ys[i]=0; yu[i]=0;
          for (j=0;j<b;j++) {ys[i]+=stats[i*b+j]; yu[i]+=statu[i*b+j]; y[i]=ys[i]+yu[i];}
          y[i]=y[i]/b; ys[i]=ys[i]/b; yu[i]=yu[i]/b; x+=y[i]; xs+=ys[i]; xu+=yu[i];
          }
     x=x/40; xs=xs/40; xu=xu/40; s=0; ss=0; su=0;
     for (i=0;i<40;i++) {j=y[i]-x; s+=j*j; j=ys[i]-xs; ss+=j*j; j=yu[i]-xu; su+=j*j;}
     s=s/39; ss=ss/39; su=su/39;
     if (x!=0) s=z*sqrt(s/40)/x;
     if (xs!=0) ss=z*sqrt(ss/40)/xs;
     if (xu!=0) su=z*sqrt(su/40)/xu;

     for (i=1; i<=N1; i++)
          if (pkt[i].src!=0) time+=(op_sim_time()-pkt[i].ctime);

     op_stat_write_scalar("load",total_time*slot_no/(N*op_sim_time()));
     op_stat_write_scalar("cu",time*slot_no/(N*op_sim_time()));
     op_stat_write_scalar("r",(double)total_pks/(double)(total_pks+total_pku));
     op_stat_write_scalar("Q",x);
     op_stat_write_scalar("D",(delays+delayu)/(double)(processed_pks+processed_pku));
     op_stat_write_scalar("Qs",xs);
     op_stat_write_scalar("Ds",delays/(double)processed_pks);
     op_stat_write_scalar("Qu",xu);
     op_stat_write_scalar("Du",delayu/(double)processed_pku);
     op_stat_write_scalar("qd",s);
     op_stat_write_scalar("qds",ss);
     op_stat_write_scalar("qdu",su);
     op_stat_write_scalar("c",stat_ct);
     }


f_inverse (x,z1)
     double    x,*z1;
     {
     double    y,p,p0,p1,p2,p3,p4,q0,q1,q2,q3,q4;

     x=1-x/2;
     p0=-0.322232431088e0; p1=-1.0; p2=-0.342242088547e0; p3=-0.0204231210245e0; p4=-0.453642210148e-4;
     q0=0.099348462606e0; q1=0.588581570495e0; q2=0.531103462366e0; q3=0.10353775285e0; q4=0.38560700634e-2;

     p=x;
     if (p>0.5) p=1.0-p;
     y=sqrt(-log(p*p));
     *z1=y+(p0+y*(p1+y*(p2+y*(p3+y*p4))))/(q0+y*(q1+y*(q2+y*(q3+y*q4))));
     if (x<0.5) *z1=-*z1;
     }
```

```
int find_qsize(tp)
     int  tp;
     {
     int        i,tpf,j=0;
     double     qsize;
     Packet*    pk;

     qsize=op_subq_stat(0,OPC_QSTAT_PKSIZE);
     for (i=0;i<qsize;i++)
          {
          pk=op_subq_pk_remove(0,0); op_pk_nfd_get(pk,"type",&tpf);
          if (tpf==tp) j+=1;
          op_subq_pk_insert(0,pk,OPC_QPOS_TAIL);
          }
     return j;
     }


serial_col(n,p1)
     int        n;
     double     *p1;
     {
     int   i,j,k,b;
     double     p,x,x1,x2,c,c1,c2,s,s1,s2,y[400];

     b=next_stat/n; k=n/2; x=0; x1=0; x2=0;
     for (i=0;i<n;i++)
          {
          y[i]=0;
          for (j=0;j<b;j++)          y[i]+=(stats[i*b+j]+statu[i*b+j]);
          y[i]=y[i]/b; x+=y[i];
          if (i<k) x1+=y[i]; else x2+=y[ı];
          }
     x=x/n; x1=x1/k; x2=x2/k; s=0; s1=0; s2=0: c=0; c1=0; c2=0;
     for (i=0;i<n;i++)
          {
          j=y[i]-x; s+=j*j;
          if (i<(n-1)) c+=j*(y[i+1]-x);
          if (i<k)
               {
               j=y[i]-x1; s1+=j*j;
               if (i<(k-1)) c1+=j*(y[i+1]-x1);
               }
          else
               {
               j=y[i]-x2; s2+=j*j;
               if (i<(n-1)) c2+=j*(y[i+1]-x2);
               }
          }
     if (s==0) *p1=0;
     else      *p1=2*c/s-(c1/s1+c2/s2)/2;
     }


