The Georgia Tech Network Simulator

George F. Riley
Georgia Institute of Technology
School of Electrical and Computer Engineering
Atlanta, GA. 30332-0250
riley@ece.gatech.edu

ABSTRACT

We introduce a new network simulation environment, developed by our research group, called the *Georgia Tech Network Simulator* (*GTNetS*). Our simulator is designed specifically to allow much larger–scale simulations than can easily be created by existing network simulation tools. The design of the simulator very closely matches the design of real network protocol stacks and hardware. Thus, anyone with a good understanding of networking in general can easily understand how the simulations are constructed. Further, our simulator is implemented completely in object–oriented C++, which leads to easy extension by users to experiment with new or modified behavior of existing simulation models. Our tool is designed from the beginning with scalability in mind, including the support for distributed simulations on a network of workstations as part of the basic design.

We give an overview of the features of *GTNetS*, and present some preliminary scalability results we have obtained by running *GTNetS* on a computing cluster at the Pittsburgh Supercomputer Center.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General

Keywords

Network Simulation, Large–Scale Simulations, Distributed Simulation

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.

ACM SIGCOMM 2003 Workshops
August 25 & 27, 2003 Karlsruhe, Germany

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. For protocols designed for wired networks, tools such as *pdns*[15] and *SSFNet*[3] can be used on topologies of up to 100,000 network elements, although this can be time consuming. The venerable and widely used *ns2*[8] can confortably model networks of a few hundred to a few thousand network elements. 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.

The author's prior work with *Parallel/Distributed ns* (*pdns*) made it clear that attempting to backstitch scalability and performance into an existing simulation environment is difficult at best. With this in mind, we undertook to develop a new network simulation environment, designed from the beginning to be distributed, scalable, and easy to use. This simulation environment is called the *Georgia Tech Network Simulator* (*GTNetS*). Like any tool designed for use by the research community, *GTNetS* will never be completely finished, and will evolve over time as the needs and requirements of the research community change. However, it is presently fully capable of large–scale simulations of routers, end–systems, LAN's, and various end–user applications.

The motivation for the creation of this new environment is not to replace or compete with any of the existing simulation tools in widespread use. Rather, we hope that researchers will find our tool useful in instances where existing tools cannot easily model the network functions being studied, or cannot achieve the scale needed to produce the desired results. Our tool is released and freely available to the networking research community in the hope that it will be useful, and that researcers will contribute new and improved models. The software can be downloaded from our web page [12].

In the remainder of this paper we will discuss the basic design and features of *GTNetS* in section 2, some sample simulation scripts in section 3, and some representative results in section 4. Finally, we will give a summary and future directions in section 5.

2. THE DESIGN OF GTNetS

This section discusses the design and capabilities of the newly developed *Georgia Tech Network Simulator* (*GTNetS*). *GTNetS* was conceived by the author while teaching a graduate level class in network simulation methods, which was primarily focused on using *ns2* for all experiments. Many of the students pointed out difficulties in using *ns2* to achieve the stated goals of the lab projects. It became clear that, even though *ns2* is an excellent and widely used research tool, there are a number of basic design deficiencies that make it difficult to model certain aspects of network simula-

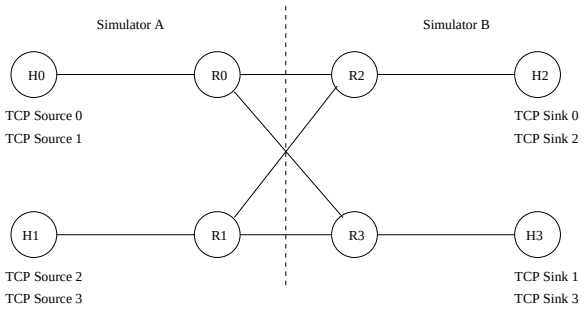

Figure 1: Simple Distributed Network Model

tion experimentation. Further, the author's experience in implementing *Parallel/Distributed ns* (*pdns*) made it clear that achieving further improvements in topology scale with the baseline *ns2* product would be difficult. The design of *ns2*, using the hybrid of *Tcl* and *C* languages, leads to substantial memory consumption in many cases.

