Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

PARALLEL EXECUTION OF A SEQUENTIAL NETWORK SIMULATOR

Kevin G. Jones

Division of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249-0667, U.S.A.

Samir R. Das

Department of Electrical and Computer
Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030, U.S.A.

ABSTRACT

Parallel discrete event simulation (PDES) techniques have not yet made a substantial impact on the network simulation community because of the need to recast the simulation models using a new set of tools. To address this problem, we present a case study in transparently parallelizing a widely used network simulator, called *ns*. The use of this parallel *ns* does not require the modeler to learn any new tools or complex PDES techniques. The paper describes our approach and design choices to build the parallel *ns* and presents preliminary performance results, which are very encouraging.

1 INTRODUCTION

The simulation models for many current generation computer and telecommunication networks are large and complex. As use of networks is proliferating, common examples being the Internet and mobile/wireless networks, the scale of models to be studied also increases and more details are incorporated in the models. Also, in many cases a complete protocol stack needs to be simulated to understand the performance impact of the protocols at various layers. This makes network simulators unbearably slow. Parallel discrete event simulation (PDES) (Fujimoto 1990) technqiues can provide a solution by breaking the large simulation model into submodels and executing them in parallel. However, the synchronization problem in PDES, however well-studied, is a hard problem to solve for a non-expert. Several tools (Bagrodia et al. 1998; Perumalla et al. 1998) have been developed recently targeting the network simulation community. However, they require learning a new set of tools instead of using the more traditional sequential simulators the networking community is familiar with. Recasting legacy simulation code in the new PDES tools is also a big burden. Thus, these PDES tools have not seen widespread use in the networking community in spite of their potential for speedup.

In this paper we address this problem by parallelizing an existing, widely used network simulator, called *ns* (McCanne and Floyd 1997). *ns* is considered to be a *de facto* standard simulator for internetworking protocol research. A large number of legacy simulation models exists using this simulator. The networking community is not inclined to rewrite them in another platform. Thus, a transparent parallel execution of *ns* using established PDES techniques will fill an important gap in simulation research. A parallel version of *ns* is likely to see wide use even if the speedup is less than optimal as such speedup will be obtained without any programming effort on the part of the simulation modeler.

The rest of this paper is organized as follows. In the next section, we introduce basic concepts of parallel simulation. In Section 3, we briefly discuss some existing work on parallel execution of network simulators. In Section 4, we describe the implementation of our parallel version of *ns*. We present the performance benchmarking results in Section 5 and conclusions in Section 6.

2 PARALLEL SIMULATION CONCEPTS

In the description that follows, we assume a general familiarity on the part of the reader about PDES (Fujimoto 1990).

A sequential discrete event simulation stores the events to be executed in timestamp order. The timestamp of an event is the simulated time at which it is to be executed. The main event loop of the simulator is quite simple:

```
while (there are events to execute)
   Retrieve the next event (with
       earliest timestamp) from the
       event set;
   Execute the event, possibly
       scheduling other events to occur
       in future;
```

Parallel discrete event simulations use more than one processor to run a simulation, utilizing either a multiprocessor machine or a network of workstations. The event set is distributed across the processors. If each processor in a parallel simulation simply used the sequential event loop, it is possible that events could be executed out of order, since the "next event" seen by each processor is different, and only one of them is the *real* next event in the global sense. The distribution of the event set makes necessary a synchronization mechanism that guarantees the events will be executed in an appropriate order.

2.1 Synchronization Mechanisms

There are two main types of synchronization mechanisms: conservative and optimistic. Conservative mechanisms, such as the null message protocol (Chandy and Misra 1979; Bryant 1977), preserve proper event execution order by allowing an event to be executed only when it is safe to do so. Safety means the processor knows it will not receive an event from another processor with a timestamp smaller than that of the event in question. A processor with no safe events must block until an event becomes safe to process. Conditions may arise in a naive implementation of a parallel discrete event simulation such that all processors block, resulting in deadlock.

Each pair of communicating processors is connected by a directed link from sender to receiver over which messages can be sent. The terms "message" and "event" are used synonymously, as an event is scheduled on a remote processor by sending a message to that processor. In order for the null messages protocol to work, messages must be sent over the link in non-decreasing timestamp order. It is assumed that messages arrive in the order sent and that no messages are lost in transit, so that messages are received in non-decreasing timestamp order. Messages are stored in a first-in, first-out (FIFO) queue until they can be processed by the receiver. Each link has a `link clock` that holds the timestamp value of the last message sent over that link. The link clock increases monotonically and provides a lower bound on the timestamp of any future message sent over that link.

