A Generic Framework for Parallelization of Network Simulations

George F. Riley Richard M. Fujimoto Mostafa H. Ammar

College of Computing
Georgia Institute of Technology Atlanta, GA 30332
{riley,fujimoto,ammar}@cc.gatech.edu

Abstract

Discrete event simulation is widely used within the networking community for purposes such as demonstrating the validity of network protocols and architectures. Depending on the level of detail modeled within the simulation, the running time and memory requirements can be excessive. The goal of our research is to develop and demonstrate a practical, scalable approach to parallel and distributed simulation that will enable widespread reuse of sequential network simulation models and software. We focus on an approach to parallelization where an existing network simulator is used to build models of subnetworks that are composed to create simulations of larger networks. Changes to the original simulator are minimized, enabling the parallel simulator to easily track enhancements to the sequential version. In this paper we describe our lessons learned in applying this approach to the publicly available ns *[9] software package, and converting it to run in a parallel fashion on a network of workstations. This activity highlights a number of important problems, from the standpoint of how to parallelize an existing serial simulation model and achieving acceptable parallel performance.*

1 Introduction

Simulation will remain the method of choice for many network analysis problems. Although analytic models are useful in many situations, the complexity of modern networks combined with the inability to apply simplifying assumptions in many analysis problems (it is well-known that Markovian traffic assumptions are often inappropriate and can lead to misleading results) limit the applicability of purely analytic methods. Even when analytic methods can be used, simulation is often used to validate the models.

However, simulation tools have not been able to keep up with the rapid increases in the size, complexity, and speed of modern networks. Today, a packet level simulation of a gigabit network containing a few hundred nodes can be expected to require hours, perhaps even days, of CPU time to simulate only a few minutes of network operation using a contemporary workstation. Next generation Internet (NGI) simulations will require models containing millions to billions of network nodes for scalability studies. Other simulations require long periods of network operation to be simulated to capture statistics concerning infrequent (rare) events such as cell loss probabilities in ATM networks. Advances in CPU speed alone can only be expected to provide perhaps an order of magnitude speedup over the next few years. It is clear that a viable approach exploiting scalable, parallel network simulation techniques is needed to achieve the performance needed to simulate these networks.

The parallelization of network simulations has been studied for some time. Nicol et. al. discuss *IDES*, a Java based parallel simulation in [11]. Ogielski et. al. present the Scalable Simulation Framework (*SSF*) in [5]. Perumalla et. al. describe *TED*, a process oriented simulation system in [14] and [13]. Bagrodia discusses the *Maisie* simulation environment in [2], and Unger describes *Telesim* in [16].

Numerous demonstrations have reported order of magnitude speed ups of simulation computations. However, overall the impact of parallel simulation technology on the networking research community has been minimal; sequential simulation remains the method of choice by nearly all network modelers today. One problem that has slowed the adoption of parallel simulation techniques is the need to adopt new, unfamiliar simulation languages and tools to exploit this technology. Models for parallel simulation engines are not as mature as those for sequential simulators, and generally have not gone through as rigorous validation efforts as the best sequential models. Even as parallel simulation tools mature, sequential simulation tools and models will also advance.

Our work focuses on a federated approach to parallel simulation where extensions to existing sequential (or parallel) simulation engines (specifically *ns*) are added which will allow them to be interconnected to create parallel simulators. Changes to the original simulator are minimized so that the parallel simulation tool can easily track new enhancements and additions to the original simulators. Each simulator will be given network topology and data flow characteristics which describe only a portion of the network being simulated.

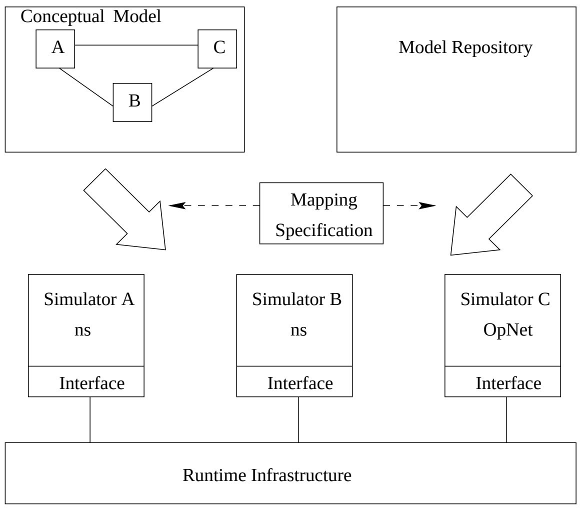

Figure 1. Conceptual Overview

Interaction between simulators is determined at run time using techniques which we will describe. One advantage of this approach is that the memory of many processors can be used in the simulation, thus allowing larger network models to be simulated. Additionally, models which have already been developed can be reused. Further, because the parallel simulator is based on existing sequential simulators, many existing tools can be readily reused and modelers are presented with a relatively familiar modeling environment.

