NASA Contractor Report 187438
ICASE Report No. 90–64

# ICASE

## CONSERVATIVE PARALLEL SIMULATION OF PRIORITY CLASS QUEUEING NETWORKS

David M. Nicol

Contract No. NAS1–18605
September 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

# Conservative Parallel Simulation of Priority Class Queueing Networks

*David Nicol* *
*College of William and Mary*

**Abstract**

This paper describes a conservative synchronization protocol for the parallel simulation of queueing networks having $C$ job priority classes, where a job's class is fixed. This problem has long vexed designers of conservative synchronization protocols because of its seemingly poor ability to compute *lookahead*: the time of the next departure. For, a job in service having low priority can be preempted at any time by an arrival having higher priority and an arbitrarily small service time. Our solution is to skew the event generation activity so that events for higher priority jobs are generated farther ahead in simulated time than lower priority jobs. Thus, when a lower priority job enters service for the first time, all the higher priority jobs that may preempt it are already known and the job's departure time can be exactly predicted. Finally, we analyze the protocol and demonstrate that good performance can be expected on the simulation of large queueing networks.

*This research was supported in part by NASA grant NAG-1-1132, in part by NASA grant NAS-1-18605, and in part by NSF Grant ASC 8819373.

# 1 Introduction

We are interested in using a parallel computer to execute a discrete-event simulation of an open queueing network with the following properties.

- Each job in the network belongs to one of $C$ priority classes; a job never changes its class as it moves between queues. We adopt the nomenclature that a job from class $j$ has lower priority than one from class $k > j$.
- A job in service is immediately preempted by the arrival of one with a higher priority.
- Once a class $k$ job enters service for the first time, it must complete its service before any other class $k$ job is serviced.
- At the time a class $k$ job joins a queue, its eventual service requirement at the queue is known.
- Once a class $k$ job enters service for the first time, its next destination queue can be determined. That destination is not permitted to depend on any information about jobs from lower priority classes.

The third assumption is satisfied by any non-preemptive priority discipline where priorities depend only on the original arrival time, e.g., FCFS, or LCFS. We do permit a preempted job to "lose" the service it received before preemption. This is useful in a context where a preempted job has to start over. This phenomenon has been described in the context of a simulation where the jobs correspond to messages being sent between processors in a simulated multiprocessor system [3]; a message is successfully sent only after it receives enough time to complete a transmission without interruption. Each transmission attempt has the same service time, as it would if the service time were determined by message length.

The last two assumptions are trivially satisfied in the most common type of stochastic simulation, where service times and destinations are chosen randomly in accordance with class-specific probability distributions. Intra-class priority disciplines that are explicitly excluded from our approach include any that depend on service time requirements, for example, processor-sharing, shortest-job-first, or longest-job-first. Nevertheless, the simulation model we assume is fairly standard for the stochastic simulation of communication networks where messages (i.e. jobs) have different priorities. It is also important to note that while the mathematical analysis of queueing networks has progressed far, this particular class of queueing networks does not admit to an exact solution even under the most benign assumptions [7]. Simulation remains an important tool for the performance analysis of these types of networks.

A principle issue in parallelizing a discrete-event simulation is synchronization. One generally assumes that each processor maintains its own simulation clock, which is advanced each time the processor evaluates an event. Evaluation of an event on a processor may cause an event which must be evaluated on another processor. The results of the parallelized simulation are equivalent to a serially executed simulation if each processor evaluates its events in monotone non-decreasing time-stamp order. Synchronization protocols that ensure this monotonicity are sometimes called *conservative*. *Optimistic* methods have also been developed to permit processors to evaluate events in non-monotonic order. The end result of an optimistic simulation is still the same, as a processor "rolls back" in simulation time whenever it is called upon to evaluate an event at a time strictly less than the processor's simulation clock. Fujimoto [4] gives an excellent survey of the current state-of-the-art in parallel simulation.

The present paper presents a conservative algorithm for simulating priority class queueing networks. With a single possible exception, an efficient solution of this problem has long eluded conservative methods, because they require *lookahead*–an ability to predict future behavior. As applied to a queue, lookahead is the ability to predict when next a job will depart the queue. The conservative conditional-event approach of Chandy and Sherman [2] may provide the exception; while they do not present any performance measurements or analysis of priority class networks, they do show how their method deals with a two-class network. Otherwise, the lookahead properties of priority class queueing networks have been thought to be poor [3], owing to the fact that at any time a highest priority job with an arbitrarily small service time may arrive at a queue servicing a lower priority job, preempt it, immediately receive service, and depart. We will show that lookahead properties can actually be quite good, provided that one concurrently simulates different job classes in different non-overlapping regions of simulation time. This is possible because the behavior of a job from a high priority class is completely unaffected by jobs from lower priority classes. Higher priority classes can therefore be correctly simulated far enough ahead in simulated time so that when a lower priority job enters service, the existence and behavior of higher priority jobs that will affect it are already known. The independence of higher priority jobs from lower priority ones was first noted and exploited by Sevcik in proposing the "shadow CPU" algorithm for approximating the performance characteristics of priority class networks [7].

The protocol we propose and analyze is an extension of one analyzed in [10] and [11]. Like its predecessors, it is a synchronous protocol, meaning that it repeatedly defines global synchronization points in such a way that processors are free to execute completely in parallel between two synchronization points. Selection of the synchronization points depends on a

run-time analysis of the simulation model state. At issue is whether enough events are found between synchronization points for one to achieve good performance. We answer this question analytically, showing that in "homogeneous" networks (meaning that queues cannot be distinguished topologically) the average number of events processed between synchronization points is $\Omega(DM + \sqrt{M})$, where $M$ is the number of queues and $D$ is a lower bound on a job's service requirement. Thus, the number of events processed between synchronizations increases without bound as the number of queues increases without bound; furthermore, the increase is super-linear in $M$ when $D > 0$. We also analyze the memory requirements of our approach, showing that the average memory required for our approach is $O(CM \log M)$.

This paper is organized as follows. §2 defines the events in the simulation model, and our assumptions concerning their meaning and ordering. §3 describes our earlier protocol, while §4 introduces its extension to priority class networks. §5 presents our performance analysis, and §6 summarizes this work.

