Proceedings of the 2003 Winter Simulation Conference
S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds.

LARGE–SCALE NETWORK SIMULATIONS WITH GTNETS

George F. Riley

College of Engineering
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250, U.S.A.

ABSTRACT

When designing a network simulation environment intended specifically for modeling large–scale topologies, a number of issues must be addressed by the simulator designer. Memory requirements for network simulation engines can grow quadratically with the size of the simulated topology and can easily exceed available memory on modern workstations. The number of outstanding simulation events grows linearly with the number of packets in flight being modeled, and can lead to performance bottlenecks when managing a sorted event list of millions of events. Tracking the results of the simulation using a packet–level log file can result in excessive usage of disk space. We discuss the design of the *Georgia Tech Network Simulator* (*GTNetS*) with emphasis on how *GTNetS* addresses these issues. We give results from performance experiments showing the reduction in memory and event list size as a result of our design decisions.

1 INTRODUCTION

Computer based simulation is widely used in almost all areas of networking research. A number of high–quality simulation tools exist and are in widespread use. These tools allow researchers to test and validate new and existing protocols under a variety of conditions. An experimental protocol can be shown to work correctly in the presence of packet losses, packet re–ordering, lengthy delays, and lengthy round–trip times. This type of protocol validation is typically done on fairly small scale topology models, since the objective at this point is protocol correctness.

Once these protocols are known to be correct, the behavior of these protocols must be demonstrated in realistic size networks to insure that the performance of the protocol will be acceptable when deployed on a large scale. The venerable and widely used *ns2* (McCanne and Floyd 1997) can comfortably model networks of a few hundred to a few thousand network elements. Tools such as *pdns* (Riley, Fujimoto, and Ammar 1999) and *SSFNet* (Cowie, Ogielski, and Nicol 2002) can be used on topologies of up to 100,000 network elements, although this can be time consuming. The creation of larger–scale simulation topologies often consumes excessive amounts of CPU time and system memory, making this type of experimentation more daunting and therefore less common.

As the size of the simulated topology (and the number of data flows being modeled) grows to very large scales, computer resources in the simulation environment become the most critical issue, and generally are the limiting factor for the topology size of the network model. These resources include memory constraints, disk space constraints and and CPU time. See Riley and Ammar (2002) for a detailed analysis of the overhead in all three of these areas using the popular *ns2* simulation environment. This analysis indicates that, to achieve moderate to large–scale simulations for computer networks, the simulator must be designed from the beginning with scalability in mind. The *Georgia Tech Network Simulator* (Riley 2003) was designed in this way, and has demonstrated good scalability and efficiency with network models consisting of several million network elements. *GTNetS* is designed to utilize distributed simulation techniques, enabling linear scalability on a low–cost network of workstations. Here, we focus on efficiencies in a single sequential simulation only.

The remainder of this paper is organized as follows. Section 2 gives an overview of some of the network simulation tools presently in use. Section 3 discusses the tradeoffs and design decisions used in the *GTNetS* design that led to memory and CPU efficiency when simulating large–scale networks. Section 4 gives some performance results using *GTNetS*. Finally, section 5 summarizes this work.

2 RELATED WORK

There are a number of existing network simulation tools in existence that are in widespread use within the networking research community. Each of these has strengths and weaknesses, and no single simulation environment is suitable for all possible simulation requirements. In this section, we discuss briefly several existing simulators, and indicate the approximate level of scalability achieved by each.

The *ns2* Network Simulator. The venerable *ns2* simulator (McCanne and Floyd 1997) is certainly the most popular and widely used simulation environment for networking research. It includes detailed models of a number of TCP variations, a large number of queuing disciplines, several application models (such as HTTP web traffic), and extensive logging and tracing support. In addition, *ns2* has support for both wired fixed networks, and wireless ad–hoc networks. In the wireless domain, there are models for a number of ad–hoc routing protocols, including *Dynamic Source Routing* (*DSR*) (Johnson and Maltz 1996), *Ad–Hoc On–Demand Distance Vector* (*AODV*) (Perkins and Royer 1999); as well as a very detailed model of the IEEE 802-11b *MAC* protocol specification. *ns2* can comfortably model network topologies up to about 1,000 network elements with the default routing methods, and about 16,000 network elements using our *NIx–Vector* (Riley, Ammar, and Fujimoto 2000) routing method.

Furthermore, by using parallel and distributed simulation methods, we have shown good scalability for *ns2*, by extending the basic simulation model to run on a network of workstations. Our *Parallel/Distributed ns* (*pdns*) (Riley, Fujimoto, and Ammar 1999), has been shown to scale to a network topology of 250,000 network elements (Riley, Ammar, and Fujimoto 2000).

The *GloMoSim/Parsec* Simulation Engine. Another well known and popular simulator is *GloMoSim* (Zeng, Bagrodia, and Gerla 1998), which is built on top of the *Parsec* (Bagrodia, Meyer, Takai, Chen, Zeng, Martin, Park, and Song 1998) simulation engine. *GloMoSim* has been designed specifically to model wireless networks, and is designed to run in a parallel environment on a tightly coupled symmetric–multiprocessor system. It has very detailed models of the 802-11b *MAC* protocol, a number of routing protocols (including Bellman/Ford, AODV (Perkins and Royer 1999), DSR (Johnson and Maltz 1996), and others), TCP, and various application models.