This approach is not unlike that described by Nicol and Heidelberger [10]. That work focused on parallelization of queueing network simulations written in a package called CSIM. However, as will soon be evident, more sophisticated techniques are required to compose network simulations because global information is often needed. Perhaps closer in spirit to the work described here are efforts such as the Department of Defense High Level Architecture [7], however, that work has been largely focused on military applications (e.g., virtual environments for training and war game simulations). Some work in applying the HLA concept to network simulations is in progress [1].

Longer term, we envision a composable simulation infrastructure where network simulations are routinely constructed by extracting model components developed by others (possibly using different simulation packages) from other individual's web sites, configured to create a large network simulation, and executed on parallel and/or distributed computing platforms. Model execution might take place locally on the user's own computing facilities, or remotely on high performance compute servers. In situations where it is difficult to co-locate all of the sub-models due to porting costs or the potential for release of proprietary information, geographically distributed execution across the Internet may be employed. Figure 1 shows a conceptual overview of our vision of how this can be done.

In this paper, we present an outline of the issues that must be addressed by the designer of a composable, distributed network simulation package and give some proposed solutions. We give our experiences with applying these solutions to the popular network simulator *ns*. The remainder of this paper is organized as follows. In section 2 we give a general methodology which we used to construct a parallel simulation from the original serial version. In section 3 we discuss the issues that are faced by the network modeler when taking an existing serial simulation of a subnetwork and composing it with models of other subnetworks so that they can be run in parallel. In section 4 we discuss the issues that affect the simulator itself which are required to coordinate the events between the sub-models and obtain correct results. In section 5 we discuss several of the performance related problems we encountered and discuss our solutions. In section 6 we give some performance comparison results using our parallel *ns* implementation. In section 7 we give some conclusions from our research.

2 A Methodology for Parallelization

In this section, we formulate a general methodology that can be applied to create a parallel version of a serial simulation. We assume that the parallel simulation is to be run on a shared memory, symmetric multiprocessor (SMP). An additional enhancement is subsequently given to modify the methodology to create a distributed simulation on a loosely coupled network of workstations without shared memory. The basic steps are as follows:

1. Determine how many physical processes (or threads) will be assigned to run the parallel simulation.
2. Create a one-to-one mapping of the state variables in the serial simulation to the physical processes of the the parallel simulation.
3. Maintain a separate event list for each physical process.
4. Distribute events during the execution among the physical processes.
5. Add mechanisms to insure the state managed by the different physical processes remains consistent.
6. Perform any optimizations that may be needed to increase performance.

Determine number of physical processes. Ideally, on a system that has n CPU's, the serial simulation could be split into n separate physical processes (or threads), which each handling $1/n^{th}$ of the workload. Depending on how the physical processes are defined, the entire simulation state may have to be replicated into each physical process.

Create a mapping. Determine some mapping of state variables to physical processes. In other words, decide which of the physical processes will be responsible for maintaining which subset of the state space. In the simplest case, one can just divide the entire state space into n partitions (where n is the number of physical processes), and map the entities by the partition number.

Maintain separate event lists. Each physical process will only be concerned with events that affect the state variables mapped to that physical process. Thus events which affect other state variables can simply be ignored. Some form of synchronization protocol is needed to determine when events are safe to process.

Distribute events. During the execution of the simulation on each of the physical processes, any event generated must also be mapped to a (potentially different) physical process for processing. This can be done by examining the mapping of the state variable that is the target for the generated event. If it is a non-local event (mapped to a different physical process), then that event must be made known to the correct physical process, by some event transfer mechanism.

