A Nonblocking Algorithm for the Distributed Simulation of FCFS Queueing Networks with Irreducible Markovian Routing [1]

Manish Gupta and Anurag Kumar
Dept. of Electrical Communication Engg.
Indian Institute of Science
Bangalore, 560 012, INDIA
e-mail: manish, anurag@ece.iisc.ernet.in

Abstract

In this paper we consider the distributed simulation of queueing networks of FCFS servers with infinite buffers, and irreducible Markovian routing. We first show that for either the conservative or optimistic synchronization protocols the simulation of such networks can prematurely *block* owing to event buffer exhaustion. Buffer exhaustion can occur in the simulator whether or not the simulator is stable, and, unlike simulators of feedforward networks, cannot be prevented by interprocessor flow control alone. We propose a simple technique (which we call *compactfication*), which, when used in conjunction with interprocessor flow control, prevents buffer exhaustion. This leads to a general algorithm, for both conservative and optimistic synchronization, that allows one to simulate the queueing network within the finite amount of memory available at each processor. For each algorithim presented we also provide the proof that it cannot get deadlocked owing to buffer exhaustion.

1 Introduction

In distributed discrete event simulation, the simulation of a physical process (PP) is partitioned into several logical processes (LPs) which are assigned to several processing elements. The time evolution of the simulation at the various LPs is synchronised by means of time stamped messages that flow between the LPs. We are concerned with the situation in which the messages are not just for synchronisation [2], [11], but also carry "work" which when done modifies the state of the receiving LP. A typical example is the distributed simulation of a queueing network model, in which one or more queues is assigned to each LP, and the messages indicate the motion of customers between the queues in the various LPs. The simulation makes correct progress if each LP processes the incoming events, from all other LPs, in time stamp order [2].

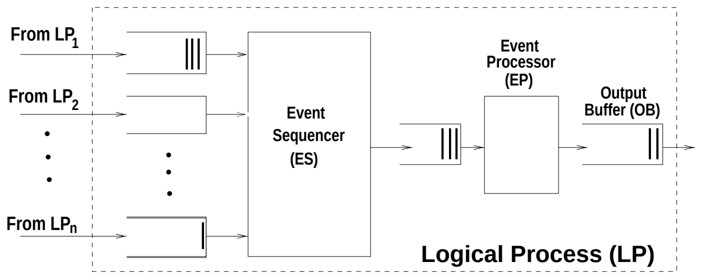

Figure 1: Schematic View of a Logical Process

We view each LP in terms of the model shown in Figure 1 [15]. Each LP comprises an input queue for each channel over which it can receive messages from other LPs. Since the messages must be processed in time stamp order, the event processor (EP) must be preceded by an event sequencer (ES). The messages must emerge from the ES in time stamp order; to achieve this the ES implements an event sequencing algorithm. Event sequencing algorithms fall into one of two classes: *conservative* or *optimistic.* A conservative ES allows a message to pass through only if it is

[1] This work was supported by the Education & Research Network (ERNET) Project of the Department of Electronics, Govt. of India.

sure that no lower time-stamped event can arrive in the (real-time) future [1], [11]. An optimistic ES, on the other hand, lets messages pass through without being sure that no lower time-stamped message can arrive in the future. If then a lower time-stamped message does arrive, corrective action is taken resulting in a *rollback* of the simulation [2], [6]. Another important component (though not always explicitly shown) is the Output Buffer (OB) which contains events that have been *processed* in the EP, and whose output events are *waiting* (events may have to wait because of flow-control between processors, etc.) to be sent to their destination.