GloMoSim was designed from the beginning to run in parallel, and thus can achieve good parallel performance in some cases. However, the *GloMoSim* simulator uses tightly coupled shared memory for inter–process communication, which leads to limited scaling. In large–scale simulations, main memory is often the first resource to become exhausted. The dependence on shared–memory message passing limits the scale of a *GloMoSim* simulation to a single computing system, with the corresponding limits on available memory. Additionally, the original design criteria for the *GloMoSim* product were focused mainly on performance rather than scalability. Experiments by the developers of *GloMoSim* have demonstrated scalability up to about 1000 nodes.

The Scalable Simulation Framework (*SSFNet*). The *SSFNet* (Cowie, Nicol, and Ogielski 1999, Cowie, Liu, Liu, Nicol, and Ogielski 1999) simulator was originally developed at Rutgers University (DIMACS), in collaboration with Dartmouth University, and is presently maintained and distributed by Renesys Corp. The product was designed from the beginning to be scalable and to give good performance across a wide range of topology scales. The product is available both in a *Java* environment, as well as a *C++* implementation. Like *GloMoSim*, *SSFNet* is designed to run on a tightly coupled *SMP* system, relying on a custom designed thread scheduler and efficient memory–to–memory message passing to achieve good parallel performance. Unlike *GloMoSim*, the *SSF* simulator was designed from the beginning with scalability in mind, and thus is careful about limiting memory consumption wherever possible. The *SSF* simulator has been demonstrated by the developers on network topologies as large as 100,000 network elements.

The *OpNet* Network Simulator. The *OpNet* (Bertolotti and Dunand 1993) simulator is a widely–used commercial software product developed by OpNet Technologies Inc. This simulator contains very detailed models of a large number of network devices, including most commercial routers, switches, and hubs; as well as a number of wireless devices and MAC protocols. The product is commercially successful, and has a large installed customer base. The product is presently limited to a single process running on a single computing platform, but a version able to execute on parallel processors in under development. Opnet also includes a *High Level Architecture* (*HLA*) interface, but this interfaces supports interoperability with other tools such as traffic generators, rather than distributed execution. Furthermore, efforts to model large networks using OpNet have had only limited success, primarily due to the limitations of single process execution. As part of our existing *COMPASS* research effort, we developed a rudimentary distributed version of this product (Wu, Fujimoto, and Riley 2001), but achieved only limited success.

3 *GTNetS* DESIGN

In this section, we discuss the design decisions in *Georgia Tech Network Simulator* that allowed more efficient simulation of very large–scale networks. These efficiencies fall into three basic categories:

1. Reducing Event List Size
2. Managing Memory
3. Reducing Log File Size

Each of these areas are discussed in more detail below.

3.1 Reducing Event List Size

We have three optimizations that substantially reduce the overall size of the pending event queue. Since insertions

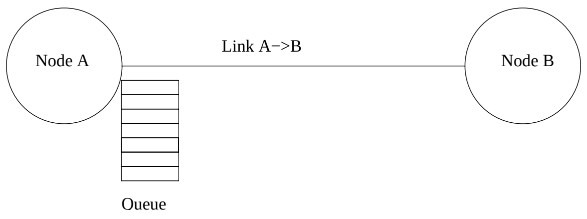

Figure 1: Simple Network Topology

into sorted queues are $O(\lg N)$, reducing the size of this queue results in improved execution time.

3.1.1 *FIFO* Receive Queues

Consider the simple network topology shown in figure 1. This consists of two nodes, A and B, a simplex link connecting the nodes, and a packet queue associated with the link. To model the correct behavior of this topology, the simulator typically needs two pending simulation events for each packet transmitted. Each packet transmission will schedule one simulation event to indicate that the packet is received by the receiver, and a second event to indicate that transmission is completed at the sender. Note that these two events are at different simulation times due to the speed–of–light propagation delay on the link. Pseudo–code for a packet transmission action is similar to the following:

```
PROCEDURE TransmitPacket
  /* Process Packet Tx Event */
  IF (Link Not Busy) THEN
    /* Calculate time to transmit */
    TransmitTime =
        PacketSize / LinkBandwidth;
    /* Schedle Rx Event at Receiver */
    SchedulePacketRxEvent
        (TransmitTime + PropogationDelay);
    /* Schedule Link Free Event */
    ScheduleLinkFreeEvent(TransmitTime);
    Set Link Busy;
  ELSE
    /* Link is busy */
    /* Enque the packet for later Tx */
    EnquePacket();
  END IF;
END Transmitpacket;

PROCEDURE LinkFreeEvent
  /* Process a LinkFree Event */
  Set Link Not Busy;
  IF (Queue Not Empty)
    Remove Packet From Queue;
    TransmitPacket;
  END IF;
END LinkFreeEvent;
```

Since each packet transmission requires two events, the count of pending events, assuming all links are active, is at least $2N$ (N is the number of simplex links defined in the simulated topology). In reality, it is often several times N if there is a significant propagation delay on the link. For a simulation of 1 million full duplex links, the event queue can easily exceed 10 million pending events. Since the time to insert an event in a sorted queue is $O(\lg N)$ in the general case, this can result in significant CPU overhead. We point out that the *Calendar Queue* (Brown 1988) (which is the default event scheduler in *ns2*) can in some cases realize $O(1)$ insertions, but degenerates to $O(n)$ insertions in extreme cases. For *GTNetS* we chose an $O(\lg N)$ event scheduler based on the *C++* standard template library *Map* container, which works consistently in all cases.