A conservative protocol, being based on blocking, is prone to deadlocks. Null messages are used to avoid deadlocks. These messages do not correspond to real events in the system being simulated, but provide an updated lower bound on the timestamp of the next message using a `lookahead` value, which is the smallest amount of simulation time that must elapse between an event occurrence in one processor and its effect on another processor. The value of lookahead depends on the particular model being simulated and is usually derived from the model descriptions.

The following pseudocode describes the null message protocol for deadlock avoidance as presented in Fujimoto (2000):

```
while (simulation is not over)
   wait until each FIFO contains at
               least one event message;
   remove smallest time stamped event
               M from its FIFO;
   clock := time stamp of M;
   process M;
   send null message to each neighboring
               process with time stamp
               = clock plus lookahead;
```

Optimistic synchronization prototols, such as Time Warp (Jefferson 1985), aren't concerned with safety of event execution, with the advantage that processors never block, but with the possibility that events are executed out of order. Such out-of-order execution is corrected by rollback when it occurs. The rollback mechanism requires periodic state-saving of the simulator. Time Warp is not generally suitable for parallelizing existing simulators as state-saving of unknown, arbitrary and possibly dynamically allocated data structures is complex. State saving overheads thus can be high to erode performance potentials.

3 RELATED WORK

In the past, a couple of university research projects developed large scale parallel network simulators. In UCLA a `C`-based parallel simulation language, called `Parsec` (Bagrodia et al. 1998) has been developed that supports sequential and multiple parallel simulation protocols (conservative, optimistic and adaptive). A library of network simulators (particularly wireless networks) have been developed in the `GloMoSim` (Global Mobile Information Systems Simulation) (Bagrodia, Zeng and Gerla 1998) project that uses `Parsec`. In Georgia Tech, the Telecommunications Description Language (`TeD`) (Perumalla et al. 1998) has been developed. `TeD` is an object-oriented language for parallel simulation of telecommunications networks. Simulators using `TeD` run on top of the Georgia Tech Time Warp (`GTW`) (Das et al. 1994) for parallel execution. Both `GloMoSim/Parsec` and `TeD/GTW` systems require the user simulation modeler to learn new languages or language extensions to describe their network models. Thus either of these efforts has not been widely used outside the research community that developed them, except in cases where users just wanted to run or modify already developed models in `TeD` or `GloMoSim` rather than building their own. In contrast to these efforts, our work attempts to parallelize a very widely used network simulator transparently to the simulation modeler.

In an effort similar to our own, a distributed version of *ns* has been developed in Georgia Tech to run on a network of workstations (Riley et al. 1999). The main difference in implementation is that they used the Georgia Tech `RTI-Kit` library (Fujimoto and Hoare 1998) for synchronization, whereas we used a variation of the null message protocol for deadlock avoidance as described before. `RTI-Kit` is a set of libraries supporting implementation of the Run-Time Infrastructure component of the U.S. Department of Defense's High-Level Architecture (HLA) (Dahmann et al. 1997) for distributed simulation systems. As for performance, much of the improvement came from parallelization of the setup of the simulation, not in the actual execution. Each processor has less work to do for the setup since each is only simulating a part of the entire network. Our results, on the other hand, involve only actual simulation execution since we ignore the setup times in our analysis.

4 IMPLEMENTATION OF PARALLEL NS

ns, which stands for network simulator, is widely used in the network simulation community. It is an object-oriented discrete event simulator used for network research. It is written in `C++`, and `OTcl` is used as a command interface and for describing the network models. The user can define network nodes, physical connections (links) between nodes, and logical connections between agents for traffic in the network. Parallelizing *ns* allows a federation of separate *ns* processes to execute a simulation in parallel. In this section, we describe the steps undertaken in parallelization.

4.1 Parallelizing the *ns* Main Event Loop

The main event loop in sequential *ns* consists simply of retrieving the next event from the event set and executing it. This is repeated as long as the simulation is not finished. We augmented this loop to implement the null message protocol for deadlock avoidance discussed previously.