Queueing Networks are widely used as a modelling tool in processor networks, manufacturing systems, etc. To get reliable measures of performance for complex queueing models, we may have to simulate these models for a "long" time and would not like the simulation to halt prematurely. We show in this paper that, in fact, for the distributed simulation of FCFS queueing networks with irreducible Markovian routing and infinite buffers, such an undesirable thing can happen. We show that the cause for the halting or blocking of the simulation is that the number of events in the system can become more than the available storage space and the simulation has to stop. This problem can be aggravated if some of the queues are working close to saturation. Both the conservative and the optimistic mechanisms have this problem. In the case of the optimistic mechanism our work is different from the usual memory management schemes [6], [7], [9], [10], [13], where the actual problem is the large and inefficient use of memory by the Time-Warp protocol. In the case of conservative methods our work can be said to be related to the problem of nonoptimality of space (see [7]) in *asynchronous* conservative simulation. This problem has not been discussed in literature and we feel that it is important for a simulationist to be aware of it in the detail presented in this paper. We show that, without modification, both the mechanisms (conservative and optimistic) are inadequate for the simulation of FCFS queueing network models with infinite buffers and irreducible Markovian routing. We, in this paper, also provide a solution for this inadequacy such that with both the mechanisms the simulation can run without blocking, only needing a finite amount of memory at each processor. The outline of the rest of the paper is as follows. In Section 2 we see how flow control between LPs helps to prevent instability of a distributed simulator of a feedforward queueing network and cannot help if we simulate a queueing network with irreducible routing. Section 3 considers a simple example to motivate the general algorithm for conservative simulation, which is presented in Section 4, with a fixed amount of memory at each processor. In Section 6 we carry over the above discussion to the optimistic paradigm. Some conclusions and details are left for discussion in Section 7.

2 A Key Difference

By way of example, we shall first bring out a key difference between the distributed simulators of open *feedforward* queueing networks, and open queueing networks with *irreducible*[1] Markovian (where a customer goes after finishing current service does not depend upon the sequence of queues visited prior to the present queue) routing.

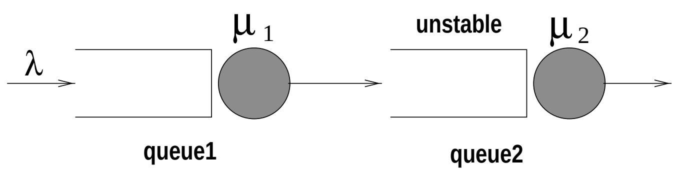

Figure 2: Jackson Network: Two Queues in Tandem

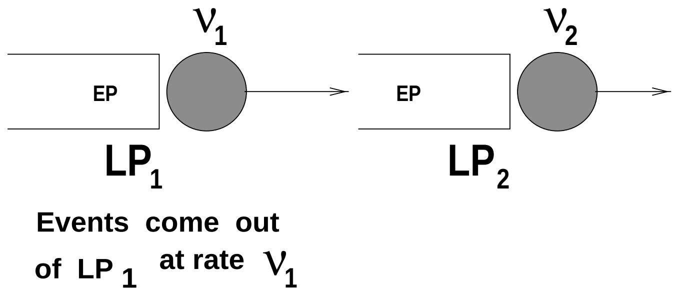

Figure 3: A Simulator for the Queueing Network in Figure 2

In Figure 2 we show a feedforward Jackson queueing network – two queues in tandem. The second queue is given to be **unstable**[2] (see [17] for a formal definition of stability). One way to simulate the queueing network of Figure 2 is depicted in Figure 3.

[1]Any customer in any queue has a *nonzero* probability of going to any queue (including the present queue), once it finishes service at the current queue.

[2]Informally speaking, by *instability* for any queue we mean that, the number of customers/events in that queue grow unboundedly with time.

We show in Figure 3 two LPs. LP_i simulates queue i (cf. Figure 2)[3] and lies on a processor with processing rate ν_i events per second, for $i = 1, 2$. Note that LP_1 also simulates the external Poisson arrivals to queue 1. It is given that $\nu_1 < \nu_2$. Now note that both LPs cannot be unstable. If we had not chosen $\nu_1 < \nu_2$, we still could have precluded instability for the simulator by having flow control between the two LPs. In essence one can choose processor speeds and apply flow control between processors to preclude instability of the distributed simulator even though the feedforward queueing network simulated is unstable. Thus the simulator can be run indefinitely without any requirement of more buffer space than already present. (We point out that it has been shown (see [15], [16]) that conservative simulators of feedforward queueing networks are unstable and thus flow-control is necessary.)