Insure consistent state. During the processing of each event one must insure that for any replicated state variable that the state is consistent among the physical processes. This can be done by simply insuring that no physical process will generate events for another physical process in the simulated past, and by forwarding state change information between physical processes using some communications mechanism between the physical processes. A synchronization protocol is needed for this task.

Perform optimizations. Once the previous steps have been done, one can then begin to make optimizations to enhance performance. For example, it may sometimes be determined that the complete simulation model need *not* be replicated onto all physical processes. In the case of *ns* (for example) we were able to save considerable memory by only creating simulated entities on each physical process that were mapped to that physical process. However, this optimization did cause other difficulties as described in section 3. Other optimizations might be insuring that the mapping of state space to physical processes gives sufficiently large *lookahead* to allow for efficient parallel event processing, and to efficiently determine which events are safe to process.

The above methodology will allow any serial simulation to be modified to run in parallel on a tightly coupled SMP system. The addition of the a new *step 0* given below will allow the serial simulation to be run in parallel on a loosely coupled network of workstations.

Step 0, Replicate the entire simulation model. Create a separate copy of the entire serial simulation model on each of the distributed systems in the network of workstations. By doing this, each physical process will have any global state knowledge necessary to complete the simulation, but will only be responsible for maintaining and modifying state variables that are mapped to it. An obvious optimization is reducing the state space maintained by each physical process to perhaps only include state variables mapped to that physical process. This optimization can cause difficulties however, as outlined in the next section.

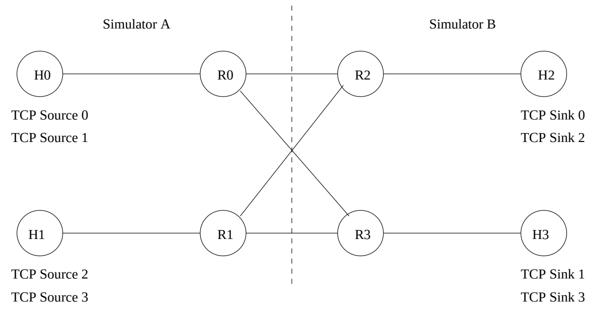

Figure 2. Simple Network Model

3 Describing the Simulation

In this section we discuss some of the issues which must be addressed when describing a network simulation to be run in a distributed fashion on a network of workstations. These issues are a direct result of the memory optimization discussed previously which allowed us to *not* replicate the entire network model on each physical process.

Consider the simple network shown in figure 2 consisting of four end hosts (H0 – H3) and four network routers (R0 – R4), connected with physical communication links as shown by the solid lines. This simple network model also has four logical dataflow connections consisting of four "TCP Source" to "TCP Sink" pairs, as shown in the figure. Further consider that the network modeler has decided to run the complete model in parallel on two systems, simulator A and B, splitting the model into sub-models as shown by the dashed line. From the point of view of the modeler, there are two basic problems which must be addressed and solved, specifically:

1. Defining physical connectivity
2. Defining logical connectivity

3.1 Defining Physical Connectivity

If the network in figure 2 were being simulated on a single workstation using a serial simulator the description of the overall network topology is quite straightforward. Using the semantics of the network simulator being used, the modeler would simply define the simulation elements representing each of the eight nodes, and then define the physical links by referencing those nodes as the endpoints of the links. The ability to define a network topology of nodes and interconnecting links is a fundamental requirement of any network simulator.

But now consider what problems face a modeler when he decides to split the model into submodels A and B (as shown in the figure) and run those in parallel, by composing two network simulators A and B. As mentioned previously, we assume that only those nodes that will be managed by simulator A will be defined on that simulator, in order to keep the memory requirements for each system as small as possible. Thus it is obvious that it becomes problematic to define a simple physical link, such as the link from $R0$ to $R2$ in figure 2. By our assumption above, simulator A will not have a node entity for node $R2$ and would not be able to define that endpoint for the physical link. A similar problem would exist in sub-model B trying to describe the physical link from R2 to R0 and the link from R2 to R1. The basic problem is that a network simulation package designed to be run in serial on a single system will assume the existence of a variable or object representing each and every network element being modeled, and assume those objects can be referenced by name. When decomposing a model in to distinct sub-models, only those network elements which are within the same sub-model can refer to each other by variable name.