Our design for packet transmission and receipt processing in *GTNetS* reduces the number of pending events from two events per packet in–flight to one event per simplex link, using two optimizations described below. While the number of packets in–flight and number of simplex links have the same “Big–O” limit, in practice the packet in–flight count is several times the simplex link count.

We observed that the timestamps for packet transmissions on a single simplex link are strictly in increasing order of simulation time. A packet transmitted on link A->B must always be sent at a simulation time greater than that of the previous packet transmitted on that link. With this in mind, we replaced the scheduling of receive events for all packets in–flight with a single *first–in–first–out* queue at each receiver. This *FIFO* queue has a constant insertion and removal time. At any point in time, there is only one event in the sorted event queue for each simplex link with packets in–flight, which will process the packet receipt event *for the earliest pending packet on that link only*. All other pending packets for that link are simply stored in the *FIFO* queue at the receiver. When the packet receive event for a given link is removed from the sorted event queue and processed, another pending event is scheduled for the next packet, if one is present in the *FIFO* queue. For simplex links with significant propagation delays and potentially large numbers of packets in–flight, this optimization results in a substantial reduction in the size of pending event list.

3.1.2 Abstract Packet Queues

Secondly, we noted that in many cases the queuing delay for a packet in a *FIFO* queue (such as a simple DropTail style packet queue) can be deterministically calculated when the packet is inserted in the queue. This fact allows us to reduce the size of the pending event list by a factor of two by eliminating the need for the *LinkFree* event, as follows.

When a packet is transmitted on a non–busy link, the packet receipt event is created normally, either by scheduling the event in the sorted event queue, or by appending it in the

FIFO event queue as described above, and the link is set to *busy*. However, the *LinkFree* event is not created. Instead, we maintain a *FIFO* data structure at the transmitting link, called the *Abstract Queue Info Deque* (*AQID*), that stores the transmission start time, transmission end time, and the size of the packet. When another packet is later transmitted on the same link, we first remove any stale packets from the *AQID* (packets that have already completed transmitting), and the number of bytes in the DropTail queue is reduced accordingly. If there are still remaining entries in the *AQID*, we calculate the time that the packet is to be received at the receiver by adding the packet transmission time and propagation delay to the *transmission end time* of the most recent *AQID* entry, and schedule the packet receipt normally. If the *AQID* is empty, the link is no longer busy and the packet transmission is handled normally. We call this method *Abstract Queuing*, since packets are never actually queued at the transmitter, but rather forwarded directly to the receiver with the appropriate queuing delay accounted for in the packet receipt time.

By eliminating the scheduling and processing of *Link-Free* events, we reduce the total number of events processed due to packet transmissions by a factor of two. We hasten to point out that this *Abstract Queuing* method can only be used in cases where the queuing delay of a packet can deterministically be calculated when the packet is enqueued. For links with collision detection or avoidance methods, such as Ethernet LANs or 802.11 wireless links, this optimization cannot be used. Further, this optimization has little effect on the instantaneous event list size, since at any one time there would be at most one *LinkFree* event for each simplex link in the topology.

3.1.3 Timer Buckets

Finally, we observed that for simulations modeling TCP data flows, there is always at least one pending event in the event queue for every active TCP connection. Every active TCP flow has at least one pending *Timeout* event for the most recently transmitted sequence number. These events are almost always later canceled and removed from the pending event queue when an acknowledgment packet is received. Timeout events are only actually scheduled and acted upon when data packets are lost along the path from the sender to the receiver. Further, we observed that in a simulation environment, these timeout events are typically created and processed with a high degree of accuracy in the timestamp of the event. Timeout events are scheduled and processed with nano–second accuracy or better, resulting in a somewhat unrealistic simulation. In actual end–systems, timer events are processed on more granular *Clock Tick* intervals, which are typically on the order of milliseconds.

With these issues in mind, we designed *GTNetS* using the concept of *Timer Buckets*, as follows. We start with specifying a user configurable timer buckets interval, which defaults to ten milliseconds. We then create a simple vector data structure, with each entry in the vector representing all pending timeout events for a fixed timer buckets interval in the future. For example, the zeroth entry in the vector contains all timeout events scheduled for ten milliseconds in the future, the next entry is all timeouts for twenty milliseconds, and so forth. Each of these entries is called a *Timer Bucket*. We note that, for each timer bucket, entries are scheduled in strictly increasing timestamp order, and thus a simple *FIFO* queue, with constant insertion and removal time, can be used to maintain all pending timeouts for each bucket. When scheduling a new timeout event, we simply append the event to the tail of the *FIFO* queue for the appropriate timer buckets, which is a constant time operation. If the bucket was empty prior to the insertion, we schedule an event in the sorted event queue that represents the earliest timeout for any event in that bucket. When a timeout event is canceled (which does not necessarily occur in *FIFO* order), we do not remove the event from the *FIFO* queue unless it is the earliest pending event for that bucket. Rather, we simply mark it as canceled. When a timeout event is scheduled for processing, it will by definition be the earliest pending event for its bucket. At that time, it is removed from the head of the *FIFO* queue. If the new head is marked as canceled it is also removed (as are all consecutive canceled events that become the head of the queue). If the resulting *FIFO* queue is non–empty, a new sorted event list entry is scheduled for the appropriate simulation time for later processing.