In the Summer of 2002, we began a research effort to create a new network simulation environment that could be used in cases where existing simulations lacked the capabilities to create the desired experiments. This effort has resulted in the *Georgia Tech Network Simulator*. *GTNetS* has a number of basic design goals, which we categorized into several high–level goals

2.1 Distributed Simulation and Scalability

Care must be taken in the basic design of the simulator when a distributed simulation is planned. When a single simulation topology model is decomposed into a number of small sub–models, and executed in a distributed environment, there will often be simulated links that connect nodes which are defined on two simulation processes. Consider for example the simple topology in figure 1. In this example, the link connecting router $R0$ and router $R2$ has its endpoints on two different simulators. Simulator A has no representation of router $R2$, and simulator B has no representation of router $R0$. *GTNetS* allows the creation of *Remote Links*, in which only the local node must be specified. Remote links are assigned IP addresses and address masks. No other information about the connectivity of the remote link is needed. At initialization time of the distributed simulation, any remote links having matching *Network Address* portions of their IP addresses are assumed to be connected, and any packet generated at router $R0$ will be forward to simulator B and delivered to router $R2$. This is nearly identical to the approach used by *pdns*. The *SSFNet* simulator is designed to operate on a share–memory multiprocessor, with global state available to all simulation threads, and thus is not faced with similar problems.

Another potential problem is the possibility that a remote endpoint (a *TCP* server application for example) is defined and managed in a separate address space, which means that a pointer to any remote object may not be available. To address this issue, *GTNetS* always identifies remote connection endpoints by *IP Address* and port number. The remote endpoint can be represented in the same simulation process as the local endpoint, or by any other process in the distributed simulation. The actions to create a connection are identical in both cases. Again, this solution is similar to that used by *pdns*, and is not an issue with the global–state, shared–memory design of *SSFNet*.

Another concern in distributed network simulations is route calculations. A common approach to routing in a network simulator is to use global topology knowledge to calculate a priori the best path from any node to all other nodes. However, consider a single large topology of n nodes that is distributed on k simulation processes. The topology is split into k sub–models (in a fashion identical to *pdns*), and each sub-model is simulated in a separate simulation process, with approximately n/k nodes modeled and simulated in each of the k processes. With this approach, each simulation may not have sufficient information to make routing decisions at the links spanning the sub–model boundaries. For example, router $R0$ in simulator A above does not have sufficient information to determine which of the two remote links is the appropriate link to forward packets destined to host $H2$.

The simplest approach to solving this problem in to include information in the simulation script that describes which set of IP addresses should be forwarded along which of the remote links. Each remote link is given a list of prefixes that can be reached using this link, and the routing algorithms of each simulator include that information when calculating routes.

A second approach is the use of *Ghost Nodes*. With the ghost node approach, the remaining $n - n/k$ nodes are also modeled on each of the simulation processes, along with information about the links connecting those nodes to others. These ghost nodes do in fact use memory in each simulation process, but the ghost is a reduced state object, containing only connectivity information used for routing decisions but none of the other objects normally associated with full-state nodes. [1] By using *Ghost Nodes*, each simulator has a complete picture of the overall topology of the simulated network and can make routing decisions, but only has full state representations for those nodes mapped to that simulator in the distributed simulation.

A scalable simulator must also have fine grained control over logging of simulation events. When running large–scale simulations, the total number of simulation events processed can be excessively large, potentially trillions or more. Given this large number of events, it is not practical to log a complete record (on disk files) of all simulation events for post–analysis. With *GTNetS*, the events that are logged are completely controlled by the simulation script, with very fine grain control. For example, the simulation program can request that log entries be created only for layer 4 protocols at specific nodes in the topology. Further, all logging can optionally be disabled at selected nodes (such as interior routing nodes). For a given protocol layer, the logging of individual data items can be turned on or off. For example, we can specify that for any TCP header that is logged, the sequence numbers are to be logged, but the checksum field is not. Finally, an individual flow can be tagged with a *forced logging* flag, that would cause every packet in this flow to be logged, even if logging is disabled along the path.

GTNetS uses NIx-Vector routing [13, 14] as the default packet routing mechanism. The NIx-Vector approach does not calculate an all–pairs shortest–path–first graph and does not create routing tables. Instead, the routes are calculated on demand, and cached using a compact representation called a NIx-Vector. While this is not in fact the way existing networks are designed, the resulting savings in simulator memory is believed to be a beneficial tradeoff. For those simulation experiments that do require the existence and maintenance of routing tables, a routing–table based routing method is also included.