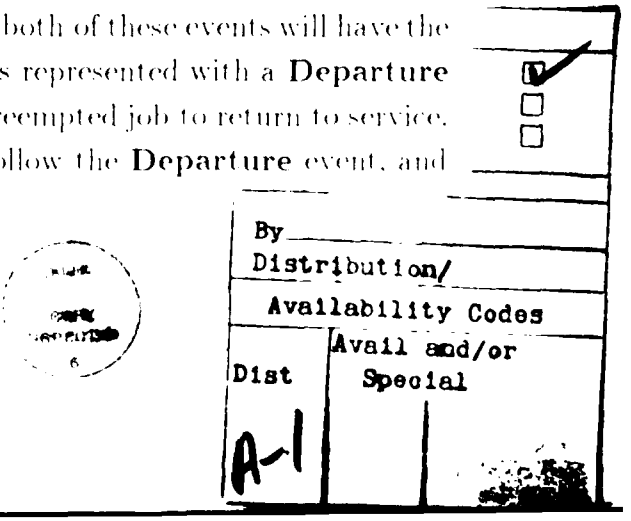
# 2 The Discrete Event View

The parallel algorithm we propose is described in terms of manipulation and analysis of event lists. We assume that each queue has its own event list, organized as a linearly ordered doubly linked list. This simplifies the logic of the synchronization protocol, and permits easy run-time analysis of the pending events for a given queue. Since the list is for a single queue, its length should not get so long as to make the linear-time insertion cost too great. This section gives a brief summary of the events, their meaning and ordering.

We will assume that an event's representation includes the event type, the event time, the identity of the job, and the job's priority class.

Obviously, the first event for a given job and queue is the job arrival, an event we denote as **Arrival**. The job may have been sent from another queue or it may be the result of an external arrival process. At some point the job enters the server for the first time, an event we denote by **FirstServiceEntry**. If the job enters service immediately we will place the **FirstServiceEntry** event directly after the **Arrival** event in the event list; of course, both events have the same time-stamp. Once a job enters service it may be preempted by the arrival of a job with higher priority. Should this occur, a **LeaveService** event is inserted immediately prior to the preemptive **Arrival** event. Again, both of these events will have the same time-stamp. The departure of a job from the queue is represented with a **Departure** event. The departure of a high priority job may permit a preempted job to return to service. In this case an **ReenterService** event will immediately follow the **Departure** event, and have the same time-stamp.





If two job arrivals have the same time-stamp we will order them by priority class, with the higher priority job occurring first.

The arrangement we describe has the useful property that one can easily analyze the event list to determine whether an **Arrival** ought to place its job immediately into service—one need only examine the event just prior to the **Arrival**. Obviously the server is busy if the prior event is a **FirstServiceEntry** or **ReenterService** event for a job of equal or higher priority, because every departure from service is marked with a **LeaveService** or **Departure** event. However, the arrival may enter service if the prior event relates to a lower priority job or is a **Departure** event for any priority. It may be that the **Arrival** has no prior event in the event list. We will assume the existence of a *queue-state* record to describe the queue state at the time of the first event in the list. The queue state will indicate whether the server is busy, and what class of job is receiving service, if any.

It is also easy to determine the next time a class $k$ job may enter service, relative to any event in the list. Given a pointer to the reference event we need only scan forward looking for the first **Departure** event (which may be the event initially pointed at) whose immediate successor either has a time-stamp strictly larger than the **Departure**, or is an event for a job with priority lower than $k$. Once we determine the next time $t$ a class $k$ job may receive service it is a simple matter to compute the length of time the job may receive service before being preempted: we scan forward looking for the first **Arrival** event for a job of higher priority. The class $k$ job may receive service time up to the difference between the **Arrival** event's time-stamp and $t$.

Our algorithms will make frequent use of these abilities to determine the server's status, find the next time a class $k$ job may enter service, and determine maximal service intervals.

# 3 Single Class Protocol

The synchronization method we develop here is the concurrent application to different job classes of a protocol we have developed and analyzed elsewhere [10]. The earlier protocol is now reviewed.

Imagine a queueing network that satisfies the assumptions outlined in §1, save that there is *only one* job class. The queueing discipline is non-preemptive; whenever a job enters service we can exactly predict when the job leaves service, and the queue to which it departs. Parallelism is enhanced if, whenever a job enters service, we immediately report its future arrival to the next queue the job will visit. We call this the *pre-sending* of the departure *job-message*. The time-stamp on the pre-sent message is the arrival time of the job-message, **not** the clock value of the sending queue. The non-preemptiveness of the queueing discipline

provides us with the needed lookahead.

We assume an implementation where each queue has its own event list. We assume that the $M$ queues in the network have been distributed among $P$ processors in a multiprocessor system. A description of a testbed implementing this protocol is found in [9]; performance measurements from various problems are reported in [10].

Suppose the entire simulation has advanced up to time $t$, so that all job-messages caused by jobs entering service before $t$ have already been sent and received. From among the jobs in queue at $t$ and the jobs known to arrive at $Q_i$ after $t$ we can identify the next job $J$ to enter service, provided that no further jobs arrive at $Q_i$ before $J$ does enter service. Because we are able to select a job's service time upon receipt of the job-message reporting its arrival, we are able to predict the time $\delta_i(t)$ when $J$ would complete service: $t + \max\{r_i, a_i - t\} + s_i$, where $r_i$ is the residual service time of any job receiving service at $t$, $a_i$ is $J$'s arrival time, and $s_i$ is $J$'s service requirement. $\delta_i(t)$ is defined to be $\infty$ if there are no known jobs enqueued or in the list of job-arrivals. Because we pre-send job departures, $\delta_i(t)$ is a lower bound on the time of the next job-message $Q_i$ will send, provided it receives no further job arrivals before $J$ enters service.

Finally, define

$$\delta(t) = \min_{\text{all queues } Q_i} \{\delta_i(t)\}. \tag{1}$$

The protocol is very simple, and is given in Figure 1. Its net effect is to define a window of global simulation time, $[w_n, \delta(w_n))$. The window is defined so that all queues may be simulated completely in parallel over this range of time. Once all queues have been so simulated the protocol is engaged again to define a new window.

The synchronization is implemented by having each processor determine the minimal $\delta_i(t)$ among all queues for which the processor is responsible. The processors then engage in a global min-reduction, which can be accomplished in $O(\log P)$ parallel steps on many parallel architectures.

In [10] we have proved that the protocol is "safe", meaning that no job-message will ever be received with a time-stamp less than a processor's simulation clock.

**Theorem 1** *Let $[w_n, \delta(w_n))$ be a window established by the protocol. Then every job-message sent during the processing of $[w_n, \delta(w_n))$ has a time-stamp at least as large as $\delta(w_n)$.*
□

Another result which will prove to be useful is that a job entering service at $Q_i$ during the *nth* window has completion time $\delta_i(w_n)$.