Our solution to this problem is to borrow some well known abstractions from the networking community, namely that of an *IP Address*, and a *Network Mask*. We propose that the syntax of the topology specification in a network simulator be extended to allow the specification of these values for any link endpoint. In the case where a link connects to a node declared in a different sub-model, then only the local endpoint of the link need be specified. We refer to this as a *Remote Link*, or *rlink*. At runtime, the physical connectivity of the *rlinks* can be determined by matching the network portion of the *IP Address*.

3.2 Defining Logical Connectivity

In our model of figure 2, we also are modeling four logical dataflow connections consisting of pairs of sources and sinks. A network simulator might allow for the declaration of an entity which generates data (such as *TCPSource0* in our example), the declaration of a receiving end of the data flow (such as *TCPSink0* in our example), and some way to specify that the two ends have logical connectivity.

Again assuming that we want to decompose the model into sub-models as in the previous section, we are faced with a similar problem. We again assume that a submodel will only define and manage data flow endpoints for those nodes that are managed by the local simulator. Simulator A will be unable to identify the remote endpoint *TCPSink0* of data source *TCPSource0* since *TCPSink0* is defined on a different simulator.

To solve this problem, we again borrow well known abstractions from the networking community, using an *IP Address* and a *Port number*. We propose allowing the specification of an *IP Address* for a physical link endpoint (as previously mentioned), and also allowing the specification of a binding of a logical connection endpoint to a 16 bit port number unique within a node. Then a logical data connection can be specified by giving the *IP Address* and *port number* of the remote endpoint.

4 Implementing the Simulator

We now turn our attention away from issues of concern to the network model creator, to those issues which must be addressed by the simulator itself in order to execute on a distributed platform. Of course, of primary importance is that a distributed simulation run on a set of simulators in parallel must produce exactly the same results as a serial, single system simulator running the same model. In order to properly coordinate the processing of network simulation events between simulators, there must be support for inter-system communication in three basic areas. Those are:

1. Determining Routing Paths
2. Event Time Management
3. Event Communication

4.1 Determining Routing Paths

The problem of how to determine correct routing paths between sub-models in less obvious than those of the previous sections. Again referring to the model of figure 2, consider what the simulator must do when simulating the arrival of a packet at router R0, with the ultimate destination of end host H2. In a single system simulation the correct route (which link to forward simulated packets on for each possible destination) can be computed a priori using global knowledge of the topology. In our example, router R0 would have predetermined that any packet being routed to end host H2 must be forwarded on the link connected to R2. With global knowledge of the network topology, this is a simple and straightforward computation (although it can be time consuming for large topologies).

Once we decide to decompose the model into sub-models A and B as in the previous sections, we are faced with a problem. Without global network topology information, simulator A does not have sufficient information to make a routing decision at routers R0 and R1. Router R0 has two remote links in our example, only one of which is the correct link to forward packets addressed to end host H2, but it has insufficient information to determine which one is correct.

We propose three possible solutions for this problem. The first and simplest is to put the burden on the modeler to define the appropriate routes. The model topology specification can be enhanced to specify a list of remote *IP Addresses* for each remote link. Anytime a packet is to be forwarded on a simulated node with multiple remote links, the correct route can be determined from the list of specified routes. This solution works as long as the appropriate route will not change over the course of the simulation. Routing changes could occur due to the simulation of link failures or node failures in the model.

A second and more desirable solution is to start with a single model which specifies the entire network topology. With global network knowledge, the correct routes between nodes (even routes between simulators), can be determined automatically by the simulator without effort on the part of the modeler. However, this method requires the entire topology to be defined and processed on each simulator, which creates a duplication of memory use. We propose the use of a topology preprocessor, which will be given a complete picture of the simulated network and will compute the needed routing information. The preprocessor will then create the submodels (based on either a mapping given by the modeler or by some other optimization), and pass the submodels to the simulators (with the routing information included). This solution still lacks the ability to adapt in simulation time to changing network topologies.

A third solution is the have the simulator run some existing and well known routing protocols while the simulation is running. Simulated network nodes will start with routing tables generated a priori (with one of the two previous approaches), and will exchange dynamic routing information in the simulation (using for example the *Border Gateway Protocol* (*BGP*[15]) to adapt to any changes in simulated network topology.