The timer buckets optimization reduces the size of the event list from $O(k)$ where k is the number of active TCP flows, to $O(j)$ where j is the number of timer buckets. The number of timer buckets is a function of the computed TCP round trip times and cannot be known precisely in advance. However, assuming a ten millisecond bucket size, the number of buckets in a reasonable simulation is no more than a few hundred. Interestingly, this optimization also results in a more realistic simulation, since real TCP implementations schedule and process timeouts in coarse grain *clock tick* intervals which are analogous to our timer bucket interval.

3.2 Managing Memory

A second area where care must be taken in simulator design is in memory management. While the *GTNetS* design is careful with memory usage in almost all areas, we present three particular memory management design decisions that result in substantial savings in the overall memory footprint of the simulation.

3.2.1 Routing Tables

The amount of memory required to create and maintain simulated routing tables in a large–scale simulation can become excessive. In order to route packets through a network, each node must make a routing decision to determine the *Next Hop* node that represents the shortest path (or some other routing metric) to the ultimate destination. This decision typically involves referencing a *Routing Table* that has, in the extreme, one entry for every possible packet destination at each possible routing decision point. This routing table lookup method results in memory requirements of $O(N^2)$, where N is the number of nodes in the simulated topology. The popular *ns2* simulator uses a hash–based routing lookup by default, which reduces this limit somewhat by ignoring unused entries. The *SSFNet* simulator uses routing aggregation methods based on *IP* prefixes, which also reduces the overall memory requirements. In *GTNetS*, we use two optimizations which reduce or eliminate the memory requirements for routing tables.

First is the use of the on–demand *NIx–Vector* routing method by default. This method does not use routing tables at all, but rather uses a source–based routing method whereby routing information is contained in each packet. Routes are computed as needed, and are cached at packet sources for later reuse. The *NIx–Vector* representation for the routing information is very compact and introduces little overhead in the packet routing decisions. The *NIx–Vector* routing approach was previously published (Riley, Ammar, and Fujimoto 2000) as part of our work with *pdns*, so details of this method are omitted here. *GTNetS* also supports the creation of static routing tables (based on a all–pairs shortest–path–first algorithm) if the presence of routing tables is required for the simulation results desired.

Next is the elimination of any routing information (either the *NIx–Vector* creation or the static routing tables) at *leaf nodes* in the simulation. Consider the two simple topologies shown in figures 2 and 3. Figure 2 shows a portion of a topology with five *leaf* nodes, connected to the remainder of the topology by a single gateway using point–to–point links. Figure 3 shows a similar topology, with five leaf nodes on an Ethernet LAN and a single gateway.

For the case of the point–to–point links in figure 2 it is easy to observe by inspection that routing decisions at the leaf nodes are trivial. In all cases, any packet generated by a leaf node is unconditionally forwarded to the gateway, excepting packets addressed to the leaf node itself. The *GTNetS* routing computations will never calculate routing information for leaf nodes with a single point–to–point link, resulting in a substantial memory savings when using the static routing method. Surprisingly, the popular *ns2* simulator does not implement this simple optimization.

The Ethernet LAN case shown in figure 3 is less obvious, but in fact can use the same optimization. When creating an Ethernet LAN in *GTNetS*, an optional *single gateway* can be specified which denotes the gateway node as shown. Packets generated at the leaf nodes can either be destined for another leaf on the same LAN, or somewhere else in the network. *GTNetS* checks for local routes using *IP Addresses* and *Network Masks* much like real network implementations. If a local route is found, then no routing decision is needed. If a local route is not found, then the packet is unconditionally forwarded to the gateway as described above. In neither case is any routing information required, and thus route computations can be skipped for these leaf nodes.