col(n,p1)
     int        n;
     double     *p1;
     {
```

```
    int    i,j,b;
    double    p,x,s,y[400];

    b=next_stat/n; x=0;
    for (i=0;i<n;i++)
        {
        y[i]=0;
        for (j=0;j<b;j++)            y[i]+=(stats[i*b+j]+statu[i*b+j]);
        y[i]=y[i]/b; x+=y[i];
        }
    x=x/n; s=0;
    for (i=0;i<n;i++) { j=y[i]-x; s+=j*j; }
    s=s/(n-1); s=sqrt(s/n);
    if (s==0) *p1=0; else *p1=z*s/x;
    printf("%d: %f\t%f\n",stat_ct,x,*p1);
    }


int find_resource(packet)
    Packet*    packet;
    {
    int                    i,j,k,ap,ap1,nd,ct,oh,go_back,waste,i_ct,fit,next,sc,dt,tp;
    int                    conflict[130][2],empty[10],store[1025][4],temp[130],asgn[130],rmb[130],tmp_rmb[130];
    double                 svc;
    extern int             no_in_svc,slot[130][10];
    extern other_asgn      other[1025][15];
    extern channel_asgn    pkt[1025];

    op_pk_nfd_get(packet,"type",&tp); op_pk_nfd_get(packet,"src",&sc);
    op_pk_nfd_get(packet,"dst",&dt); op_pk_nfd_get(packet,"svc",&svc);

    ct=0;
    for (j=0; j<wavelength_no; j++) empty[j]=0;

    for (i=0;i<fm_size;i++)
        {             .

        conflict[i][0]=-1; conflict[i][1]=-1;
        for (k=0; k<wavelength_no; k++)
            {
            if ( slot[i][k]==0 ) empty[k]+=1;
            if ( pkt[slot[i][k]].src==sc ) conflict[i][0]=k;
            if ( pkt[slot[i][k]].dst==dt ) conflict[i][1]=k;
            }

        go_back=0;
        if ( conflict[i][0]==-1 && conflict[i][1]==-1 )
            {
            for (k=0; k<wavelength_no; k++)
                {
                if ( (slot[i][k]!=0 || i==(fm_size-1)) && empty[k]>0 )
                    {
                    if ( slot[i][k]!=0 ) j=i-empty[k]-1; else j=i-empty[k];
                    if ( j<0 ) oh=0;
                    else
                        if ((conflict[j][0]==-1||conflict[j][0]==k)&&(conflict[j][1]==-1||conflict[j][1]==k)) oh=0;
                        else oh=2;
                    if ( oh==2 && empty[k]<2 ) { empty[k]=0; continue; }
                    insert:
```