4.2 Event Time Management

A single system discrete event simulation typically consists of the maintenance of a pending event queue, which is sorted by increasing simulation time. A simulator simply retrieves the next event from the event queue (which will be the earliest pending event by virtue of the sorting), and processes that event. The processing of an event causes zero or more new future events to be generated, each of which will placed in the pending event queue in the proper sorted order. When the simulation is run on a single system this process is quite straightforward. However, when attempting to run a parallel simulation, a single simulator cannot just retrieve the most recent event from it's local event queue and process it.

To see why this is true, consider a simple distributed simulation such as shown in figure 3. This model consists of two sub-models, A and B, each of which has a single node ($N0$ and $N1$ respectively) which are connected by a physical link $Link0$. Node $N0$ has a TCP source which has a logical connection to a TCP sink on node $N1$. Suppose initially that simulator A has a single event in its local event queue, specifically the generation of a data packet at simulation time 0.1 which is to be forwarded to node $N1$. When A processes this event, a number of new future events are generated, the only one of importance for this discussion is a *Retransmit Timer Expired* (*RTE*) event, which may have a simulation time tag of 6.1 for example. If A just blindly continues to process events from its local event queue, the *RTE* event will be processed immediately, and a duplicate packet retransmission will be generated, and another *RTE* event will be generated at time 12.1. This process could repeat indefinitely, and of course is not the correct behavior for this model.

The correct sequence of events is for simulator B to

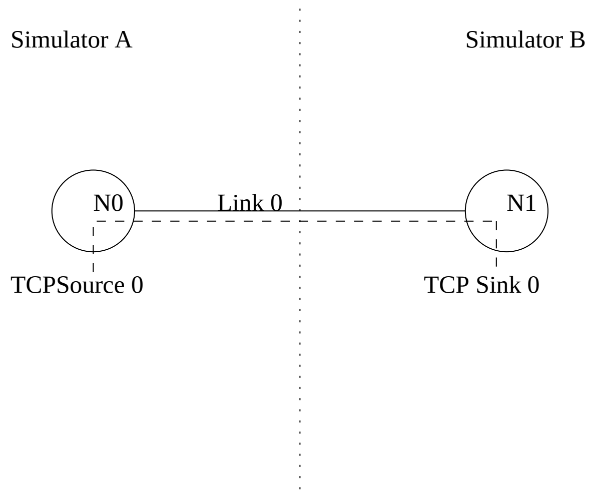

Figure 3. Simple Distributed Simulation

process the receipt of the data packet at the TCP sink on node $N1$, and generate a reply packet acknowledging the receipt of the data packet. This *Ack* packet would in turn would be received by the TCP source on node $N0$ at some time prior to time 6.1 and would cancel the *RTE* event before it is processed. To state the correct sequence of events more generally, no simulator can process an event until it can be proven that no earlier event can be received from any other simulator. In our specific case, simulator A should not process the *RTE* event at simulation time 6.1 until it can be assured that simulator B will not send events with a simulation timestamp less than 6.1.

This is a common and well known problem within the Parallel and Distributed Simulation (PADS) community. In order to insure that the above problem does not occur, all simulators participating in the distributed simulation must agree on the *lower bound time-stamp* (*LBTS*). Essentially each simulator must determine that no other simulator can create events at an earlier time before it can be allowed to process its most recent event. A number of protocols exist which allow a group of simulators to agree on the *LBTS*[3, 4, 12]. Using a good time management runtime library, such as that provided by the Georgia Tech *RTIKIT*, the main event processing loop of a network simulator can easily be modified to handle events in proper timestamp order.

Using *RTIKIT* each simulator will call the *Next Event Request* (*NER*) service of the *RTIKIT* prior to processing any pending event, specifying the timestamp of the pending event. The *RTIKIT* will determine whether it is safe to process the pending event at the specified time, and provide a *Time Advance Grant* (*TAG*) indicating what simulation time can safely be advanced to. Ideally, the *RTIKIT* can determine the *TAG* value using locally available information, or information previously obtained from the other simulators. In many cases however, all other simulators must be polled to determine the *LBTS* value.

4.3 Event Communication

Once a given simulator has determined that it can safely advance the simulation time to time T, it can then process any events with timestamp less than or equal to T, which in turn may generate other events. In some cases, the new events generated may be events that affect other simulators. Again using our example, B might process an event at simulation time 0.20 which may cause an acknowledgment packet to be sent to A to arrive at simulation time 0.021 (allowing for the link delay). Clearly, there must be some mechanism for B to inform A of the new event, and all of the information associated with the event such as the event time, sequence number being acknowledged, connection identifier, etc.