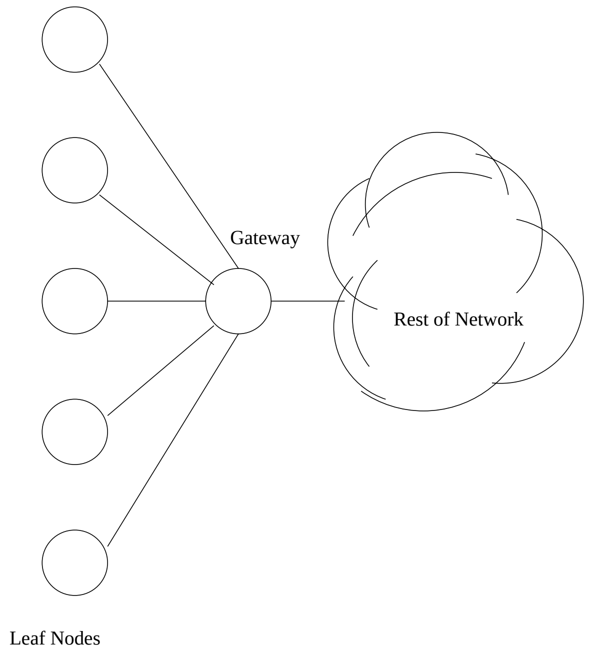

Figure 2: Point–to–Point Leaf Nodes

3.2.2 Representing Packets

In a large–scale simulation with millions of nodes and flows, the number of *Packets* flowing through the network can be tens or hundreds of millions. Thus, careful attention must be given to efficient representation of simulated packets in order to keep the overall memory footprint manageable. However, there are design tradeoffs to be considered which may result in less flexibility and ease of use if memory usage is the only consideration.

A typical packet in *GTNetS* is shown in figure 4. The packet consists of two parts. First are three fixed position fields identifying the packet, followed by the *PDU Stack*. The first three fields are included for ease of use of the simulator, and have no correspondence to real network packets. The unique identifier is a 32 bit value unique to this packet, and can be used in the simulator to track individual through the network. The timestamp field indicates the simulation time when this packet was created. The size

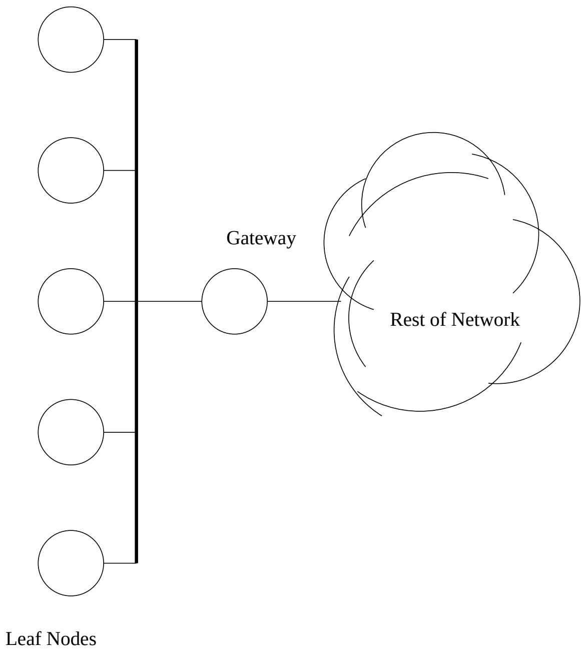

Figure 3: Ethernet Leaf Nodes

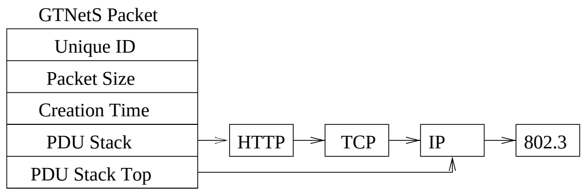

Figure 4: Typical *GTNetS* Packet

field indicates the size, in bytes, of the packet including all *Protocol Data Units* (*PDUs*) associated with this packet. This size is the actual size of the data being transmitted, rather than the amount of memory used by the *GTNetS* packet representation. For example, the *GTNetS IPV4 PDU* is a C++ class object with a total size of 32 bytes. However, the size used to calculate the transmission time of the *IPV4 PDU* is of course 20 bytes plus the size of any *IP Options*.

The *PDU* stack consists of *PDU*s which are appended and removed from the packet as it moves down and up the protocol stack. When a new packet is generated by an application (for example a browser creating an HTTP request), the HTTP header is created and pushed on the *PDU* stack. At the next lower layer (TCP for example), an appropriate *PDU* is again generated and pushed, and the packet continues down the stack. When a packet is received from a link at a layer two processor (for example IEEE 802.3(IEEE 2000)), the *PDU* is popped and examined. However, in our design we do not actually remove the *PDU* pointer from the data structure, but rather simply adjust a *Top* pointer. This allows for the case where a *PDU* that is popped from the stack will immediately pushed back on (for example at the *IPV4* layer at interior routers) and prevents an unnecessary *delete* and *new* operation.

It is clear that this approach is somewhat memory efficient in that each packet contains information for only those protocol data units that are present in the packet. In contrast, *ns2* packets have data fields for all possible *PDU*s, regardless of whether that *PDU* is actually present in each particular packet. However, we could have gained additional memory efficiency by bit–packing and byte–packing, as is specified by RFC 760 (Postel 1980a) for example. We made the design decision in this case to sacrifice some memory efficiency for ease–of–use and flexibility. Since one use of network simulators is to study the affect of changes or additions to existing protocols, our method of *PDU* representation allows easy addition of new fields in protocol headers while at the same time maintaining an accurate representation of the actual size of a packet to be transmitted on a communications link.