Each iteration of the parallel version of the *ns* main loop consists of: (i) removing the next event from the event set; (ii) sending necessary null messages to neighbors; (iii) receiving a message on each incoming link whose link clock is less than the timestamp of the next event, either scheduling it on the event set or saving it as the new next event depending on its timestamp; (iv) setting the clock; and (v) processing the next event. Following is the pseudocode of our parallel *ns* main event loop.

```
while (simulation is not over)
  //deque
  remove smallest timestamped event M
  from event set;

  //send null messages
  for each neighboring process j
  (which receives messages from
  this process)
     nullmsg_time := lower bound on
     timestamp of all future messages
     sent to j;
     if nullmsg_time > last_nullmsg_time
     from i to j
        send null message to j with
        timestamp = nullmsg_time;
        last_nullmsg_time from i to j :=
        nullmsg_time;

  //get smallest timestamped message
  for each incoming link i (from a
  neighboring sender)
     if timestamp of M > link clock of i
     // link clock is the most recent
     // timestamped message passing
     // through a link
        wait for next message, N, to
        arrive on the link;
        set link clock of i to timestamp
        of N;
        // swap events if new minimum is
        // received
        if timestamp of M <= timestamp
           of N schedule event N on
           event set;
        else
           reschedule event M on event
            set;
           //(since an earlier event, N,
           // was received)
           M := N;
  clock := time stamp of M;
  process M;
```

In an *ns*-based simulator all traffic connections may not become active at the beginning of the simulation (time 0). They may start at different times. This presents a problem as a series of null messages need to be transmitted to bring the link clock to the start time of the corresponding traffic connection. This problem is explained below with the help of an example (see Figure 1).

Initially, all link clock values are 0.0, indicating that no messages have yet been sent between processes. Suppose, node A is to start sending messages to node B at time 1.0, and node B should start sending to A at 2.0. Assume there is no other traffic in the network. Processor 0 has a local event with timestamp equal to 1.0, corresponding to the start of node A's communication with node B. Likewise, processor 1 has an event with timestamp equal to 2.0. Since each link

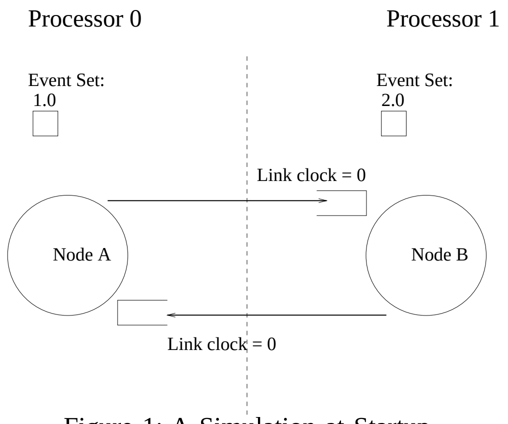

Figure 1: A Simulation at Startup

clock has the value 0.0, neither processor can proceed to process its first event, since neither knows it will not receive a message between 0.0 and the timestamp of its first event (1.0 for processor 0 and 2.0 for processor 1), so the system is deadlocked unless we start with sending null messages.

The question is what should be a good value of the timestamp of the initial null message. It could be just time 0 plus the value of the lookahead. Lookahead in parallel *ns* is the fastest amount of time a message can take passing from one processor to the next, and is a function of packet sizes, link bandwidths, and link delays. Specifically, the lookahead from processor i to j is the minimum transmission time plus link delay over all links from i to j. Notice that the lookahead value may be small (e.g., typically tens of milliseconds or less) compared to possible communication start times (for example, communication may start after a few hours of simulated time). This would mean a large number of null messages are required to bump up the simulation clock to the communication start time.

We tried to cut down on the number of null message transmissions as much as possible in order to make parallel execution efficient. The first null message value sent from processor 0 to processor 1 in Figure 1 is equal to the starting time of communication plus lookahead, in order to avoid unnecessary null message exchanges. With the same goal, other optimizations, such as avoiding sending of duplicate null messages, are also used. Duplicate null messages may be generated when an iteration of the main loop generates a null message with a timestamp the same as that in a previous iteration.