To assist with creating large–scale simulation experiments *GTNetS* has a single object that creates a random topology based on the existing *Georgia Tech Internet Topology Modeler* (*GTITM*)[18]. When the *GTITM* object is constructed, parameters are passed specifying the desired average degree, average leaf counts, and average transit node counts. Once the *GTITM* object is created, member

[1] As of this writing, the Ghost Node implementation is work in progress in *GTNetS*.

functions for the *GTITM* object allow for querying node counts, subnetwork counts, leaf node identifiers, and other information about the random topology that can be used to create and manage the simulation.

2.2 Extensibility and Ease of Use

The simulator is written entirely in C++ using an object–oriented design methodology. To use the simulator, the simulationist creates a C++ main program, instantiating C++ objects to represent the various network elements comprising the simulation. Most of the supplied C++ objects that encapsulate the functionality of network elements use *virtual* functions, to allow easy extension and modification of behavior. For example, there is a single virtual base class describing the behavior of a Queue. All queuing methods, such as *DropTail* and *RED*, use a subclass of queue to define their behavior. As another example, the basic functionality of the *TCP* protocol is found an abstract base *TCP* class (called, not surprisingly, *TCP*). Each of the *TCP* variants uses this as a base class, and simply redefines the desired behavior by overriding the necessary methods. With this approach, the class that implements *TCP Reno* is only about 100 lines of code. A similar object oriented approach was used in the *OMNet*[16, 17] simulator, which is a general purpose simulation environment that can be adapted to create network simulations. The *ns2* simulator is designed with a mixture of *Tcl*, *otcl* and *C*. *SSFNet* is written entirely in *Java*, but the model descriptions are specified using a non–standard text–based description language.

The simulator is designed like real networks are designed. In *GTNetS*, there is a clear distinction between nodes, interfaces, links, and protocols. *Node* objects represent the basic functionality of a network node (either a router or end–user system), and contain one or more *Interface* objects. Each interface object has an *IP Address* and associated network mask, as well as a *Link* object encapsulating the behavior of the transmission medium. Packets in *GTNetS* consist of a list of *Protocol Data Unit* objects (*PDUs*). This list is created and extended while a packet moves down the protocol stack through the various layers. When moving up the stack, each protocol layer removes and processes the corresponding protocol header in a fashion closely modeling a real protocol stack. Each protocol layer communicates with the layer below it by invoking a *DataRequest* method, specifying the packet (and current state of the PDU stack), and any protocol specific information required by the next lower layer. Similarly, protocols accept upcalls from the layer below using a *DataIndication* method. Layer 4 endpoints are bound to port numbers, either well know fixed values or transient ports, just like real layer 4 endpoints. Connections between layer 4 endpoints are by IP Address and Port Number, in a fashion nearly identical to actual protocols. In general, when faced with a design decision for the simulator, a design similar to actual networks was chosen whenever possible. Thus, any user who has a good understanding of the design and operation of real networks will find that *GTNetS* works similarly.

Simulation models for a number of different random number generators are provided, including *exponential*, *pareto*, *normal*, *uniform*, *empirical*, and *constant*. We have found the use of a *Constant* random number generator particularly useful. The *Constant* RNG object returns the same constant value every time a new value is requested. The constant value is specified when the *Constant* RNG is created. This *Constant* RNG can be passed to any object needing a random variable, such as the on or off time for an ON–OFF data source. By passing this *Constant* RNG to the ON–OFF data source, it becomes a deterministic data source with on and off times exactly the same every time. A large number of the *GTNetS* objects use random number generator objects during initialization, and the use of the *Constant* RNG allows deterministic behavior where desired.

GTNetS allows the specification of *Default* object types wherever practical. For nearly every simulation object (links, queues, protocols, etc.), a default value is provided that allows for creation of this object without specifying details. For example, by specifying the default queue object is a *DropTail* queue with a queue limit of 60,000 bytes, the user would never need to specify what type of queue (and what size) would be needed for *Interface* objects. If not specified, the default object is used. Similarly, the user can specify the default TCP object is TCP Reno, with a window size of 64,000 bytes. Then whenever an application object (such as a Web browser) needs a TCP endpoint, the default will be used. All default values can be specified by the user, and all default values can be overridden on an instance by instance basis.