3.3 Reducing Log File Size

Network simulations typically create a log file which traces the flow of packets through the simulated network. In the extreme case, every packet transmission and receipt on every link, plus every packet enqueue and dequeue operation at every queue is logged to a disk file. With moderate sized simulations of a few hundred network elements, this method of packet logging is reasonable. However, when attempting to model millions of network elements and potentially billions of packet transmissions, this method becomes unwieldy. Our *GTNetS* design addresses this issue in two ways.

First, we provide built–in statistics collection that in some cases can obviate the need for the log file. In many cases, the log file is used as input into a post–processing analysis program which further reduces the data and produces statistics. For example, if the simulation is modeling web browsing activities, the log file analysis program might produce statistics regarding web response time. Other metrics gathered by the log file analysis might include average queue length, queuing delay, or loss rate. Built into the *GTNetS* implementation are a number of statistics gathering mechanisms which calculate these statistics as the simulation is executing, if requested by the user. For example, the web browsing models included with *GTNetS* will gather metrics for web response time, and will output these statistics as either a histogram or a *CDF*. All queue models in *GTNetS* will keep statistics on average size and lost packets, and will log that information to a summary file if requested. Thus, in many cases, the need for a detailed packet log file may not exist.

Secondly, we provide very fine–grained control over the logging of packet events, which leads to the logging of only those events that are of interest, and ignoring others.

Log file entries in *GTNetS* are optionally created at every protocol stack layer, both when packets are requested from a higher layer, and when packets are indicated from a lower layer. At each protocol layer, the logging of packets is specified to be in one of three states, *enabled*, *disabled*, or *default*. Additionally, each node in the simulated topology has the same three packet logging states. The state of the logging at each protocol stack layer and at each node is optionally specified individually when the topology is created. By appropriate settings of the logging state at nodes and protocol stack layers, the logging of packets at a given node can either be always off (if the state is *disabled*), always on (if the state is *enabled*, or deferred to the decision of the protocol stack layer if the state is *default*. Further, it may be desirable for all packets for one or more data flows to be logged regardless of the logging state at nodes or layers. This can be accomplished by a *force log packet* flag in the *GTNetS* packet.

When a packet arrives at a particular protocol stack layer at a particular node, the algorithm for determining if the packet should be logged is as follows:

```
if (no log file) DO NOT LOG;
else if (force log packet) LOG
else if (node = disabled) DO NOT LOG
else if (node = enabled) LOG
else if (protocol = disabled) DO NOT LOG
else if (protocol = enabled) LOG
else DO NOT LOG
```

In addition to the selective logging of packet information as described above, *GTNetS* allows detailed specification of exactly what information should be logged at each protocol stack layer.For example, a *PDU* header for TCP in *GTNetS* consists of eleven individual data items, corresponding to the protocol information specified in RFC 761 (Postel 1980b). Each of those can be selectively enabled or disabled by the simulator user, allowing for the logging of only desired information. For example, it may be the case that a particular simulation has no use for source port and destination port information for TCP packets, but needs the sequence number and acknowledgment numbers from each packet. Using the selective enabling and disabling of individual protocol items, only the desired information is logged to the disk file.

A sample excerpt from a *GTNetS* log file is shown in figure 5. This shows a single TCP flow from node 0 to node 4, with intermediate routers at nodes 2 and 3. The TCP endpoint at node 0 creates the `SYN` packet at time 1.77891, which is routed by nodes 2 and 3, and arrives at node 4 at time 1.88897. Node 4 creates the `SYN|ACK` packet which is received at node 0 at time 1.99904. Node 0 responds with an `ACK` packet followed by 512 data bytes. Each log file entry for *IPV4* shows the `TTL`, the layer 4 protocol identifier (6 for TCP in this case), the source and destination *IP* addresses, and the packet unique identifier.

4 EXPERIMENTS

We designed a set of experiments to demonstrate the effectiveness of the efficiency optimizations previously discussed. For these experiments, we used the *Campus Network* topology (Nicol 2002) shown in figure 6 as the basic building block for the simulation. To scale the topology to larger sizes, we replicated the campus network multiple times, and connected the subnetworks with gateway links as shown in the figure.

The first set of experiments demonstrates the reduction in the pending event list size for the optimizations discussed. These experiments used a fixed topology size 200 campus networks, which results in 107,600 nodes and 100,800 flows. The size of the pending event list was tracked at one second intervals. These results are shown in figure 7. In this figure, the x–axis is the simulation time and the y–axis is the size of the pending event list using a logarithmic scale. The three individual plots on the figure represent the size with no optimizations, with the *FIFO* receive queue optimization, and with the *FIFO* receive queue plus timer buckets. We can see that with none of the optimizations, the event list grows to nearly 1.4 million events. The *FIFO* receive queue reduces the event list size to a maximum of about 100,000 entries, and with both optimizations the maximum size of the event list is only about 10,000. Figure 8 shows the total number of events scheduled for the same simulations, with the x–axis on a linear scale. This figure shows clearly the reduction in total event count by using the abstract queuing optimization, reducing the event count by nearly a factor of two. Also the timer buckets optimization reduces the event count slightly, since many timer events are canceled before getting scheduled.