The timestamp of a null message is equal to the lower bound on the timestamp of any future message to be sent from the sender to the receiving processor. In order to compute this lower bound on messages from processor i to processor j, processor i must compute the minimum timestamp of the next packet that will be sent over all logical connections that flow from processor i to processor j. The source of the packets may reside on processor i or it may be on a different processor further upstream. Likewise, the final destination of the packets may be on processor j or further downstream. In any case, processor i keeps track of each logical connection spanning the processor boundary between processors i and j, for each downstream neighbor j, so that it can compute the minimum timestamp of the next packet. Assuming static routing, processor i can compute when it will send a packet out on its next hop to processor j as soon as it receives the packet, whether it receives the packet from an agent residing on the same processor, or from an upstream processor, simply by summing the transmission and delay times for each link in the path the packet will travel inside processor i, along with the final transmission time and delay of the i-to-j hop.

4.2 Communication Across Processors

Each *ns* node has an object called an *address classifier* that looks at each incoming packet to determine if it is destined to the node or if it should be forwarded downstream to another node. When a classifier receives a packet, it checks to see if the node that owns the classifier resides on the current processor. If so, execution proceeds as normal, with the classifier delivering the packet downstream to the next node or to a port classifier, depending on the destination of the packet. If, however, the node does not reside on the current processor, the classifier knows the packet has just traversed a cross-processor link. It acts as a proxy by packaging the packet inside an MPI (Message Passing Interface) message and sending it, via a call to `MPI_Send` (a communication primitive in MPI), to the corresponding classifier on the destination processor. Execution then proceeds as usual.

We needed to make some extensions to the *ns* input syntax to facilitate parallel execution. We tried to make the changes as transparent as possible so that someone who is already familiar with *ns* could learn to use our parallel version with minimal effort. The syntax changes are related to partitioning and mapping of the network for parallel simulation, which must be specified manually by the user.

4.3 Limitations

We currently support parallel execution of *ns* programs consisting of point-to-point links with static routing and UDP traffic. We hope to extend the functionality to include support for TCP connections, dynamic routing, and shared medium networks. Another major drawback is that the whole network still resides on each processor, even though each processor only simulates the activities of its own part of the network. This was done initially to facilitate message passing between processors by having corresponding *classifier* objects be identified with the same number (they are assigned identifiers as they are created by *ns*). Since each processor has the definition for the whole network and the network objects are created in the same order, each object has the same identifier across all processors. We will be able to save memory, thereby allowing us to simulate even larger networks, once we are able to specify partitions of the network to the corresponding processors.

5 PERFORMANCE ANALYSIS

Here we describe the benchmark model used for our experiments and an analysis of the performance results.

5.1 Benchmark Model

It is important to use fairly large, realistic networks as a benchmark for our case study. Since it is somewhat tedious to manually generate large networks in *ns*, we chose to use the *Georgia Tech Internetwork Topology Models* (`gt-itm`) topology generator to generate our benchmark network (Zegura 1996). One type of network `gt-itm` generates is called a transit-stub network, which consists of a collection of inter-connected transit domains. Each node in a transit domain is connected to a sub-network called a stub domain, consisting of some number of nodes.

The output of `gt-itm` is in Stanford GraphBase (SGB) format (Knuth 1994). We used a tool called `sgb2ns` available with *ns*, which translates graphs from SGB format to a `Tcl` file that is readable by *ns*. We further modified this program to generate separate `Tcl` files for each processor for the parallel executions, and the corresponding file for the sequential execution needed as a baseline for speedup computations. Our modification also generates a user-specified number of local and cross-processor connections for the traffic.

For the performance results that follow, a transit-stub network is used with eight transit domains, four nodes per transit domain, and five nodes per stub domain. This makes a total of 192 nodes as shown in Figure 2.

It is natural to partition a transit-stub network by the transit domains. We ran experiments using four processors, each simulating two transit domains of our benchmark model. Local traffic, traffic in which packets stay entirely within the sub-network simulated by a single processor, is obtained by defining sources and sinks within each stub domain. Cross-processor traffic flows from a source stub node in one transit domain to a destination stub node in another transit domain modeled by a different processor, and may travel through any number of intermediate processors. Some effort was made to balance the load evenly across processors by balancing the number of hops taken in each logical connection. Each local connection consists of four hops. Each cross-processor connection contains three local hops on the source processor, one cross-processor hop to the neighboring destination processor, and three local hops on the destination processor to the final destination node.