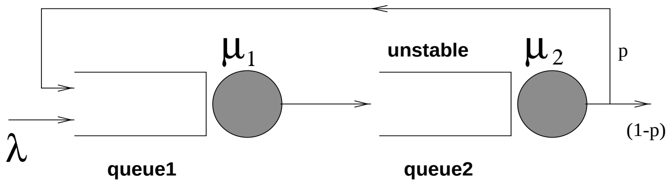

Figure 4: Jackson Network: Two Queues with a Feedback

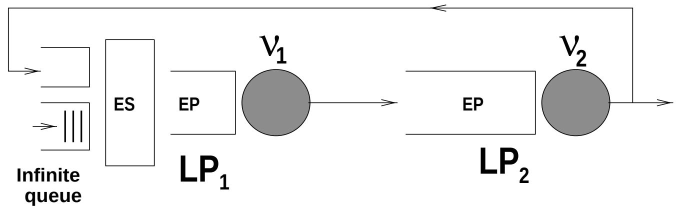

Figure 5: A Simulator for the Queueing Network in Figure 4

Now we contrast the above situation with the distributed simulator of a queueing network with irreducible routing. Consider the network in Figure 4. It is a Jackson network with queue 2 given to be unstable. External arrivals to queue 1 are at rate λ. The service rate at queue i is μ_i, $i = 1, 2$. At queue 2 the events after service leave the system with probability $1 - p$ or go to queue 1 otherwise.

[3]This is only an example; the analytical approach applies to more general mappings than the simulation of one physical queue per processor.

Next consider its distributed simulator shown Figure 5. LP_i simulates queue i and is situated on a processor with rate ν_i, $i = 1, 2$. Note that, the *infinite queue* at LP_1 models the external arrivals which can be generated when necessary. It is interesting now to observe that the simulator in Figure 5 is unstable no matter what be the processing speeds $\{\nu_i, i = 1, 2\}$ we choose and whether we have flow control or not. This follows from our result in [4]. Indeed we have the following result:

Theorem 1 *In a conservative distributed simulator of an* <u>*unstable*</u> *queueing network of FIFO queues, with infinite buffers and irreducible Markovian routing, the buffers at one of the processors* eventually *get exhausted and the simulation must stop prematurely.*

Sketch of Proof: The result follows from the main theorem in [4], that if the queueing network is unstable, the simulator is unstable (with or without interprocessor flow-control) with respect to the number of events in the system. Intuitively speaking, the large queue lengths in the unstable queueing network show up as a large number of events in the simulator. □

3 Motivation for an Algorithm

We shall use the example presented in the previous section and show how we can achieve stability of the simulator in Figure 5. This will act as a spring-board for the following section where we present the general algorithm.

We assume for expository simplicity that queue 2 in Figure 4 is unstable, and after time $t = 0$ there is no idle period for queue 2 and the queuelength goes to infinity. Thus the server at queue 2 is always *busy*, and the output stream after the first departure at queue 2 is a Poisson process with rate μ_2. This fact can be capitalized upon for redesigning the algorithm for the simulator. At LP_2 all those events that have finished service need not be *immediately* despatched to their destination. But if we do not want to send them immediately, we need to store them at LP_2 itself, and, if nothing else is done, large queues in PP will still appear as a large number of events at LP_2. Let $e_1, e_2, \ldots, e_k$ be the first k events in succession that finish service at LP_2 and let us assume that they all have to be routed to LP_1. Obviously, timestamp(e_1) $<$ timestamp(e_2) $< \ldots <$ timestamp(e_k). Again note that these epochs are a fragment of a Poisson process.