```
                    if ( ct==0 )
                        {
                        if ( slot[i][k]!=0 || go_back ) store[0][0]=i-empty[k];
                        else        store[0][0]=i-empty[k]+1;
                        store[0][1]=k;        store[0][2]=empty[k];          store[0][3]=oh;
                        }
                    else
                        for (j=(ct-1); j>=0; j--)
                            {
                            if ( store[j][2]<empty[k] || (store[j][2]==empty[k] && store[j][3]>oh) )
                                {
                                store[j+1][0]=store[j][0]; store[j+1][1]=store[j][1];
                                store[j+1][2]=store[j][2]; store[j+1][3]=store[j][3];
                                }
                            if ( store[j][2]>empty[k] || (store[j][2]==empty[k] && store[j][3]<=oh) )
                                {
                                store_ct:
                                if ( slot[i][k]!=0 || go_back ) store[j+1][0]=i-empty[k];
                                else        store[j+1][0]=i-empty[k]+1;
                                store[j+1][1]=k; store[j+1][2]=empty[k]; store[j+1][3]=oh; break;
                                }
                            if ( j==0 ) { j=-1; goto store_ct; }
                            }
                    if ( go_back ) goto back;
                    empty[k]=0; ct+=1;
                    }
                }
            }
        else
            {
            for ( k=0; k<wavelength_no; k++)
                {
                if ( empty[k]>0 )
                    {
                    if ( slot[i][k]==0 ) empty[k]-=1;
                    if ( empty[k]==0) continue;
                    j=i-empty[k]-1;
                    if ( j<0 ) oh=0;
                    else
                        if ((conflict[j][0]==-1||conflict[j][0]==k)&&(conflict[j][1]==-1||conflict[j][1]==k)) oh=0;
                        else oh=2;
                    if ((conflict[i][0]==-1||conflict[i][0]==k)&&(conflict[i][1]==-1||conflict[i][1]==k)) oh=oh;
                    else        if ( oh==0 ) oh=1; else oh=3;
                    if ( ((oh==2 || oh==1)&& empty[k]<2) || (oh==3 && empty[k]<3 )) { empty[k]=0; continue; }
                    go_back=1;
                    goto insert,
                    back:
                    empty[k]=0; ct+=1;
                    }
                }
            }
        }

    nd=slot_no; i_ct=0, waste=0; next=0; asgn[0]=-1; asgn[1]=0;
    for (i=0; i<fm_size; i++) { rmb[i]=0; tmp_rmb[i]=0; }
    for (i=0; i<ct; i++)
        {
        if ( store[i][3]==0 ) fit = 0;
        else if ( store[i][3]==3 ) fit = 2; else fit = 1;
```

```
          j=store[i][2]-fit;

          if (next)
               {
               if ( j>=nd )
                    {
                    k=j-nd+fit;
                    if ( fit<= ap1 && k<ap )
                         {
                         for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
                              if ( rmb[oh] ) { k=-1; break; }
                         if ( k!=-1 )
                              {
                              ap=k; ap1=fit; temp[i_ct]=i;
                              for (oh=0; oh<fm_size; oh++) tmp_rmb[oh]=0;
                              k=store[i][0]-1; if ( k<0 ) k=0;
                              go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                              for (oh=k; oh<go_back; oh++) tmp_rmb[oh]=1;
                              }
                         }
                    }
               if ( j<nd || i==(ct-1) )
                    {
                    waste+=ap1; i_ct+=1;
                    if ( waste<asgn[0] || asgn[0]==-1 )
                         {
                         asgn[0]=waste; asgn[1]=i_ct;
                         for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2].
                         if ( asgn[0]==0 ) goto assign;
                         }
                    if ( j<nd )
                         {
                         temp[0]=i; nd=slot_no-j; waste=fit; i_ct=1; next=0;
                         for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
                         k=store[i][0]-1; if ( k<0 ) k=0;
                         go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
                         for (oh=k; oh<go_back; oh++) rmb[oh]=1;
                         }
                    }
               continue;
               }

          k=0;
          for (oh=store[i][0]; oh<(store[i][0]+store[i][2]); oh++)
               if ( rmb[oh] ) { k=1; break; }
          if ( k==1 ) continue;
          else
               {
               temp[i_ct]=i;
               k=store[i][0]-1; if ( k<0 ) k=0;
               go_back=store[i][0]+store[i][2]+1; if ( go_back>fm_size ) go_back=fm_size;
               for (oh=k; oh<go_back; oh++)
                    if ( j>nd ) tmp_rmb[oh]=1; else          rmb[oh]=1;
               }

          if ( j<nd )
               { nd-=j; waste+=fit; i_ct+=1; continue; }
          if ( j==nd )
               {
```