As with time management, a runtime library such as *RTIKIT* should provide data distribution management functionality. The *RTIKIT* provides these services using a multicast group management strategy known as *MCAST*. Essentially the *MCAST* services allow for the creation of *groups*, the sending of data by a simulator to a particular group, and the delivery of that data to zero or more simulators who have *subscribed* to the group. Each simulator should join groups for any remote link defined (as described in section 3), using the network address as the group identifier, and should send event messages to other simulators by sending the information to the same group. Using such a scheme, simulators can generate event messages for other simulators without necessarily knowing the identity of which other simulators receive the messages.

5 Performance Related Issues

In this section, we discuss two areas where the performance of a distributed simulation can be seriously degraded if not done properly. During the course of our development and testing of the distributed *ns* code, we found that significant CPU time, network overhead, and wall clock time can be wasted performing the *LBTS* computations and while polling network sockets for potential data. Our findings and improvements are discussed in detail in the following subsections.

5.1 Eliminating Excessive *LBTS* Overhead

As mentioned in the previous section, each of the simulators working in parallel needs some assurance that it is safe to advance its local simulation time from its current value T to some new value $T + \Delta T$. From the perspective of simulator A, this time advance is safe if no other simulator can possibly generate a new event affecting simulator A with a timestamp less than $T + \Delta T$. The only way simulator A can get this assurance is to gain knowledge of the the smallest simulation time of all other simulators, adjusted for the minimum time delay of events between simulators (known as the *lookahead*).

Table 1. LBTS Message Overhead

FTP Size	Original	Improved
10000	28,070	10,200
100000	262,000	91,944
1000000	2,597,103	910,560

Our original implementation used a one-pass butterfly barrier as described in [6]. As the barrier is being processed, each processor exchanges information regarding the smallest simulation time known, as well as message transmit and receive counts. This becomes problematic when there are an excess of *transient messages*. Transient messages are messages sent by one processor but not yet received and processed by the recipient. A processor may be in the situation of having advertised a given simulation time T, and then discover that a smaller timestamp message is subsequently received. In the original implementation, all processors would realize at the end of the barrier that there were unaccounted for transient messages, and thus are not yet able to decide on a *LBTS*. When any given processor receives a transient message, it will propagate an *Update Message* through the barrier indicating that the transient message has been received, and potentially a new, lower simulation time. This implementation gives a minimum of one update message for every transient message, causing substantial network overhead.

We modified the implementation to use a two-pass butterfly barrier with vector message counts and simulation time values being exchanged between pairs of simulators, similar to that proposed by Mattern in [8]. This resulted in a substantial decrease in total network messages exchanged per *LBTS* computation, a factor of almost 3 (see table 1). With this implementation, the message count exchanged per *LBTS* computation is bounded by exactly $2n \lg n$ (where n is the number of processors), regardless of the number of transient messages.

We then noted that the total number of *LBTS* computations was larger than we expected. Our implementation was such that simulator B would respond to an *LBTS* request by simulator A at any point in time, regardless of whether or not B also needed to obtain an *LBTS* value. In our application, we found that this technique results in about one *LBTS* computation for every 25 events processed within a given simulator. Since each *LBTS* computation requires $2n \lg n$ (n is the number of simulators participating in the distributed simulation) messages sent on the network, its easy to see that the number of network messages used calculating an *LBTS* can be more than the number of messages actually communicating events.