**Lemma 2** *Suppose that at time $t \in [w_n, \delta(w_n))$ job $J$ enters service at queue $Q_i$. Then $J$ departs service at time $\delta_i(w_n)$.*

1. Define $w_1 = 0$, $n = 1$.

2. Given $w_n$, the processors cooperatively determine $\delta(w_n)$.

3. Each queue may be simulated in parallel with all others until the time of the event with least time-stamp at the queue is as large as $\delta(w_n)$. The processing of any event which puts a job into service must include pre-sending the associated departure's job-message to the job's next destination queue.

4. Queues receive the job-messages sent during the processing of $[w_n, \delta(w_n))$, select service times for the arriving jobs, and insert events into their event lists.

5. $n = n + 1$. Set $w_n = \delta(w_{n-1})$. Goto step 2.

Figure 1: Synchronization protocol for single class network

**Proof:** $\delta_i(w_n)$ is the completion time of the next job, say $J'$, to enter service at $Q_i$. By Theorem 1 we know that no further jobs arrive at $Q_i$ before time $\delta(w_n) > t$. Furthermore, a job that enters service within a window cannot depart within that same window. Hence the job that was foreseen to have highest priority to receive service next after time $w_n$ is the same job that enters service, i.e., $J' = J$.
□

# 4 Priority Class Protocol

We now extend the technique of §3 to priority class networks. The key observation is that the behavior of a job in one class is completely unaffected by jobs in a lower priority class. This permits us to correctly generate the events for high priority jobs farther ahead in simulation time than lower priority jobs. We will do so in such a way that whenever we generate the event that a job $J$ first enters service at $Q_i$, the events at $Q_i$ for all jobs of higher priority will have already been generated past $J$'s completion time, say $t_c$. This means that we are

able to exactly compute $t_c$, and so immediately generate $J$'s departure event and subsequent arrival event at another queue. As with the earlier protocol, we have the ability to predict job completion times, at least if we can push the generation of high priority job events sufficiently far ahead of lower priority job events. Since we are also able (by assumption) to determine a job's service requirement upon receipt of the job-message reporting the arrival, we will be able to compute $\delta_i(t)$-type values for each class at each queue, and so employ the protocol described in §3 to each individual job class. In fact, we will apply that protocol simultaneously to different classes; each class will have its own window, and the individual class windows will be sufficiently far apart in simulation time to ensure the necessary ability to predict departure times.

Our protocol maintains separate windows for different classes. An individual window is defined just as it was for single class queueing networks. At the $nth$ synchronization point suppose we have generated all class $k$ events up through time $w_{n,k}$, for $k = 1, 2, \ldots, C$. Recalling that class $C$ has highest priority, we will always ensure that

$$w_{n,1} \leq w_{n,2} \leq \cdots \leq w_{n,C}.$$

At each queue $Q_i$, for each class $k$, we can identify the next job $J$ to enter service (barring further arrivals to the queue). We then estimate $J$'s completion time, $\delta_i(k, w_{n,k})$, using a method to be described in §4.1. Within each class $k$ we may then take the minimum estimate over all queues. This minimum is denoted $\delta(k, w_{n,k})$. Applying this rule to each class, we define a set of windows $[w_{n,k}, \delta(k, w_{n,k}))$ for $k = 1, \ldots, C$. Unless otherwise noted, the discussion to follow will assume that the windows are separated enough to yield the properties we exploit. Later we develop methods that ensure the separation.

A useful way to view our approach is to think of a queue's simulation time-line. A discrete-event simulation places events on the time-line; our protocol defines non-overlapping windows on the time-line, one window for each class. Within the window for class $k$ one finds job arrival and job departure events deposited there by the processing of previous windows. As we scan through the events in a window we will *expand* certain ones, further fleshing out the event-list. For example, within the class $k$ window we will expand a class $k$ **Arrival** event into a "first-time-in-service" event (**FirstServiceEntry**) if the arriving job goes immediately into service. Likewise, we expand a **Departure** event from any class to include a class $k$ **FirstServiceEntry** event, if the departure permits a class $k$ job $J$ to enter service. The **FirstServiceEntry** event is in turn expanded into various service departure (**LeaveService**), service reentry (**ReenterService**), and finally queue departure (**Departure**) events to reflect the effects of preemption on $J$. The process of scanning through a window's events in increasing time-stamp order and expanding them is called a *window expansion*. In the course of a window expansion no event is ever removed from the

event list. One should think of it as building up the correct sequence of queue events, rather than as an execution of known events. Event execution will occur in the lowest priority window, where we know that all events have been generated.

One special problem arises if the network is open. At every queue where external jobs may arrive we maintain separate arrival processes for each job class. The processing of an external arrival for class $k$ includes the generation of the next external class $k$ arrival. This arrangment permits a queue $Q_i$ to correctly incorporate the external arrivals when computing the $\delta_i(k, w_{n,k})$ values.

We would like to expand every priority class window once between each pair of global synchronization points. This is not always possible, but we do make every effort to permit the concurrent expansion of different class windows. This improves performance by amortizing the synchronization cost over more computation. Following a synchronization the edges of newly expanded windows are moved forward to cover a new region. A separation test is applied to determine which windows may be safely expanded. Following these expansions the synchronization protocol is engaged once again. Eventually every simulation time instant will have been contained within a window once for every class.

The complete state of a queue at simulation time $t$ is not known until after the window for each class has encompassed $t$. Thus, the actual simulation with its attendant statistics collection and system measurement (presumably the point of the simulation) is performed on events within the lowest priority window, after the lowest priority window is fully expanded. These ideas are illustrated in Figure 2.

The scheme described above requires the solution of three inter-related problems.

1. For each queue $Q_i$, class $k$, and lower window edge $w_{n,k}$ we must be able to compute the departure time $\delta_i(k, w_{n,k})$ of the next job $J$ to enter service at $Q_i$, barring further arrivals. This computation must consider all the preemptions that may interrupt $J$'s service; it must consider the value of $J$'s residual service following a preemption.

2. We must design a scheme for expanding events and populating a queue's simulation time-line.

3. We must ensure that before we expand the **FirstServiceEntry** event for job $J$, we have already generated all events at $Q_i$ for jobs of higher priority, up through the time of $J$'s departure from $Q_i$. That is, if $J$ has priority $k$ we must ensure that the class $k+1$ window is sufficiently separated from the class $k$ window.

These problems are now addressed in turn.