A Sun multiprocessor with four UltraSPARC-II processors and 1.5 GB of main memory was used for the experimental study. The MPICH implementation (Gropp et al. 1996) of the Message Passing Interface (MPI) was used for communication between processors. Because of a general-purpose MPI-based implementation, the parallel *ns* can be run on any platform supporting MPI. In particular it can execute on a network of workstations without any modification by simply linking with platform-specific libraries. However, here we present results only for a multiprocessor because of a much lower message passing overhead. As an example, sending an MPI message between processors on the Sun multiprocessor takes around 2 microseconds, where the same message takes around 40 microseconds when used on a network of workstations connected by an ATM switch.

5.2 Experimental Results

The speedup results shown in Table 1 come from dividing the execution time of the unmodified, sequential version of *ns* by the execution time of our parallel version of *ns* simulating the transit-stub network model as described a previous subsection. Setup time is ignored; the times presented in the table are from actual event execution.

Most of the simulations were run to an endtime of 900 seconds, generating between 15 and 70 million events, but some of the smaller simulations (with 0 or 2 local connections) were run up to ten times longer to be sure that the simulations stabilized. As expected, speedup increases as the number of local connections increases, since the processors are more heavily loaded, thus finding more work to do within the lookahead interval and between message communications. The fraction of time spent blocking, waiting for messages to arrive on incoming queues, decreases as the local load increases. We found that this fraction decreases from above 60% for 0 local connections to about 20% for 8 local connections. Speedup is also generally higher for larger lookahead for the same reason: a processor finds more safe events to process in a larger time interval than a smaller one, all else being equal.

6 CONCLUSIONS

We presented a case study in parallelizing an existing, widely used network simulator *ns* using well-known PDES techniques. A conservative synchronization technique is adapted to suit the *ns* platform. MPI is used for inter-processor message communication. Existing network models can be run in parallel with minor modifications to the input script. This is in contrast with many existing PDES systems, where the users have to reprogram their model on an entirely new platform.

Experimental results demonstrate a speedup of up to 2.83 on 4 processors. As expected, the speedup is depen-

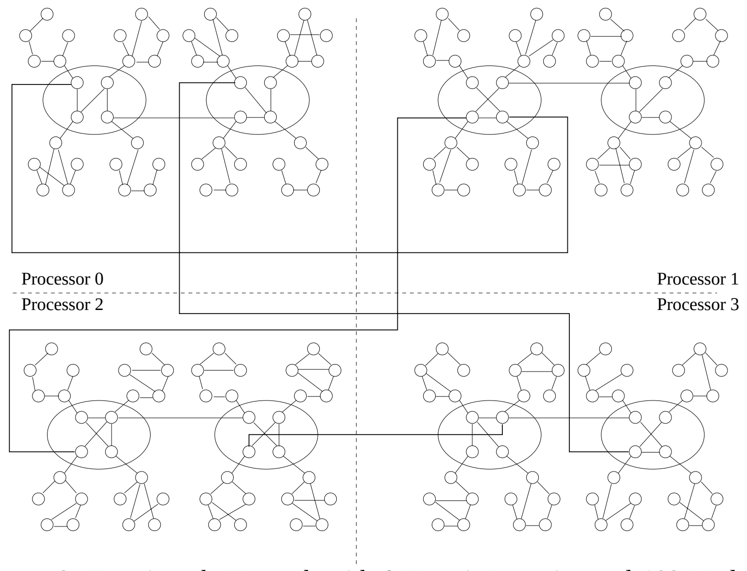

Figure 2: Transit-stub Network with 8 Transit Domains and 192 Nodes

Table 1: Execution Times and Speedups for Various Lookahead Values and Numbers of Local Connections on a 4-processor Sun UltraSparc

Lookahead (ms)	No. of local connections per processor	Sequential run time (sec.)	Parallel run time (sec.)	Speedup
3.75	0	246 (2.5×)	268 (2.5×)	0.918
	2	507 (2.5×)	379 (2.5×)	1.34
	4	305	159	1.92
	6	410	196	2.09
	8	529	236	2.24
7.50	0	243 (5×)	287 (5×)	0.847
	2	760 (5×)	475 (5×)	1.60
	4	258	114	2.26
	6	364	154	2.36
	8	463	188	2.46
17.86	0	203 (10×)	237 (10×)	0.857
	2	1199 (10×)	546 (10×)	2.20
	4	234	84.7	2.76
	6	331	120	2.76
	8	438	155	2.83

dent on the relative numbers of local and cross-processor connections, and the value of the lookahead. We find the speedup numbers to be reasonable considering the fact the parallelization effort on the part of the user is minimal.