Suppose we keep the *triple* – the timestamp of e_1, the random number generator seed used to generate the service sample that the customer corresponding to e_1 receives at queue 2, and the number k — then we need not know anything else. If at a later time LP_2 wants to send say e_1 and e_2 to LP_1 then we can generate these two events using the triple, and after generating the two events the triple gets updated to – the timestamp of e_3, the new state of the random number generator, and the number $k-2$. Now consider the scenario where after sending e_1 and e_2 to LP_1, e_{k+1} finishes service at LP_2. The triple can now be simply updated by incrementing its third coordinate, i.e., changing $k-2$ to $k-1$. With this technique LP_2 can send LP_1 events (that are to be routed to LP_1) on demand from LP_1. This way LP_1's (or LP_2's) buffers do not "overflow" (to be proved below for the general case). Note, although the space required to hold events/messages is bounded, yet the integer counters can still overflow if the simulator is allowed to run indefinitely. Two points to be noted are that (i) the computational effort per event at LP_2 is higher than in the original conservative algorithm, and (ii) some kind of flow control is needed between LP_1 and LP_2. Obviously this technique is not computationally efficient, but we must appreciate that if we do not simulate this way (i.e., used our old distributed conservative simulation paradigm) we will surely run out of the buffer space (needed for storing events) at one of the processors, and then be forced to stop.

An important observation is that if in Figure 4 we chose a *stable* Jackson network, it is still not possible to simulate the network indefinitely because when large queue lengths occur in the queueing network, or, the network behaves as if it is unstable (this has a nonzero probability), then the simulator also shows the same behaviour and hence has a nonzero probability for one of its LPs to exhaust its buffer space, causing the simulation to end prematurely. We now present a theorem that formalises this last observation:

Theorem 2 *We assume that there is a nonzero probability for the number of customers in the queueing network to exceed any given number. In a conservative distributed simulator of a stable queueing network with FIFO queues, infinite buffers and irreducible Markovian routing, the buffers at one of the processors* eventually *gets exhausted and the simulation must end in finite time with probability one.*

Sketch of Proof: By hypothesis there is a nonzero probability for the number of customers in the queueing network to exceed any given number. When eventually atypical behaviour occurs in the queueing network it also induces a similar behaviour on its distributed simulator (this coupling can be seen more clearly once we consider the key inequality in the main result in [4]). Thus there is a nonzero probability for the number of events in the simulator to exceed the available buffers. This causes the simulation to stop prematurely. □

Note that the above result is an asymptotic result; as the running time of a simulation goes to infinity the simulation will deadlock with probability one. In many practical situations the probability of buffer exhaustion over a finite simulation time may be negligibly small.

4 Algorithm for the Conservative Case

In this section we present a simple technique, called "compactification", which when used with the conservative synchronization algorithm, allows one to simulate the queueing network within the given amount of memory. In the following it will be seen that the technique makes use of counters (which store integer values) and during the course of the simulation the value of the counter can be unbounded. Thus, though our technique precludes the possibility of the queues of events in the simulator from blowing up, yet it does not allow the simulation to run indefinitely.

4.1 Assumptions

1. The events are identical in all respects except for the timestamps that they carry.
2. For each queue of the queueing network simulated by an LP, there is a separate random stream for generating the service samples, and one for routing of the serviced events.
3. The queueing network simulated has First-In-First-Out queues, with infinite buffers, and irreducible Markovian routing.
4. Service times of customers in the queueing network are nonzero with probability one (this is necessary for a nonzero lookahead).
5. We simulate the work-in-the-system process (see [8]) for each of the queues and therefore output events can be immediately generated.