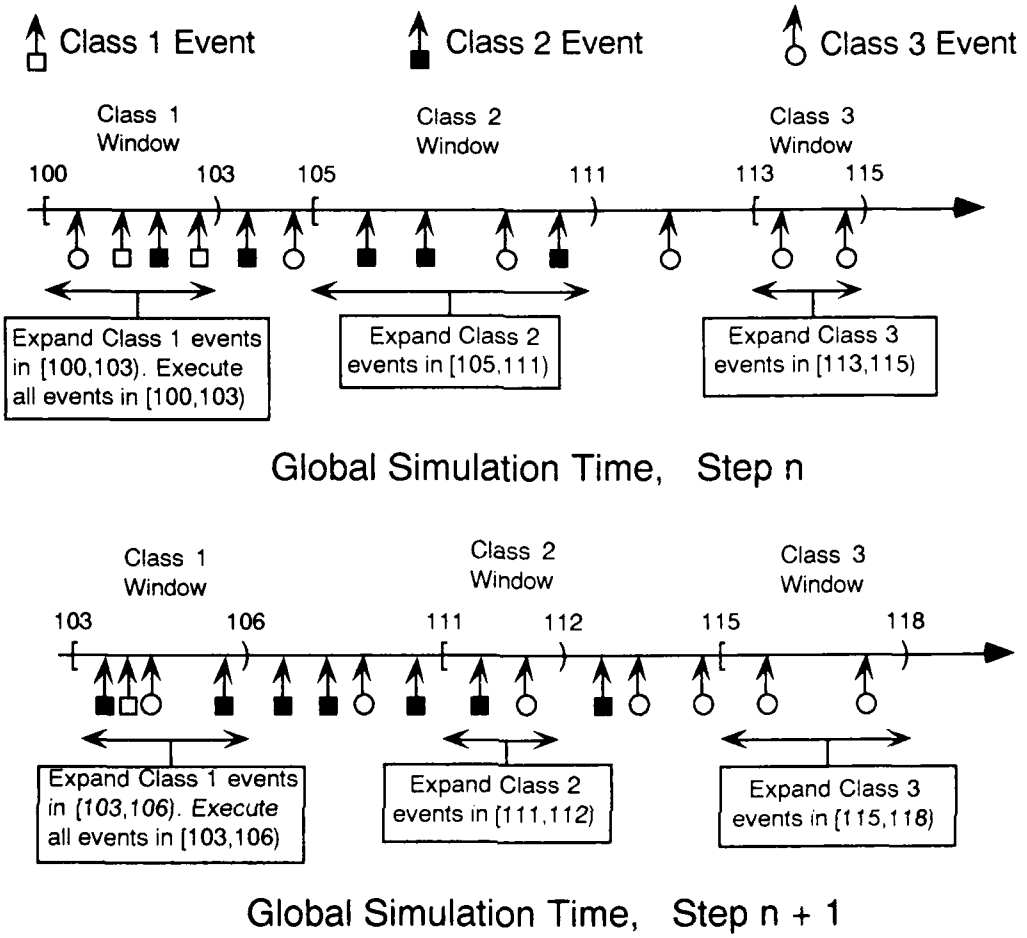



Figure 2: Class windows on simulation time-line

## 4.1 Computing a lower bound on departure time

The computation of a lower bound on a job's departure time is straightforward. We provide each queue with its own event list, maintained in increasing time-stamp order as a doubly linked list. This single data structure will serve all priority classes. We also need a queue status record that records the status of the server (is the server idle or busy, what class of job is in service) just prior to the time of the first event in the list. For each priority class we maintain a *Start* pointer into the list reflecting the point up to which all events for that class have already been expanded. This arrangement is illustrated in Figure 3. In addition, for each priority class at each queue we maintain a *queued-list* of jobs which have not yet

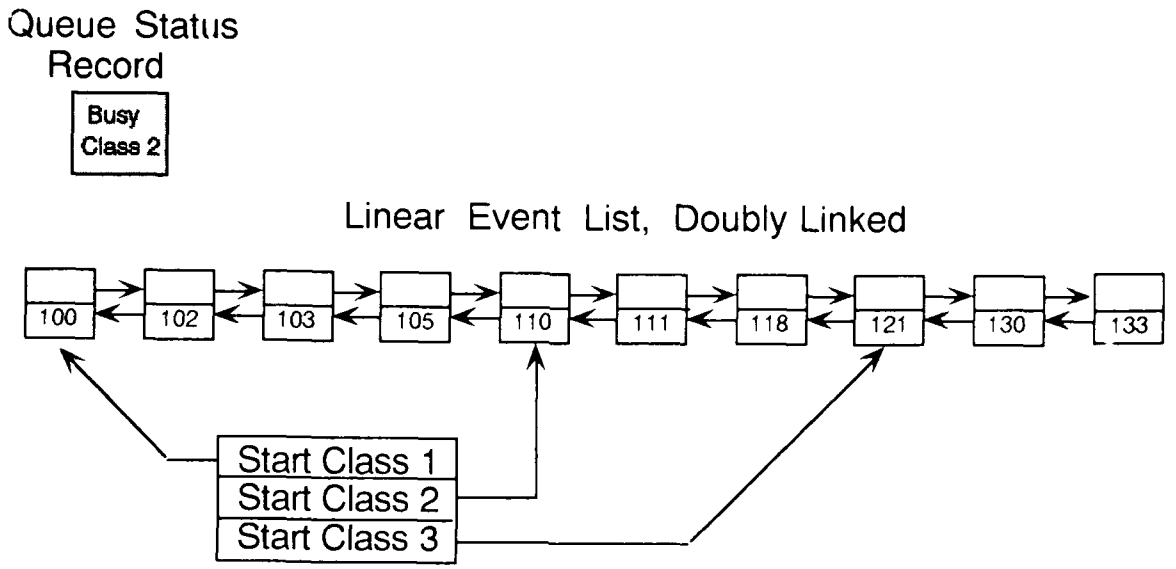



Figure 3: Data structures for concurrent class simulation

had their service and departure events generated. Each such list for each class is maintained in a priority heap based on the intra-class priority scheme. For example, a LCFS priority scheme will order the jobs by decreasing arrival times.

Let $J$ in class $k$ be the next job at $Q_i$ to enter service after time $w_{n,k}$, barring further class $k$ arrivals to alter $J$'s dominant priority within class $k$. For the sake of illustration let us suppose that each time $J$ is preempted it must start over with the same service requirement, $s$. Let $t$ be $J$'s arrival time. Our first problem is to determine when $J$ enters service. There are two cases to consider, depending on whether $t \geq w_{n,k}$. If so, then $J$'s **Arrival** event has not yet been expanded. We enter the event list following the class $k$ *Start* pointer, and scan forward until we find $J$'s **Arrival** event. We then determine whether the server is busy at that point, using the method described in §2. If $J$ does not immediately enter service we scan forward searching for the server's first idle period (again using a method described in §2). The second case to consider is when $t < w_{n,k}$. The fact that $J$ has already arrived and is not in service implies that the server is busy. We may therefore begin a scan forward from the class $k$ *Start* event to find the server's next idle period.