Currently, the work is at a proof-of-concept and proof-of-performance stage. Only a restricted class of networks can be simulated. Our ongoing work includes extending the design to include TCP agents, dynamic routing, shared medium networks, and actually partitioning the network model so that each processor keeps track only of its part of the network. We can expect even better speedup this way, as the *ns* setup computation would then be parallelized, since each processor works only on its partition.

ACKNOWLEDGMENTS

This work is partially supported by a NSF Career Award no. ACI-9733836 and DoD/AFOSR grant no. F49260-96-1-0472.

REFERENCES

Bagrodia, R., R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song 1998. Parsec: A parallel simulation environment for complex systems. *IEEE Computer* 31(10): 77–85.

Bagrodia, R., X. Zeng and M. Gerla 1998. GloMoSim: A library for the parallel simulation of large wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98)*, 154–161.

Bryant, R. E. 1977. Simulation of packet communication architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology.

Chandy, K. M. and J. Misra 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* SE-5(5): 440–452.

Dahmann, J. S., R. M. Fujimoto and R. M. Weatherly 1997. The Department of Defense high level architecture. In *Proceedings of the 1997 Winter Simulation Conference*, 142–149.

Das, S. R., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette 1994. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, 1332–1339.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33(10), 30–53.

Fujimoto, R. M. 2000. *Parallel and distributed simulation systems*. Wiley-Interscience.

Fujimoto, R. M. and P. Hoare 1998. HLA RTI performance in high speed LAN environments. In *Proceedings of the Fall Simulation Interoperability Workshop*.

Gropp, W., E. Lusk, N. Doss, and A. Skjellum 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6): 789–828.

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

Knuth, D. 1994. *The Stanford GraphBase: A platform for combinatorial computing*. Addison-Wesley.

McCanne, S. and S. Floyd 1997. The UCB/LBNL/VINT network simulator. `<www-mash.cs.berkeley.edu/ns/>`. Lawrence Berkeley Laboratory.

Perumalla, K., R. Fujimoto, and A. Ogielski 1998. TeD - A language for modeling telecommunications networks. *Performance Evaluation Review* 25(4): 0–0.

Riley, G. F., R. Fujimoto, and M. H. Ammar 1999. A generic framework for parallelization of network simulations. In *Proceedings of the 7th International Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, 128–135.

Zegura, E. 1996. Modeling topology of Internetworks. Web URL `<www.cc.gatech.edu/fac/ellen.zegura/graphs.html>`.

AUTHOR BIOGRAPHIES

KEVIN G. JONES is a Ph.D. student in the Division of Computer Science at The University of Texas at San Antonio. He received his B.S. degree from The University of Texas at San Antonio in 1993 and M.S. degree from The University of Texas at Austin in 1995, both in computer science. His research interests are in parallel discrete event simulation and distributed virtual environments.

SAMIR R. DAS is an associate professor in the Department of Electrical & Computer Engineering and Computer Science at University of Cincinnati. He received his B.E. degree in electronics and telecommunication engineering from Jadavpur University, Calcutta, in 1986; M.E. degree in computer science and engineering from the Indian Institute of Science, Bangalore, in 1988; and Ph.D. degree in computer science from the Georgia Institute of Technology, Atlanta, in 1994. From 1994 to 1999 he was on the faculty in the University of Texas at San Antonio. His current research interests include parallel simulation, mobile/wireless networking and performance evaluation. He published over 30 technical papers in these areas. Dr. Das served on the program committees of several recent annual ACM/IEEE/SCS workshops on Parallel and Distributed Simulation (PADS). He also served in the program or organizing committees of several recent networking and performance evaluation conferences such as MobiCom, MobiHOC, IC3N and MASCOTS. he received the U.S. National Science Foundation's Faculty Early CAREER award in 1998. He is a member of the IEEE, IEEE Computer Society and ACM. His email and web addresses are `<samir.das@uc.edu>` and `<www.ececs.uc.edu/~sdas>`.