We solved this excessive overhead by not allowing

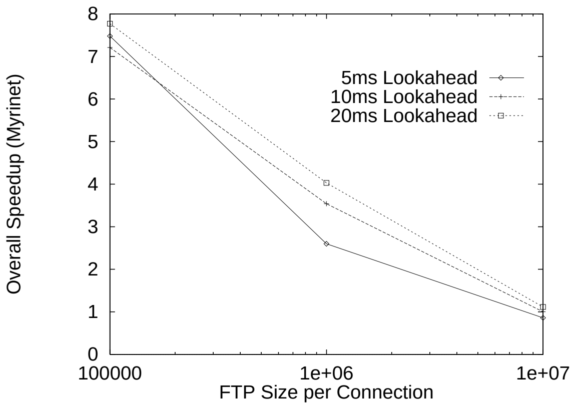

Figure 4. Overall Speedup Factors, Myrinet

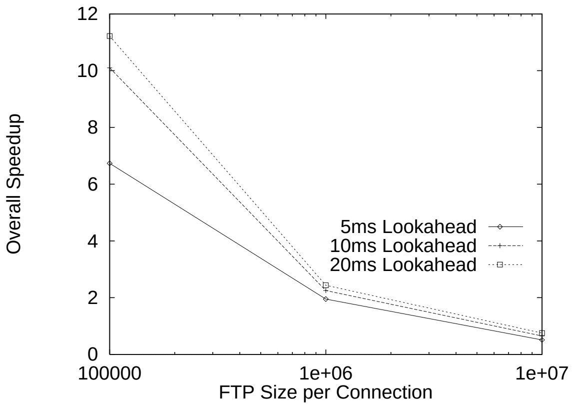

Figure 5. Overall Speedup Factors, TCP

any simulator to participate in a requested *LBTS* computation until such time as that simulator itself requires a new *LBTS* value. With this technique, the global *LBTS* value known to all simulators will advance by approximately the minimum lookahead value. We were able to increase the events per *LBTS* to about 60, thus significantly decreasing the network traffic overhead used by the *LBTS* computations.

5.2 Eliminating Socket Polling

A common design for a distributed simulation infrastructure (such as *RTIKIT*), requires that the application frequently call a background function to allow the infrastructure to perform periodic tasks, such as receiving data from other simulations and participating in *LBTS* computations. Our *RTIKIT* uses `RTIKIT_Tick()` for this purpose. As originally designed, the *RTIKIT* could not use blocking system calls to receive data from other simulators, since there was no assurance that such calls would not block forever. Thus the `Tick` calls were polling the sockets, spinning in a CPU loop waiting for data to arrive if there was nothing else to do. With this technique, the CPU scheduling algorithm used in the host operating system had a large impact on the overall performance. Since no *LBTS* computation can complete until all simulators have communicated their local information, any simulator which had been preempted by the operating system would delay all other simulators until it got rescheduled. Our timing data showed that many *LBTS* computations were completing in 2 to 3 milliseconds, but others were taking 40, 80 , or 120 milliseconds to complete. Further testing showed that a preempted host process on our operating system (Sun Solaris 5.5.1) may not reschedule for about 40 milliseconds, thus explaining the large delay times for the *LBTS* computations.

Our solution was to allow the end application (*ns* in this case) to use a blocking scheme to receive data from the network, but only when the application could be sure that it would not block forever. This puts more of a burden on the application, but results in a factor of 2 speedup on the overall running time of the distributed simulation. Actual performance results are given in section 6.

6 Results

In this section, we present some performance results. Our simulator is the popular and publicly available *ns*, which is a full featured and extensible network simulator using *TCL* and *C++* as the implementation platform. We used the techniques previously discussed to implement a distributed version of *ns* which we used for testing the performance.

The network model simulated consists of eight separate subnetworks, each containing twenty-five end hosts and one border router. Four of the eight subnetworks are designated *TCP Source* networks, and the other four are designated *TCP Sink* networks. Each of the four source subnetworks is connected to all of the four sink networks. The logical connections from the sources to the sinks are such that each of the four source subnetworks has some connections to all four of the sink subnetworks.

The distributed simulation was run on a set of eight Sun Ultra Sparc 1 Model 170 systems, each with 64Mb of main memory and a 167Mhz Sparc CPU. We tested the simulation using two different physical configurations, first using an eight-by-eight Myrinet network using the *Fast Messages* message passing software, and secondly using normal TCP protocol stacks on a 100Mbps Ethernet network. The simulation was distributed by mapping each of the eight Ultra's to a single simulated subnetwork, so each system managed twenty-five end host nodes and one border router. Each system used four *rlinks* to connect to other simulators. As a baseline, the simulation was also run on standard *ns* on a single processor.