Having determined when $J$ first goes into service we must now determine when it departs the queue. Assume that our scanning pointer is positioned at an event whose time $t_e$ is equal

to $J$'s first entry into service (this event will not be $J$'s **FirstServiceEntry** event, as that event has not yet been generated). $J$ will finish service at time $t_e + s$ if the next **Arrival** event in the list for a higher priority job occurs after $t_e + s$. Otherwise $J$ is preempted before completing its service, and we must scan forward to find the next idle period where $J$ might reenter service. This process continues until we find a long enough service period, and so compute $J$'s completion time.

If we scan past the end of the event list we'll assume that $J$ starts its service at the time of the last event in the list.

It may happen that this procedure is used to predict $J$'s departure time $t_c$ before all events for higher priority jobs have been generated past $t_c$. We can easily detect this occurrence by determining whether $t_c$ exceeds $w_{n,k+1}$ (the lower window edge of the next highest priority class). Note that an erroneous completion time is still a lower bound on $J$'s completion time—any unaccounted for higher priority jobs can only serve to delay $J$ by preemption. Using $\delta_i(k, w_{n,k}) = t_c$ still provides a lower bound on the time of the next job-message $Q_i$ will send; we may thus compute $\delta(k, w_{n,k})$ type windows with the assurance that Theorem 1 still holds.

## 4.2 Window Expansion

Suppose we have just established the next class $k$ window $[w_{n,k}, \delta(k, w_{n,k}))$. As the first step, for every queue we move the class $k$ *Start* pointer up to point at the queue's first event in the window. The window encompasses an interval of simulation time which may have class $k$ **Arrival** and/or **Departure** events deposited there by the processing of previous windows. It may also have completely expanded events for jobs from higher priority classes. The window expansion process scans forward through the list of all events in $[w_n, \delta(k, w_{n,k}))$ expanding any class $k$ event that it can according to the rules described below.

Every class $k$ **Arrival**, **Departure**, and **FirstServiceEntry** event is expanded, as follows. To expand an **Arrival** event for $Q_i$ we first determine whether the arrival goes immediately into service. We can determine whether the server is busy at the time of the arrival, as described in §2. We know that if the server is busy, it is servicing a job with priority at least as high as the arrival. In this case the job is placed in the class $k$ queued-list at $Q_i$, and the event list remains unaltered. If the job *is* to enter service, we generate a **FirstServiceEntry** event for it at the arrival time, and insert that event directly following the **Arrival** event. The **FirstServiceEntry** event will be expanded as the next step of the window expansion.

If we encounter a class $k$ **Departure** event we know that it is possible to put another class $k$ job into service. The highest priority job in the class $k$ queued-list is removed, and a **FirstServiceEntry** event for it is inserted directly following the **Departure** event, with

the same time-stamp. The only possible hindrance is if a higher priority job just happens to arrive at the same instant as the class **Departure** departure—a situation which is easily detected.

The expansion of a **FirstServiceEntry** event for a class $k$ job, say $J$, is closely related to the departure time prediction procedure. Suppose that the **FirstServiceEntry** event has time-stamp $t$ and that $J$ requires $s$ units of service. We must determine the maximal service burst available to $J$. We do so by scanning forward until we either reach the end of the list, or encounter an **Arrival** event for a higher priority job. If the gap of time between $t$ and the preempting arrival is greater than $s$ we simply generate a **Departure** event for $J$ at time $t + s$, and send a job-message reporting a new arrival at time $t + s$ to $J$'s next destination. If $J$ cannot complete service we must record the time it spent in service. We generate a **LeaveService** event with the same time-stamp as the preempting **Arrival** event, and insert it directly prior to the preempting event. Then we scan forward looking for the server's next idle period. Once it is identified we generate an **ReenterService** event for $J$, and insert it directly following the **Departure** event which frees the server. Again we test to see if $J$ receives sufficient service before preemption. If not, we generate another **LeaveService** event and repeat the process; if so, we generate a **Departure** event and send a job-message.

## 4.3 Keeping Windows Separated

The correctness of the expansion for a class $k$ window $[w_{n,k}, \delta(k, w_{n,k}))$ depends on having already expanded all higher priority jobs sufficiently far into the future. In this section we determine how to ensure sufficient separation between class windows. The expansion of a class $k$ window will be correct if, during the expansion of a **FirstServiceEntry** event for job $J$, all events for higher priority jobs have already been expanded past the time of $J$'s completion, say $t_c$. In other words, we must have $t_c < w_{n,k+1}$.

Recalling equation (1), $\delta(k, w_{n,k})$ is the minimum "next-served-job-completion-time" for class $k$. Now suppose that in addition to computing this minimum we compute the *maximum* value

$$\tau(k, w_{n,k}) = \max_{\text{all queues } Q_i} \{\delta_i(k, w_{n,k})\}. \tag{2}$$

The following lemma uses this value to define a test for sufficient separability.

**Lemma 3** *Suppose that $\tau(k, w_{n,k}) < w_{n,k+1}$ for some class $k < C$. Let $J$ be any class $k$ job that enters service at $Q_i$ for the first time during $[w_{n,k}, \delta(k, w_{n,k}))$. Then the expansion of $J$'s* **FirstServiceEntry** *event is correct, meaning that the class $k$ window is expandable.*

**Proof:** By Lemma 2 we know that $J$'s estimated completion time defines the value $\delta_i(k, w_{n,k})$ computed during the construction of $\delta(k, w_{n,k})$. That estimate is constructed using the departure time prediction mechanism described in §4.1. The estimate is exact if at the point it is made all higher priority events have already been generated past time $\delta_i(k, w_{n,k})$. We know that all higher priority events up through time $\delta(k+1, w_{n-1,k+1}) = w_{n,k+1}$ have already been generated. Furthermore, $w_{n,k+1} > \tau(k, w_{n,k}) \geq \delta_i(k, w_{n,k})$. Thus all higher priority events are accounted for in the expansion of $J$, so that the expansion is correct.
□

Lemma 3 tells us whether the window for a given class can safely be expanded. Given $[w_{n,k}, \delta(k, w_{n,k}))$ and $\tau(k, w_{n,k})$ for each class $k$ we may identify all expandable windows between two synchronization steps. These windows are expanded, and new windows for the expanded classes are positioned. We adopt the notional convention that $w_{n+1,k} = w_{n,k}$ for any class $k$ whose window is *not* expanded during the $nth$ step. The expansions at step $n$ may permit the expansion during step $n+1$ of windows which failed to satisfy Lemma 3 at step $n$. However, a certain amount of recalculation of $\tau$ values is unavoidable, as we now illustrate.