4.2 Algorithm to be added to the Basic Conservative Algorithm

Before we present the algorithm we define: Consider some queue Q simulated by an LP. Let $e_1, e_2, \ldots, e_k$ be the events corresponding to the departures from Q. Assume that $e_i : i = 1, 2, \ldots, k$ are served back-to-back in the PP (see Section 3). The process of compactly representing $e_i : i = 1, 2, \ldots, k$ in the output list of the LP (as we did in Section 3) is defined as **compactification**.

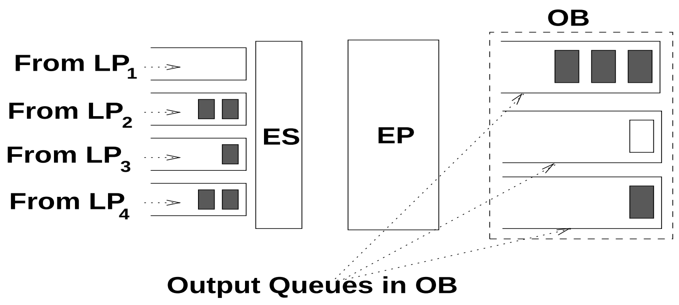

Figure 6: Partitioning of the events in the OB of an LP

Let the total number of LPs in the system be m, and define $\mathcal{LP} := \{1, 2, \ldots, m\}$. This following algorithm is to be executed at each LP_i, for $i \in \mathcal{LP}$, in addition to the basic conservative algorithm.

Algorithm : CON+ at LP_i

1. LP_i keeps a variable $\#sent_j$ for each LP_j ($j \in \mathcal{LP}$) from which it receives events. $\#sent_j$ corresponds to the number of tokens that LP_i has sent to LP_j. Each time LP_i receives an event from LP_j it decrements the corresponding variable $\#sent_j$ by one. Each time LP_i sends a token to LP_j it increments the corresponding variable $\#sent_j$ by one.
2. LP_i has a variable $\#received_j$ corresponding to each LP_j ($j \in \mathcal{LP}$) to which it sends events. The variable $\#received_j$ keeps the number of events that LP_i can still send to LP_j. Each time LP_i receives a token from LP_j it increments the corresponding variable $\#received_j$ by one. Each time LP_i sends an event to LP_j it decrements the corresponding variable $\#received_j$ by one. LP_i sends an event LP_j only if the corresponding $\#received_j$ is greater than zero.
3. For each queue of the queueing network simulated by LP_i, the LP maintains a separate queue of output events in the OB[4], i.e., the output events (which correspond to the departures of customers in the queues of the PP simulated by LP_i) are partitioned according to the queue of the PP (which is simulated by LP_i) to which they belong (see Figure 6). The event buffer space in the system is shared dynamically among all the simulated queues. However, there is always an event worth of buffer space allocated to each output queue. In Figure 6, we have shown that the middle queue in the OB is empty but still has one event worth of buffer space.
4. If LP_i reaches a state where it has no tokens to send to its sender LPs then it performs compactification. Compactification may be done only partially as and when required.
5. At the beginning of the simulation, LP_i sends at least one token to each LP that sends events to it. Further distribution of the remaining tokens may depend on the "particular" policy adopted. After an event is sequenced and then processed at LP_i, the resultant output event
   - may be immediately sent to the the destination, in which case the event may be removed from the event list,
   - may go to an empty output queue,
   - may be compactified with some event already in the output queue,
   - could not be compactified with the last event of the output queue in which it was put.

   Only in the first three cases a token should be sent to the source LP of this event, if the source LP does *not* have any token. Note that the compactification of a processed event with some event in the output queue results in freeing of an event buffer space.
6. Events in the output list (corresponding to all queues simulated) are sent to there destination in the increasing order of their timestamp.