The simulator provides a number of *stock objects* for creating well–known topologies, such as *star*, *dumbbell*, *grid*, and *tree*. By using these stock objects, a single line of code can create a dumbbell topology with a random number of nodes on each end, and a specified bandwidth restriction at the bottleneck. Once the dumbbell object is created, member functions can query the number of leaf nodes, and return *Node* object pointers to specific nodes in the dumbbell. Similar capabilities could be included in the *ns2* simulator by addition of specialized objects representing collections of nodes, but this is not presently included in *ns2*.

Finally, *GTNetS* keeps and optionally reports detailed statistics about the simulator's performance. These statistics include the number of objects created, number of simulator events, memory used, just to name a few, which assist the simulator user in identifying resource limitations or performance problems should they occur.

2.3 Support for Popular Protocols

GTNetS includes simulation models of a number of popular protocols at the application layer, transport layer, network layer, and link layer, as discussed below. A protocol graph is used to map protocol numbers to protocol objects, in a fashion similar to actual protocol stacks. At initialization time, the IPv4 implementation registers the use of protocol number 0x800 at layer 3. When a packet is later received at any layer 2 protocol, the L3 protocol number is extracted from the L2 header, and the the layer 3 protocol number is looked up in the protocol graph. A pointer to the IPv4 instance is returned (for protocol number 0x800). Layer 4 works similarly, with TCP registering protocol number 6 and UDP registering number 17.

2.3.1 Application Layer Models

The application models in *GTNetS* use an interface very similar to the familiar *sockets* interface to create and manage layer 4 connections. There are equivalent functions to the familiar `connect`, `listen`, `send`, `sendto`, and `close` just to name a few. The major difference between our implementation and the *sockets* API is the use of upcalls for received data, rather than the more familiar blocking read calls from the socket library. In a discrete event simulation environment, we cannot easily implement the behavior of blocking system calls, and thus we use upcalls to achieve similar results. However, with *GTNetS*, applications do receive notifications of incoming connection requests, connection refusals from peers, connection closure from peers, and failed connections.

The web browser application in *GTNetS* is based on empirical models reported by Mah in [7]. These empirical distributions are used to determine the number of objects per web page, size of the

requests and responses, and the think time between page requests. Our models also include detailed data collection of response time per object, total number of objects, and total size of objects. The web server model allows the enforcement of a limit on the number of simultaneous connections processed, which provides a basis for simulated denial of service style attacks.

For studying the behavior of popular Peer–to–Peer overlay networks, we have simulation models for the behavior of the Gnutella protocol [1] and the *GCache* [2] web server scripts. Our *GCache* model includes the querying of peer *IP Address*es from the cache, the posting of new *IP Address*es, querying of other *GCache* hosts, and the posting of new *GCache* hosts. The Gnutella models include the querying of the *GCaches* for initial peer selection, connection to peers (with both successful and unsuccessful connections), and peers terminating connections. Using these models, large–scale studies of Gnutella client initialization, peer discovery, peer selection, and content searching can be modeled in detail.

Finally, the simulator has models for the well–known *Syn–Flood* and *UDP Storm* distributed denial of service attacks. Our TCP server application models track the number of simulataneous connections, and enforce a limit on ths count to model the behavior of servers under this type of attack. When connections are refused, a *RST* packet is returned to the requestor, who in turn retries the connection after a delay period. Using these models, detailed study of the affect of this type of DDoS attacks can be performed, under a variety of conditions such as the number of attackers, frequency of attack, duration of attacks, etc.

2.3.2 Transport Layer Protocols

The *GTNetS* simulator has models for TCP Reno, TCP NewReno, TCP Tahoe, and TCP SACK. In addition, the design of the TCP model uses a client/server paradigm identical to real TCP implementations. To create a web server for example, a single TCP endpoint is assigned to a node, bound to port 80. Connection requests (SYN packets) received by this endpoint cause the creation of a new TCP endpoint which responds to the connection request, and the original TCP on port 80 continues to listen for SYN packets. This is especially important for distributed simulations, since the client and server may be on different processes, and will have no direct way of communicating that a new server endpoint is needed. With our approach, we simply define the server on a well–known *IP Address* and port, and any other endpoints can connect without further action. To contrast this with the TCP client/server model found in *ns2*, the *ns2* model requires the simulation script to manually create both endpoints of each connection before the connection establishment process is initiated. This can cause difficulties in a distributed simulation where the two endpoints are modeled in different simulator instances.