Suppose the class $k$ window fails to satisfy the hypothesis of Lemma 3, and so is not expanded during the $nth$ step. As part of defining the $(n+1)st$ step we must recompute $\delta_i(k, w_{n+1,k})$ for any queue $Q_i$ where during step $n$ we observed $\delta_i(k, w_{n,k}) \geq w_{n,k+1}$. The need for recomputation follows from the fact that $\delta_i(k, w_{n,k})$ is then only a lower bound, whereas if $\delta_i(k, w_{n,k}) < w_{n,k+1}$ we would know it to be exact by Lemma 2. Consequently, the job whose estimated completion time defined $\tau(k, w_{n,k})$ may actually complete at a later time—possibly later than $w_{n+1,k+1}$. We can only safely expand class $k$ after recomputing $\tau(k, w_{n+1,k})$, and observing that its new value is less than $w_{n+1,k+1}$.

These observations lead us to a statement of the complete priority class algorithm, given in Figure 4.

Observe that the highest priority class window will always be expandable. However, the algorithm above does nothing to prohibit the highest priority class from moving its window arbitrarily far into the future. Eventually memory constraints will prohibit unconstrained advances. This problem is easily fore-stalled by including a *maximal* separability check. For example, to keep the class $k$ window from advancing too far we might require $w_{n,k} - \tau(k-1, w_{n,k-1}) < \Delta_k$, for some $\Delta_k > 0$ as a further requirement for the expandability of the class $k$ window. The inclusion of this constraint does not affect the fact that at every step, at least one window is expandable.

1. $w_{1,1} = \cdots = w_{1,C} = 0$, and $n = 1$.

2. Given $w_{n,k}$ for $k = 1, \ldots, C$ the processors cooperatively determine $\delta(k, w_{n,k})$ and $\tau(k, w_{n,k})$ for $k = 1, \ldots, C$. Each $\tau(k, w_{n,k})$ value is explicitly recomputed regardless of whether the class window was expanded during step $n-1$. $\delta(k', w_{n,k'})$ need not (but may) be recomputed for a class $k'$ whose window was not expanded at step $n-1$.

3. Identify each *expandable* class: any class $k$ for which $\tau(k, w_{n,k}) < w_{n,k+1}$.

4. For every expandable class, say $k$, the class $k$ events for each queue may be expanded, in parallel with all other queues.

5. If the lowest priority class (class 1) was expanded, then execute all events for all classes in $[w_{n,1}, \delta(1, w_{n,1}))$. Release the memory used to represent these events.

6. Queues receive the job-messages sent during the window expansions, select service times for the arriving jobs, and insert **Arrival** events into their event lists.

7. $n = n + 1$. For every expanded class $k$ set $w_{n,k} = \delta(k, w_{k,n-1})$. For every non-expanded class $k'$ set $w_{n,k'} = w_{n-1,k'}$. Goto step 2.

Figure 4: Synchronization protocol for priority class networks

**Lemma 4** *For every $n$, at least one window is expandable at step $n$.*

**Proof:** If class $C$ is expandable we are done. If class $C$ is not expandable, it is constrained by having $w_{n,C} - \tau(C-1, w_{n,C-1}) > \Delta_C$. This implies that class $C-1$ is expandable, unless it is constrained by $w_{n,C-1} - \tau(C-2, w_{n,C-2}) > \Delta_{C-1}$. We can repeat the argument backwards through classes until either we find an expandable window $k > 1$, or we reach class 1. In the latter case, class 1 has no maximal separability constraint on a lower class, and so must be expandable.
□

It is also straightforward to modify the algorithm in order to support the "rolling back" of a window. This would be needed if midway through the expansion of a window we discover that dynamic memory is exhausted. It is straightforward to "undo" events in the list, provided we can accurately reconstruct them when more memory is available.

We need only three additional activities in order to support rollback. The first is purely notational. Recall that we defined $w_{n+1,k} = w_{n,k}$ whenever the class $k$ window was not expandable. This convention must change, because we need the ability to rollback only some subset of windows in order to free up memory. Thus, we define a synchronization step number $n_k$ for each class $k$, and advance it only for those steps where the class $k$ window is expandable. The second activity is to stamp every event with the identity of the synchronization step (now localized to class) during which the event was inserted into the list. Then if we decide to roll the class $k$ window back from its step $n_k$ position to its step $n_k - j$ position, we simply scan through the list backwards, releasing the space for any event inserted during and after step $n_k - j$. The last activity is to periodically checkpoint the state of class $k$ activity. When one rolls back one must rollback to a checkpoint. Once the state is restored, the simulation proceeds as before with the assurance that exactly the same set of events will be regenerated. One does presume that the sequence of window expansions is somehow modified so that the memory is not exhausted again. For example, following a rollback of the class $k$ window we might tighten the separation constraint $\Delta_k$.

There is an important difference between this style of rollback, and the rollback used in optimistic systems [5]. Our rollbacks reclaim the space used to store correct events; the exact same set of events will later be regenerated. A queue therefore never sends a false job-message which will have to be undone. Thus, message cancellation is not needed, and rollback in one class will not directly cause a rollback in another class.

# 5 Analysis

In this section we develop an analysis of our method. We show that as the simulation model increases in size—all other things being constant—the average number of events executed or expanded between two synchronizations is $\Omega(MD + \sqrt{M})$, where $D$ is a minimal service time and $M$ is the number of queues. This demonstrates that on sufficiently large problems there will be enough parallel workload to achieve good performance on coarse or medium grained parallel architectures. We also show that the algorithm's expected memory requirement is $O(CM \log M)$.

## 5.1 Events Processed Per Window

Our analysis is based on results developed for the single-class model in [10]. To simplify the notation and description over that in [10] we will assume that in each class, each queue in the network has precisely the same characteristics, and that as the network size is increased these characteristics remain unaltered. This situation occurs when the network is completely homogeneous—no queue can be distinguished topologically from any other. For example, toroidal and hypercube networks satisfy this assumption. We let $\lambda_k$ be the arrival rate of class $k$ jobs at each queue, regardless of the number of queues in the network. $\lambda_k$ includes external and internal arrivals—consequently its value will not be known prior to running the simulation, because the internal arrival rate depends on the simulation's behavior. We assume that the service requirement of a class $k$ job is distributed as $D_k + \exp\{\mu_k\}$: a constant $D_k$ plus an exponential with mean $\mu_k$. Our discussion of single-class networks will drop the class subscript.