4.3 Algorithm CON+ does not Deadlock

It is to be noted that our algorithm works just like any other conservative algorithm (e.g., there is some kind

[4]As mentioned in the Introduction and shown in Figure 1, an OB comprises those processed events in the event list of an LP whose output event is yet to be sent to its destination LP. In the conservative case an event can be deleted from the event list if the corresponding output event has been sent to its destination.

of a deadlock avoidance or breaking mechanism operating to resolve deadlocks due to event synchronization) but with the difference that at times it performs compactification to create space to accommodate more events. The only way our algorithm cannot work is if a deadlock (due to buffer constraints) occurs. When a deadlock occurs, no LP (in the whole system) is in a position to output an event to any other LP and all events which could be sequenced into the EP have been done by every LP. Note that, as in the basic conservative algorithm, this is possible because every LP is eventually able to discover which events in the ES it can process.

In the following proofs the names of events and their timestamps shall be denoted by E_0, E_1, etc., and depending on the context it should be clear whether we are referring to the event name or to its timestamp. For example, $E_1 < E_0$ means that the timestamp of E_1 is strictly smaller than the timestamp of E_2.

Lemma 1 *Suppose the system deadlocks, then the smallest timestamped event in the system is in the OB of some LP.*

Proof: Suppose not, then the smallest timestamped event (which we call E_0) is in the ES of some LP (say LP_0). It could not be processed because LP_0 must process at least one event (say E_1) before it can process E_0. This means that the event E_1 which is to be sequenced before E_0 is not yet in the ES of LP_0, and also $E_1 < E_0$. Let E_2 be the predecessor event of E_1, i.e., E_2 will eventually come as E_1 to LP_0. Clearly $E_2 < E_1$ and hence $E_2 < E_0$ — a contradiction. □

Lemma 2 *The smallest timestamped event in the system, if it is in the OB of an LP, is the* next *event that should leave the LP.*

Proof: Point 6, in Algorithm CON+, says that events from any LP leave in the increasing order of their timestamp. By assumption the smallest timestamped event in the system (say E_0) is in the OB of some LP (say LP_0). Now if LP_0 sends next any other event but E_0, then it would mean that E_0 is not the smallest timestamped event. □

Lemma 3 *The smallest timestamped event in the system, if it is in the OB of an LP, can be moved to its destination.*

Proof: Suppose that E_0 (the smallest event in the system), in the OB of LP_0, cannot be sent to LP_1, its destination. By Lemma 2, since E_0 is next event that LP_0 should send next, the only possibility is that LP_0 does not have any token from LP_1. The above situation together with the algorithmic Point 5 implies that the last event (say E_1) that was sent by LP_0 to LP_1 is either (1) in the corresponding event sequencer queue of LP_1, or, (2) it has been processed. The former case can be outrightly rejected because it implies that E_0 is not the smallest event in the system as $E_1 < E_0$. The second case further implies that E_1, after being processed, did not go to an empty output queue, nor could be compactified with the *last* event in the nonempty output queue in which it was put. Letting E_3 denote the last event of the output queue, we have $E_1 > E_3$. Observe that the customers, say C_1 and C_3, corresponding to E_1 and E_3 pass through the same queue in the PP, and the arrival epoch of C_1 (i.e., E_1) is later in virtual time than the departure epoch of C_3 (i.e., E_3). But this means that E_0 is not the smallest event in the system because $E_0 > E_1$ – which is a contradiction. □

System makes progress if the GVT (*G*lobal *V*irtual *T*ime) makes progress. But GVT is really the time stamp of the smallest timestamped event in the system. If at any time this event gets blocked (forever) then GVT cannot make progress and the system is in a deadlock. Note it can be shown that, due to the irreducibility of routing in the queueing network simulated, the whole system will come to a deadlock if the smallest timestamped event in the system gets blocked. The above Lemmas and observation yield the following Theorem:

Theorem 3 *The conservative simulator, when modified according to our definition, does not deadlock.*

Proof: Suppose that at some wall clock time the smallest timestamped event (say E_0) in the system gets blocked leading to a deadlock. By Lemma 1, E_0 must be in the OB of some LP (say LP_0). By Lemma 3, E_0 can be moved to its destination (say LP_1). But this contradicts our hypothesis that the system is in a state of deadlock. □

Hence a deadlock is not possible and the algorithm can perform in the limited buffer space at each LP.