The simulator has detailed models for UDP datagram processing, and several applications that generate data for UDP flows. These data models include On–Off sources (with configurable probability distributions for the On and Off times) and Constant Bit Rate data sources.

GTNetS has full support for modeling *data contents* as well as data length when moving data between layer 4 protocols. Most current simulators, including *ns2*, do not provide an easy way for applications to specify and receive data contents. Modeling data contents is essential for a number of networking research simulation experiments, including the Peer–to–Peer network models discussed above, as well as the behavior of routing protocols.

2.3.3 Network Layer Protocols

GTNetS uses IPV4 exclusively for the layer 3 protocol. There are presently no models for IPV6, but the basic design of the simulator with a protocol graph and protocol numbers in the layer 2 header make the addition of these model possible at a future time.

For routing protocols, we presently have DSR for wireless route discoveries, with AODV in progress. Further, we are presently working on models for BGP[11] and Cisco's EIGRP protocols for wired networks.

2.3.4 Link Layer Protocols

Each interface in the simulator has an associated MAC address and layer 2 protocol assigned, and these protocols create and utilize an appropriate layer 2 PDU in the simulated packets. We presently have layer 2 models for IEEE 802-3 [6] for wired networks and IEEE 802.11 [5] for wireless. Support is included for link layer broadcasts for Ethernet LAN segments.

Further, each interface has an associated queue object, which is used to store packets to be transmitted when the link is available. We presently have implemented models for simple DropTail queues as well as the Random Early Detection (*RED*)[4] queues.

Nodes with wireless interfaces also have mobility models, and support random initial node placement. A variety of placement distributions are available, including uniform, bounded normal, and bounded exponential. The mobility is based on a random waypoint approach, but allows for the specification of specific predetermined waypoints as well.

2.4 Built in Data Collection

GTNetS has a number of data summarization primitives, to assist the user in gathering network performance statistics during the simulation execution. For example, the Web Browser object has an optional *histogram* object that is used to trace the response time for each requested web object. These histogram objects can then be queried and printed, resulting in a cumulative distribution graph (CDF) of the web response time.

Another example of the built in data collection methods is the optional logging of sequence number versus time in a *TCP* connection. If instructed to do so in the simulation program, any specified *TCP* object will track the sequence numbers sent and acknowledged as a function of simulatin time, and will log this information to a data file at the conclusion of the simulation. This can then be plotted (for example with *GnuPlot*), to produce a graphical representation of the behavior of the *TCP* connection. The *ns2* simulator can be used to produce a similar plot, but post–analysis (with *AWK* or other scripting languages) of the trace file is necessary to achieve this.

3. USING GTNetS

The *GTNetS* simulator consists of a large number of *C++* objects which implement the behavior of a variety of network elements. Building and running a simulation using *GTNetS*, consists of creating a *C++* main program that instantiates the various network elements to describe a simulated topology, and the various applications and protocols used to move simulated data through the topology. The *C++* main program is then compiled with any compiler that fully complies with the *C++* standard[2]. After successfully compiling the main program, it is linked with the *GTNetS* object libraries, which are available both *.a* and *.so* format. The resulting executable binary is simply executed as any other application, which results in the simulation of the topology and data

[2] *GTNetS* has been compiled successfully on Linux with g++-2.96, g++-3.x, Sun Solaris with SUNWS-CC and HPUX-CC (64 bit)

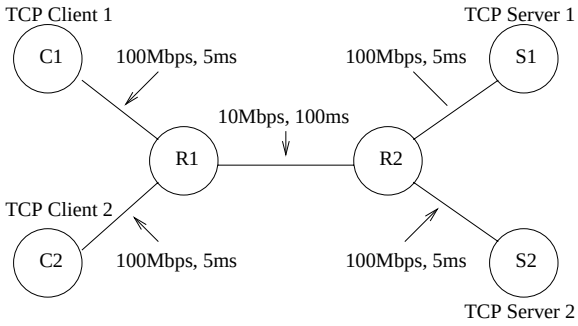

Figure 2: Sample GTNetS Topology

flows specified in the main program.