The next set of experiments shows the affect of the memory efficiency optimizations discussed. We used the same campus network topology as above, but this time used varying numbers of campus networks (from 10 to 400) and tracked the maximum memory footprint of the simulation. These results are shown in figure 9. In this figure, the x–axis is the count of the total number of nodes in the topology, and the y–axis is size of the memory footprint in megabytes. The first plot is with *NIx–Vector* routing enabled, the second is with static routing and the single gateway optimization, and the last is with static routing. The hardware that we used for these experiments had 2Gb of main memory, which is the limiting factor for maximum topology size. The *NIx–Vector* routing method can process up to 330 campus networks (177,000 nodes) within this limit, where the static routing method can only achieve 40 campus networks (21,000 nodes). The single gateway optimization with static routing shows an improvement over plain static routing, achieving a maximum size of 150 campus networks (80,000 nodes).

```
1.77891 N0 L4-TCP 10000 80 0 0 SYN 0 0 L3-4 64 6 192.168.0.1 192.168.1.1 1
1.78391 N2 L3-4 63 6 192.168.0.1 192.168.1.1 1
1.88396 N3 L3-4 62 6 192.168.0.1 192.168.1.1 1
1.88897 N4 L3-4 61 6 192.168.0.1 192.168.1.1 1 L4-TCP 10000 80 0 0 SYN 0 0
1.88897 N4 L4-TCP 80 10000 0 0 SYN|ACK 0 0 L3-4 64 6 192.168.1.1 192.168.0.1 2
1.89398 N3 L3-4 63 6 192.168.1.1 192.168.0.1 2
1.99403 N2 L3-4 62 6 192.168.1.1 192.168.0.1 2
1.99904 N0 L3-4 61 6 192.168.1.1 192.168.0.1 2 L4-TCP 80 10000 0 0 SYN|ACK 0 0
1.99904 N0 L4-TCP 10000 80 0 0 ACK 0 0 L3-4 64 6 192.168.0.1 192.168.1.1 4
1.99904 N0 L4-TCP 10000 80 0 0 0 0 512 L3-4 64 6 192.168.0.1 192.168.1.1 5
2.00404 N2 L3-4 63 6 192.168.0.1 192.168.1.1 4
2.00409 N2 L3-4 63 6 192.168.0.1 192.168.1.1 5
2.10409 N3 L3-4 62 6 192.168.0.1 192.168.1.1 4
2.10456 N3 L3-4 62 6 192.168.0.1 192.168.1.1 5
2.10910 N4 L3-4 61 6 192.168.0.1 192.168.1.1 4 L4-TCP 10000 80 0 0 ACK 0 0
2.10961 N4 L3-4 61 6 192.168.0.1 192.168.1.1 5 L4-TCP 10000 80 0 0 0 0 512
2.10961 N4 L4-TCP 80 10000 0 512 ACK 0 0 L3-4 64 6 192.168.1.1 192.168.0.1 7
```

Figure 5: *GTNetS* Log File Excerpt

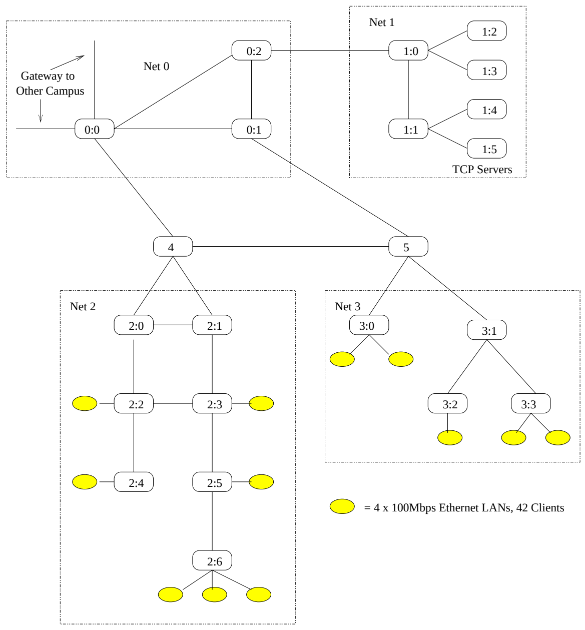

Figure 6: Basic Campus Network

5 SUMMARY

We have demonstrated experimental results showing that careful attention to event list management and memory management in a network simulation can lead to significant improvements in the overall topology size achievable in a single simulation execution. We achieved a topology size of 177,000 nodes in a single simulation address space on an inexpensive workstation with 2Gb of main memory, representing a factor of 8 improvement over the size limit without our optimizations. This enables multi–million node simulations using distributed simulation methods on a network of workstations.

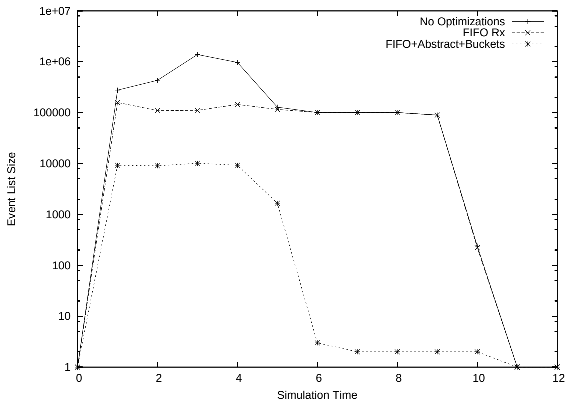

Figure 7: Event List Size

ACKNOWLEDGMENTS

This research was sponsored in part by NSF Grants Nos. ANI-9977544, ANI-0225417, ECS-0225471, and DARPA Contract number N66002-00-1-8934. The content of this publication does not necessarily reflect the position or the policy of the sponsors and no official endorsement should be inferred.