5 Class of Queueing Networks where the Algorithm is Applicable

For simplicity of exposition we have in this paper considered only single server FCFS queueing networks with state independent Markovian irreducible routing. Note that there is no assumption on the arrival and service distributions. The same ideas hold good if we consider multiple (homegenous or heterogenous) servers per queue simulated. In the multiple server case besides the partititioning of the events in the OB according to the PP queue to which they belong, the events must also be partitioned according to the server (of the queue) to which they belong. Thus our algorithm cannot work for $./ . / \infty$ queues.

Now, if the customers in the above queueing networks also bring with them priorities (the total number of priorities being finite for a queue) and the customers of the same priority are served in FCFS order, our algorithm is still applicable. We assume that the new priority of the customer who finishes service at a queue does not depend upon the state of the system.

As regards state dependent routing, the Theorems 1 and 2 can be worked out for (see also [4]) the following special type: the routing matrix $P(t)$ (where t is the time parameter) is lower bounded by an irreducible stochastic matrix for all t. The algorithm also holds good for this type of state dependent routing.

Finally we wish to add that service disciplines like last-come-first-served, processor sharing, etc., can not be simulated using our algorithm as it is not possible to regenerate the departures after performing compactification.

6 Algorithm for the Optimistic Case

As in the conservative case, we have:

Theorem 4 *In an optimistic distributed simulator of an unstable or stable queueing network with FIFO queues, infinite buffers, and irreducible Markovian routing, the buffers at one of the processors* eventually *get exhausted.*

The proof of Theorem 4 is identical to those of Theorems 1 and 2. When the optimistic simulator has aggressive cancellation the proof relies on the inequality introduced in [4]. But the proof for the case of lazy cancellation uses the additional fact that event copies are maintained in the system as long as the GVT is smaller than the event's timestamp. Thus the simulator will eventually stop because of buffer exhaustion. The algorithm and the proof that it does not deadlock is provided in [5].

7 Concluding Remarks

We have considered the distributed simulation of queueing networks with single server FCFS queues and Markovian routing and with general arrival time distribution and independent and identically distributed service times. Based upon our results in paper [4] and Theorems 1, 2, and 4 presented in this paper, the following observations were made (both for conservative and optimistic simulators):

1. If the queueing model has infinite buffers and is unstable, then the distributed simulation is necessarily unstable and hence the simulator will eventually block because of buffer exhaustion.
2. If the queueing model, with infinite buffers, is stable, then although the simulator is stable, yet with probability one the simulator will block due to buffer exhaustion.

Motivated by these observations, in this paper we have developed enhancements to the original conservative distributed simulation protocol (which we call CON+), that permit the distributed simulation for the above class of queueing networks to run until the space taken by the counters in the simulation exceed the available memory. We also provide proof that CON+ will not deadlock due to the enhancements introduced into the original protocol.

Our results will be useful if the simulation requires more processor memory than what is available. It is interesting to note that a serial (uniprocessor) simulation of an unstable or stable queueing network with N queues requires at most $2N$ pending events in the global time list. Another point to note is that a conservative distributed simulation (with interprocessor flow-control) of an unstable *feedforward* queueing network with Markovian routing can be accomplished in a finite amount of memory at each processor.

The assumption of FCFS queues is crucial. Another assumption is that the events in the simulator are identical in all respects but for their time stamps.

In the case of conservative protocol we assumed that the deadlock avoidance or breaking protocol was suitably altered to accomodate compactification of events. Compactification of events is definitely an overhead and can possibly degrade the performance of the simulator. Compactification of events has been done at the output end of an LP because by doing so we are not losing any sample-path information of the queueing network simulated.