We have the following result, adapted from [10], for a homogeneous single class network with $M$ queues:

**Theorem 5** *The limiting value* $\lim_{n\to\infty} E[\delta(w_n) - w_n]$ *is approximately bounded from below by* $D + \mu/\sqrt{M}$.
□

This theorem makes a statement about the mean width of a synchronization window. In order to bound the mean number of events processed we must consider the density of events on the simulation time line. If jobs arrive at rate $\lambda$, and each job requires at least two events, then the density of events is at least $2\lambda$ per unit simulation time. We can therefore bound the equilibrium mean number of events processed in a window by multiplying the density times the window width: $2M\lambda(D + \mu/\sqrt{M}) = \Omega(MD + \sqrt{M})$. Thus, as $M$ increases the average number of events processed in a window increases at least as $\sqrt{M}$. Furthermore, if the service times have a minimal strictly positive component then the increase is super-linear.

It is straightforward to modify this result for priority classes. Let us now make a distinction between a class $k$ job's service requirement, and its so-called *service-lag*, the total time between when a job enters service for the first time and when it departs the queue. A job's service-lag is always at least as large as its service requirement. The distribution of the service-lags comprising the $\delta_i(w_{n,k})$ values are all *stochastically larger* [12] than the service requirement distribution. This means that probabilistically a random variable from service-lag distribution tends to be larger than a random variable from the service requirement distribution. Hence, a "window" built on the basis of service requirements will be always be smaller than the real window which is based on service-lags. This implies the

extended result:

**Theorem 6** *For every class $k = 1, \ldots, C$, the limiting value $\lim_{n\to\infty} E[\delta(w_{n,k}) - w_{n,k}]$ is approximately bounded from below by $D_k + \mu_k/\sqrt{M}$.*
□

Since class $k$ jobs arrive at a queue at rate $\lambda_k$, and each job generates at least 2 events, the equilibrium mean number of events falling within a class $k$ window is $\Omega(MD + \sqrt{M})$.

Lemma 4 shows that at least one window is expandable every step. It follows that the limiting average number of events processed per step is $\Omega(MD_{\min} + \sqrt{M})$, where $D_{\min} = \min_k\{D_k\}$.

The practical importance of Theorem 5 has been verified empirically, as reported in [10]. On large models of several different types of problems (infinite server queueing networks, logic networks, Game of Life, timed petri nets) we have achieved excellent speedups on a 32-node Intel iPSC/2 [1]. We may have confidence then that the proposed protocol for priority class networks will achieve good performance on sufficiently large queueing networks.

## 5.2 Memory Requirements

Our simulation technique obviously requires more memory than an equivalent serial simulation, because more of a queue's event list is retained in memory than is necessary in the serial case. A natural question asks how much memory is required, and how those requirements change as the simulation model increases in size. Under the assumption that service times are composed of a constant plus an exponential, we will show that memory requirements are $O(CM \log M)$.

Consider again the situation where we increase the number of queues in such a way that all essential queueing characteristics remain the same. In particular, the overall rate that events appear at a queue per unit simulation time, say $\Lambda_Q$, does not change. Now consider an implementation of our protocol. Ideally the windows are sufficiently separated, meaning that $\tau(k-1, w_{n,k-1}) < w_{n,k}$ for $k = 2, \ldots, C$. Let us also assume that we restrain a window from advancing more than $\Delta$ units of simulation time past its predecessor. Defining $S_{n,k} = \tau(k, w_{n,k}) - w_{n,k}$, the span of simulation time represented in the queue is thus no more than $(C-1)\Delta + \sum_{k=1}^{C} S_{n,k}$. The mean number of events in the event list is then no larger than $\Lambda_Q(C-1)\Delta + \sum_{k=1}^{C-1} E[S_{n,k}]$. Recall the definition of the $\tau$ function, given as equation (2). We are interested in the behavior of an expected maximum, as $M$ increases.

First consider the highest priority class. Each $\delta_i$ value involved in $\tau(C, w_{n,C})$ is the sum of the residual time of a job in service, plus the full service time of another job. Now the residual is stochastically dominated by the full service distribution. Thus the distribution of

each $\delta_i(C, w_{n,C}) - w_{n,C}$ value is stochastically dominated by the sum of $2D_C$ plus an Erlang-2 with parameter $\mu_C$ (i.e., the sum of two exponentials, each with mean $\mu_C$). Assuming independence among the $\delta_i$ values [1] $S_{n,C}$ is stochastically dominated by $2D$ plus the maximum of $M$ identically distributed Erlang-2's.

Analysis of hazard rate functions [12] helps to quantify a bound on $S_{n,C}$. Let $f(t)$ and $\bar{F}(t)$ respectively be the probability density function and one minus the cumulative distribution function for some probability distribution. The distribution's hazard rate function is defined to be $h(t) = f(t)/\bar{F}(t)$; informally, $h(t)$ is the probability that a random variable from the given distribution achieves value $t$, given that it is at least as large as $t$. For example, an exponential with mean $\mu$ has a hazard rate function $h(t) = 1/\mu$; this is just another statement of the exponential's memoryless property.

We will use the fact that if (i) $m_1$ is the maximum of $M$ independent and identically distributed (*iid*) random variables having hazard rate function $h(t)$, (ii) $m_2$ is the maximum of $M$ *iid* random variables having hazard rate function $g(t)$, and (iii) $h(t) \geq g(t)$ for all $t$, then $m_2$ stochastically dominates $m_1$. In order to bound $E[S_{n,C}]$ from above we seek a random variable with a smaller hazard rate than an Erlang-2.

The hazard rate function for an Erlang-2 increases monotonically, reaching an asymptote at $1/\mu_C$ [13]. This is not surprising, given the intuitive understanding of the hazard rate function. We can view an Erlang-2 as the "lifetime" of a serial system with two stages, each of whose lifetimes are exponential. As $t$ increases, the probability that the first stage has already completed increases. If the system is into the second stage, then its hazard rate function is that of the second exponential--the constant function $1/\mu_C$. So the longer the system life is known to be, the more likely it is that the remaining life is that of an exponential. Let $h(t)$ be the hazard rate function for the Erlang-2 distribution with parameter $\mu_C$. It turns out that $h(0) = 0$, and $h(t)$ increases monotonically to approach $1/\mu_C$. We may therefore choose a value $t_0$ such that $h(t_0) = 1/(2\mu_C)$. Let $X$ be an Erlang-2, and define a random variable $Z$ as follows. If $X < t_0$ then $Z = t_0$. Otherwise, choose an independent exponentially distributed random variable with mean $2\mu_C$. The hazard rate function $g(t)$ for $Z$'s distribution is 0 for $t \leq t_0$, and is $1/(2\mu_C)$ for $t > t_0$; thus $h(t) \geq g(t)$ for all $t$. We can therefore bound the expected maximum of $M$ *iid* Erlang-2's with the expected maximum of $M$ *iid* random variables $Z_1, \ldots, Z_M$, having $Z$'s distribution. Clearly the latter expectation is no larger than $t_0$ plus the expected maximum of $M$ *iid* exponentials with mean $2\mu_C$. This latter expectation is commonly known [6] to be no greater than $2\mu_C \log M$. Thus we have shown that $E[S_{n,C}] \leq 2\mu_C \log M$.