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.

Bertolotti, S., L. Dunand. 1993. Opnet 2.4: an environment for communication network modeling and simulation. In *Proc. European Simulation Symposium*.

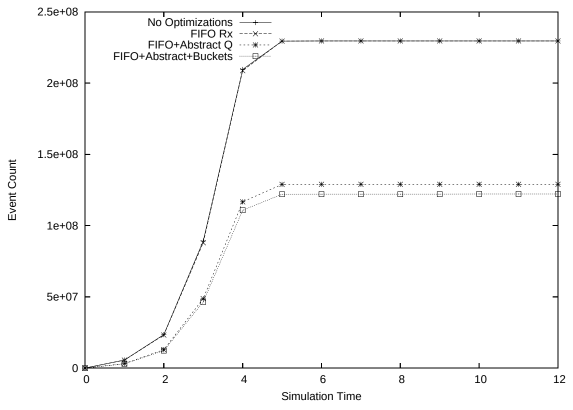

Figure 8: Total Event Count

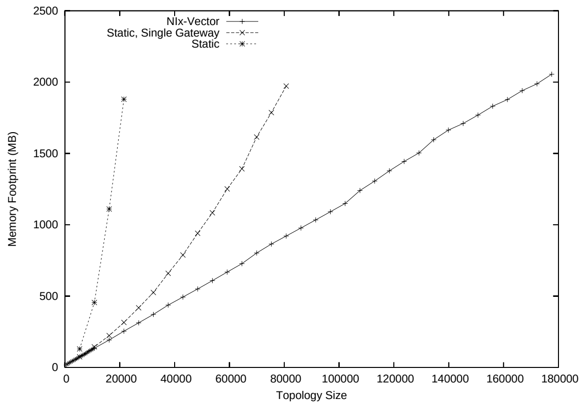

Figure 9: Memory Usage

Brown, R. 1988. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. In *Communications of the ACM*, Volume 31, 1220–1227.

Cowie, J., H. Liu, J. Liu, D. Nicol, and A. Ogielski. 1999. Towards realistic million-node internet simulations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*.

Cowie, J., A. Ogielski, and D. Nicol. 2002. The SSFNet network simulator. Software on-line: <http://www.ssfnet.org/homePage.html>. Renesys Corporation.

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

IEEE 2000. Ieee standard 802-3 carrier sense multiple access with collision detection(CSMA/CD) access method with physical layer specifications. *Institute of Electrical and Electronic Engineers*.

Johnson, D. B., and D. A. Maltz. 1996. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, Kluwer Academic Publishers, Chapter 5:153–181.

McCanne, S., and S. Floyd. 1997. The LBNL network simulator. Software on-line: <http://www.isi.edu/nsnam>. Lawrence Berkeley Laboratory.

Nicol, D. M. 2002. The baseline campus network explained. <http://www.cs.dartmouth.edu/∼nicol/NMS/baseline/>. DARPA Network Modeling and Simulation (NMS).

Perkins, C. E., and E. M. Royer. 1999. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 90–100.

Postel, J. 1980a. Internet RFC760: Dod standard internet protocol. Network Working Group.

Postel, J. 1980b. Internet RFC761: Dod standard transmission control protocol. Network Working Group.

Riley, G. F. 2003. The Georgia Tech network simulator. In *Proceedings of Workshop on Models, Methods, and Tools for Reproducible Network Research (MoMeTools) (to appear)*.

Riley, G. F., and M. H. Ammar. 2002, January. Simulating large networks: How big is big enough? In *Proc. First International Conference on Grand Challenges for Modeling and Simulation*.

Riley, G. F., M. H. Ammar, and R. M. Fujimoto. 2000. Stateless routing in network simulations. In *Proc. the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

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

Tyan, J.-Y., and C.-J. Hou. 2001. Javasim: A component-based compositional network simulation environment. In *Proceedings of the Western Simulation Multiconference, Communication Networks And Distributed Systems Modeling And Simulation*.

Wu, H., R. Fujimoto, and G. Riley. 2001. Experiences parallelizing a commercial network simulator. In *Proceedings of the Winter Simulation Conference*.

Zeng, X., R. Bagrodia, and M. Gerla. 1998. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*.

AUTHOR BIOGRAPHY

GEORGE F. RILEY is an Assistant Professor of Electrical and Computer Engineering at the Georgia Institute of Technology. He received his Ph.D. in computer science from the Georgia Institute of Technology, College of Computing, in August 2001. His research interests are large–scale simulation using distributed simulation methods. He is the developer of *Parallel/Distributed ns2* (*pdns*), and the Georgia Tech Network Simulator (*GTNetS*). He can be reached via email at `<riley@ece.gatech.edu>`.