Acknowledgement: We are grateful to some insightful remarks of referees that led us to improve Sections 3.1 and 3.2 in the final version.

References

[1] K. M. Chandy and J. Misra, *Asynchronous distributed simulation via a sequence of parallel computations*, in Commun. ACM 24, 11 (November 1981), 198-205.

[2] R. M. Fujimoto, *Parallel Discrete Event Simulation*, in Communication of the ACM 33, 10 (October 1990), 30-53.

[3] Manish Gupta, Anurag Kumar, Rajeev Shorey., *Queueing Models and Stability of Message Flows in Distributed Simulators of Open Queueing Networks*, in Proceedings of 10^{th} ACM/SCS/IEEE Workshop on Parallel and Distributed Simulation (PADS '96), May 20-23, Philadelphia, USA.

[4] Manish Gupta, Anurag Kumar, *On the Stability of Distributed Simulators of Queueing Networks vis-à-vis the Stability of the Physical Process*, in First International Congress on System Simulation (WCSS'97), September 1-3, 1997, Singapore.

[5] Manish Gupta, *Stochastic Models, Stability, and Performance Analysis of Distributed Simulators of Queueing Networks*, PhD Thesis, Indian Institute of Science, Bangalore, India, Dec. 1998.

[6] D. R. Jefferson, *Virtual Time*, in ACM Tran. of Prog. Lang. and Syst., 7, 3 (July 1985), 404-425.

[7] D. R. Jefferson, *Virtual Time II: Storage Management in Distributed Simulation*, in Proc. Ninth Ann. ACM Symp. Principles of Distributed Computing, pp.75-89, Aug. 1990.

[8] L. Kleinrock, "Queueing Systems, Volume I, Theory", John Wiley & Sons, 1975.

[9] Y. -B. Lin, *Memory Management Algorithms for Optimistic Parallel Simulation*, in Proc. SCS Multiconf. Parallel and Distributed Simulation, vol. 24, no.3, pp. 43-52, Jan 1992.

[10] Y. -B. Lin and B. R. Preiss, *Optimal Memory Management for Time Warp Parallel Simulation*, in ACM Trans. Modeling and Computer Simulation, vol. 1, no.4, pp. 283-307, Oct. 1991.

[11] J. Misra, *Distributed discrete event simulation*, in ACM Computing survey 18, 1 (March 1986), 39-65.

[12] David Nicol and R. Fujimoto, *Parallel Simulation Today*, in Annals of Operations Research, December 1994.

[13] B. R. Preiss and W. M. Loucks, *Memory Management Techniques for Time Warp on a Distributed Memory Machine*, in Proc. Ninth Workshop on Parallel and Distributed Simulation, pp. 30-39, 1995.

[14] B. Samadi, *Distributed Simulation Algorithms and Performance Analysis*, Phd thesis, University of California, Los Angeles, 1985.

[15] Rajeev Shorey, *Modelling and Analysis of Event Message Flows in Distributed Discrete Event Simulators of Queueing Networks*, PhD Thesis, Indian Institute of Science, Bangalore, India, Dec. 1995.

[16] Rajeev Shorey, Anurag Kumar and Kiran M. Rege, *Instability and Performance Limits of Distributed Simulators of Feedforward Queueing Networks*, in ACM TOMACS, Vol. 7, No. 2, pp. 210-238, April 1997.

[17] W. Szpankowski, *Towards Computable Stability Criteria For Some Multidimensional Stochastic Systems*, in Stochastic Analysis Of Computer and Communication System, Hideaki Takagi (ed.), Elsevier Science Publishers B. V. (North-Holland), 1990.

[18] D. B. Wagner, and E. D. Lazowska, "Parallel Simulation of Queueing Networks: Limitations and Potentials", in *Proc. 1989 ACM SIGMETRICS and Performence '89 Conf.*, 1989.