Lower priority classes are treated similarly, at least approximately. Let $\rho_k$ be the average

[1] This is not rigorously true, but suffices for these complexity approximations.

server utilization for class $k$ jobs. The *shadow CPU* technique pioneered by Sevcik [7] for the quantitative analysis of priority class networks approximates the service-lag of a class $k < C$ job as an exponential with mean $(D_k + \mu_k)/(1 - \sum_{i=k+1}^{C} \rho_i)$. Making the same approximation, $S_{n,k}$ becomes a constant plus the maximum of $M$ Erlang-2's as before. Each $S_{n,k}$ may therefore be bounded by a constant times $\log M$.

The discussion above shows that memory requirements at each queue are $O(C \log M)$, so that the overall memory requirements are $O(CM \log M)$.

# 6 Summary

It has long been thought that good performance could not be achieved using a conservative synchronization mechanism for the simulation of priority class queueing networks. This conclusion followed from the observation that such networks fail to provide much lookahead, upon which all conservative methods rely. In this paper we develop a conservative technique that finds lookahead in a simple, yet obvious way. The behavior of a high priority job is completely unaffected by any lower priority jobs. We may therefore simulate high priority jobs farther ahead in simulation time than low priority jobs. When determining the departure time of a low priority job we will already have the complete history of all higher priority jobs that affect it.

Using this observation we describe and analyze a synchronization protocol for the parallel simulation of priority class queueing networks. The analysis shows that the average number of events that may be processed in parallel is $\Omega(MD + \sqrt{M})$, where $D$ is a minimal service time, and $M$ is the number of queues. This implies that on sufficiently large networks we can expect good performance on a coarse or medium grained multiprocessor, as the synchronization overheads are amortized over a great deal of parallel workload. We also show the expected memory requirements to be $O(CM \log M)$, and describe a simple mechanism that permits one to recover from exhausting memory, at the expense of re-computing events.

Our synchronization protocol may be overly conservative in the sense that no topological information about the network is used to increase the width of the class windows. Our protocol essentially assumes that any queue can receive the job whose completion defines $\delta(w_n)$. Other synchronous protocols such as the bounded-lag protocol [8] explicitly use topological information, and might be substituted for our own when defining class windows. The additional issue of ensuring separability would have to be treated.

The class of queueing networks we address is practical and important. But the contribution of this paper extends beyond just that of queueing networks. We have illustrated a new way in which one might find lookahead for exploitation by conservative methods. Similar

ideas can be applied to the simulation of any physical system where there is some sense of priority, or non-symmetric independence between subclasses of components.

# References


[1] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, Sept. 1989.

[2] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Distributed Simulation 1989*, pages 93–99. SCS Simulation Series, 1989.

[3] R. M. Fujimoto. Lookahead in parallel discrete event simulation. *Proceedings of the 1988 International Conference on Parallel Processing*, 3:34–41, August 1988.

[4] R. M. Fujimoto. A survey on parallel simulation. *Communications of the ACM*, 1990. To appear.

[5] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[6] D.E. Knuth. *The Art of Computer Programming, vol. 1.* Addison-Wesley, New York, 1968.

[7] E. Lazowska, J. Zahorjan, S. Graham, and K. Sevcik. *Quantitative System Performance.* Prentice-Hall, Englewood Cliffs,NJ, 1984.

[8] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[9] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[10] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. Technical Report 90-20, ICASE, 1990. Available from ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

[11] D.M. Nicol. Performance bounds on parallel self-initiating discrete event simulations. *ACM Trans. on Modeling and Computer Simulation*, 1(1), 1991. To appear. Also available as Technical Report 90-21 from ICASE, M.S. 132C, NASA Langley Research Center, Hampton, VA, 23665.

[12] H.S. Ross. *Stochastic Processes.* Wiley, New York, 1983.

[13] K.S. Trivedi. *Probability and Statistics, with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

NASA

# Report Documentation Page

| 1. Report No. | 2. Government Accession No. | 3. Recipient's Catalog No. |
|---|---|---|
| NASA CR-187438<br>ICASE Report No. 90-64 | | |

| 4. Title and Subtitle | 5. Report Date |
|---|---|
| CONSERVATIVE PARALLEL SIMULATION OF PRIORITY CLASS QUEUEING NETWORKS | September 1990 |
| | 6. Performing Organization Code |

| 7. Author(s) | 8. Performing Organization Report No. |
|---|---|
| David Nicol | 90-64 |
| | 10. Work Unit No. |
| | 505-90-21-01 |

| 9. Performing Organization Name and Address | 11. Contract or Grant No. |
|---|---|
| Institute for Computer Applications in Science and Engineering<br>Mail Stop 132C, NASA Langley Research Center<br>Hampton, VA 23665-5225 | NAS1-18605 |
| | 13. Type of Report and Period Covered |
| 12. Sponsoring Agency Name and Address<br>National Aeronautics and Space Administration<br>Langley Research Center<br>Hampton, VA 23665-5225 | Contractor Report |
| | 14. Sponsoring Agency Code |

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to IEEE Trans. on Parallel and Distributed Systems

Final Report

16. Abstract

This paper describes a conservative synchronization protocol for the parallel simulation of queueing networks having C job priority classes, where a job's class is fixed. This problem has long vexed designers of conservative synchronization protocols because of its seemingly poor ability to compute lookahead: the time of the next departure. For, a job in service having low priority can be preempted at any time by an arrival having higher priority and an arbitrarily small service time. Our solution is to skew the event generation activity so that events for higher priority jobs are generated farther ahead in simulated time than lower priority jobs. Thus, when a lower priority job enters service for the first time, all the higher priority jobs that may preempt it are already known and the job's departure time can be exactly predicted. Finally, we analyze the protocol and demonstrate that good performance can be expected on the simulation of large queueing networks.

| 17. Key Words (Suggested by Author(s)) | 18. Distribution Statement |
|---|---|
| parallel simulation, priority classes, preemption, queueing networks, parallel computing | 61 - Computer Programming and Software<br>66 - Systems Analysis<br><br>Unclassified - Unlimited |

| 19. Security Classif. (of this report) | 20. Security Classif. (of this page) | 21. No. of pages | 22. Price |
|---|---|---|---|
| Unclassified | Unclassified | 23 | A03 |