```
               waste+=fit; i_ct+=1;
               if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
               nd=slot_no; i_ct=0; waste=0;
               for (oh=0; oh<fm_size; oh++) { rmb[oh]=0; tmp_rmb[oh]=0; }
               continue;
               }
          if ( j>nd )
               { ap=j-nd+fit; ap1=fit; next=1; }
          if ( next && i==(ct-1) )
               {
               waste+=ap1; i_ct+=1;
               if ( waste<asgn[0] || asgn[0]==-1 )
                    {
                    asgn[0]=waste; asgn[1]=i_ct;
                    for ( k=2; k<(i_ct+2); k++ ) asgn[k]=temp[k-2];
                    if ( asgn[0]==0 ) goto assign;
                    }
               }
          }

     if(asgn[0]==-1)
          return 0;
     else
          {
          assign:
          for (k=1; k<=N1; k++)
               {
               if (pkt[k].src!=0) continue;
               pkt[k].src=sc; pkt[k].dst=dt; pkt[k].len=asgn[1]; pkt[k].ctime=op_sim_time(); nd=slot_no;
               for (i=2; i<=asgn[1]; i++)
                    {
                    oh=store[asgn[i]][0]; ct=store[asgn[i]][2]; nd-=ct;
                    if ( store[asgn[i]][3]==1 ) { ct-=1; nd+=1; }
                    else      if ( store[asgn[i]][3]==2 ) { ct-=1; nd+=1; oh+=1; }
                              else      if ( store[asgn[i]][3]==3 ) { ct-=2; nd+=2; oh+=1; }
                    for (j=oh; j<(oh+ct); j++) slot[j][store[asgn[i]][1]]=k;
                    other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=ct;
                    }
               i=asgn[1]+1; oh=store[asgn[i]][0];
               if ( store[asgn[i]][3]==3 ) oh+=1;
               else if ( store[asgn[i]][3]==2 ) oh+=(store[asgn[i]][2]-nd);
               for (j=oh; j<(oh+nd); j++) slot[j][store[asgn[i]][1]]=k;
               other[k][i-2].slot=oh; other[k][i-2].wavelength=store[asgn[i]][1]; other[k][i-2].len=nd;
               no_in_svc+=1;
               if (tp) {processed_pks+=1; delays+=op_sim_time()-op_pk_creation_time_get(packet);}
               else {processed_pku+=1; delayu+=op_sim_time()-op_pk_creation_time_get(packet);}
               op_pk_nfd_set(packet,"timeslot",oh); op_pk_nfd_set(packet,"wavelength",store[asgn[i]][1]);
               op_pk_send_delayed(packet,0,svc); break;
               }
          return 1;
          }
     }
```

References

[1] R. Ramaswami, "Multiwavelength Lightwave Networks for Computer Communications", IEEE Communications Magazine, pp. 78-88, Feb 1993

[2] B. Mukherjee, "WDM-Based Local Lightwave Networks Part I: Single-Hop Systems," *IEEE Network*, pp. 12-27, May 1992

[3] B. Mukherjee, "WDM-Based Local Lightwave Networks Part II: Multihop Systems," *IEEE Network*, pp. 20-32, July 1992

[4] S.B. Alesander, R.S. Bondurant, D. Byrne, V.W.S. Chan, S. Finn, R. Gallager, R.S. Kennedy, et al, "A Consortium on Wideband All-Optical Networks," to be published in *Journal of Lightwave Technology*

[5] J.F. Shoch, J.A. Hupp, "Measured Performance of an Ethernet Local Network", *Communications of the ACM*, pp. 711-721, Dec 1980

[6] OPNET Manuals by *Mil 3, Inc.*

[7] D. Bertsekas, R. Gallager, "Data Networks", *Prentice-Hall, Inc*. NJ, p.200, 1987

[8] R. Jain, "The Art of Computer Systems Performance Analysis", *John Wiley & Sons*, pp. 432-436, 1991

[9] M.A. Crane, A.J. Lemoine, "An Introduction to the Regenerative Method for Simulation Analysis", *Springer-Verlag*, pp. 36-46, 1977

[10] P. Bratley, B.L. Fox, L.E. Schrage, "A Guide to Simulation", *Springer-Verlag*, p. 327, 1983

[11] P. Bratley, B.L. Fox, L.E. Schrage, "A Guide to Simulation", *Springer-Verlag*, pp. 86-87, 1983