Several different variations of data traffic were modeled, as follows. First, the amount of data generated by each TCP source was varied between 100,000 bytes, 1,000,000 bytes, and 10,000,000 bytes. The propagation delay on the *rlinks* (links connecting the different simulators) was varied between 5ms, 10ms and 20ms (which allows for differing lookahead values as previously mentioned). Each variation was run on both the Myrinet version of *RTIKIT* was well as the TCP version. The speedup factor, which compares the performance of the distributed simulation to a serial simulation, was calculated by dividing the elapsed time taken by the baseline single system simulation by the time taken by the distributed simulation. A large speedup factor indicates good parallelism and a smaller factor indicates less parallelism. A speedup factor of less than one indicates that the process ran slower on the parallel version compared to the serial version.

The speedup results for the Myrinet connected version are shown in figure 4, and for the TCP connected version in figure 5. The X axis is the variation of the amount of data transfered per connection (between 100,000 bytes, 1,000,000 bytes and 10,000,000 bytes) and the different plot lines show the different lookahead values. The Y axis is the speedup factor as previously discussed.

Note that the increased lookahead values do in general give better performance. However, one cannot just arbitrarily make the lookahead larger. The lookahead represents the smallest amount of simulation time that it can take for a simulated packet to traverse from one simulator to another, which is a function of the defined line speed of the connecting link times the packet size plus the defined propagation delay of the link, all of which are defined in the model being simulated. The current version of the *RTIKIT* software requires a fixed constant lookahead value, so the value specified must be the smallest propagation delay of any of the *rlinks* plus the line speed times the smallest packet size. If the network being modeled has only very high speed, low propagation delay links, then the lookahead value will be correspondingly small.

7 Conclusions

It turned out to be extremely straightforward to convert the main simulation engine (the event loop) of *ns* to run in a distributed environment. With a good set of tools, such as *RTIKIT*, the time management and event distribution is simple. However, the problems that face the network modeler are more challenging. Since *ns* was designed to run on a single serial simulator, and since we chose to *not* replicate the entire network model on each physical process, we needed some substantive enhancements to the syntax of the model description to provide for decomposition of a single model into sub-models to run in a distributed fashion. In some cases, this requires more effort on the part of the modeler (such as manually computing the routes between sub-models), and results in the simulator being more difficult to use. We plan for improvements in our implementation to alleviate some of this burden.

The performance of a distributed simulation can be problematic for long running simulations. The performance data given here suggests that much of the improvement in running the simulation in parallel is in the initial setup of the simulation, as opposed to the actual simulation itself. Further research is needed to achieve better results in this area. The distributed simulation environment does allow for simulation of larger models than would be possible in a serial environment, due to the additional memory on the extra processors.

References

[1] Seam-lss. http://www.seamlss.com, 1999.

[2] R. L. Bagrodia. Iterative design of efficient simulations using maisie. In *Proceedings of the 1991 Winter Simulation Conference*, December 1991.

[3] R. E. Bryant. Simulation of packet communications architecture computer systems. In *MIT-LCS-TR-188*, 1977.

[4] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. In *IEEE Transactions on Software Engineering*, September 1979.

[5] J. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, January 1999.

[6] R. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 1999.

[7] R. M. Fujimoto. Time management in the high level architecture. *Simulation*, December 1998.

[8] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*, 1993.

[9] S. McCanne and S. Floyd. The LBNL network simulator. Software on-line: http://www-mash.cs.berkeley.edu/ns, 1997. Lawrence Berkeley Laboratory.

[10] D. Nicol and P. Heidelberger. Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

[11] D. Nicol, M. Johnson, A. Yoshimura, and M. Goldsby. Ides: A java-based distributed simulation engine. In *Proceedings of the MASCOTS*, July 1998.

[12] D. M. Nicol and P. F. Reynolds. Problem oriented protocol design. In *Proceedings of the Winter Simulation Conference*, December 1984.

[13] K. Perumalla, R. Fujimoto, and A. Ogielski. Ted - a language for modeling telecommunications networks. *Performance Evaluation Review*, 25(4), March 1998.

[14] K. S. Perumalla and R. M. Fujimoto. Efficient large-scale process-oriented parallel simulations. In *Proceedings of the Winter Simulation Conference*, December 1998.

[15] Y. Rekhter and T. Li. Rfc 1771, border gateway protocol 4, March 1995.

[16] B. Unger. The telecom framework: a simulation environment for telecommunications. In *Proceedings of the 1993 Winter Simulation Conference*, December 1993.