In this section, we give a small example of using *GTNetS* to create a simple topology and two data flows. An example *GTNetS* simulation is given for the simple topology shown in figure 2, and discussed in detail. In this simulation, there are six nodes, and two TCP flows from clients to servers. Many of the *GTNetS* features will be used in this example, most of which were described in the prior section. The main program for the example is shown in the listing on the following page. Each line of the sample program will be discussed briefly to describe it's purpose in the simulation. After reading through this example, the reader should have a basic understanding of the various functions provided by *GTNetS* and how to use them.

Include Files. Lines 4 – 10 use the C/*C++* `include` directive to include the definitions for the various network elements and simulation objects used in this simulation. The necessary include files will of course vary from simulation to simulation depending on which of the *GTNetS* objects are used in the simulation. A complete listing of all *GTNetS* objects and their corresponding include files is given in the *GTNetS* reference manual.

Main Program. The C++ `main` entry point is defined in line 12. All *GTNetS* simulations must have a *C++* `main` function.

Simulator Object. A single object of class `Simulator` must be created by all *GTNetS* simulations before any other *GTNetS* objects are created. In our example, the `Simulator` object is created at line 15.

Defining the Trace File. Lines 17 – 22 specify the name of the trace file and the desired level of tracing. For all *GTNetS* simulations, there is a single global object of type `Trace` that manages all aspect of packet tracing in the simulation. Line 18 uses the `Trace::Instance()` function to obtain a pointer to the global trace object. Line 19 specifies that all *IP Address*s should be written to the trace file in *dotted* notation, such as 192.168.0.1. The default notation for logging *IP Address*s is 32 bit hexadecimal. Line 20 opens the actual trace file and assigns the name `intro1.txt`. Line 21 specifies that the flags field for all *TCP* headers logged in the trace file should use a text based representation for the flags (such as `SYN|ACK`), rather than a 8 bit hexadecimal value. Line 22 specifies that all *IPV4* headers should be logged in the trace file for every packet received and every packet transmitted at all nodes.

Create Simulated Nodes. Lines 24 – 30 create the node objects representing the six network nodes in the sample topology. In *GTNetS*, node objects represent either end–systems (such as desktop systems or web servers), routers, or hubs. Notice that when creating `Node` objects, the *C++* `new` operator is used to create then nodes, rather than using a statically defined object (for example by saying `Node c1; Node c2;`). This is due to the fact that node objects (and in fact almost all of the topology objects) must exist for the life of the simulation, and must not be destroyed until the simulation completes. In this simple example, either method would work correctly, since the nodes are being defined inside the `main` function, which does not exit until the simulation completes. However, if the nodes are created in a subroutine (such as `int CreateNodes()`, they would not persist after the `CreateNodes` function completed unless dynamically allocated with `new`.

Create Simulated Links. Lines 32 – 43 create the five link objects in the sample topology. First, line 33 creates a point–to–point link object of class `Linkp2p`. There are three things of interest in this declaration. First are two arguments to the constructor for `Linkp2p` objects, which specify the link bandwidth and propagation delay. The arguments are of type `Rate_t` and `Time_t` respectively, which are both of type `double`. However, in *GTNetS*, anytime a variable of type `Rate_t` is required, an object of class `Rate` may be used instead. Objects of class `Rate` require a single argument in the constructor, which is a string value specifying rates using commonly recognized abbreviations for multipliers (such as `Mb`). Similarly, anytime a variable of type `Time_t` is required, an object of class `Time` may be used instead. Objects of class `Time` require a single argument in the constructor, which is a string value specifying rates using commonly recognized abbreviations for multipliers (such as `ms`).

Secondly, notice that the object `l` is statically allocated in this case (it is not allocated using the `new` operator), and will be destroyed when the enclosing subroutine exits. This is the accepted way of defining and parameterizing links, and will be discussed in more detail below. Lines 35 – 38 specify the links connecting the clients to router `r1` and the servers to router `r2`. `Node` objects have a method `AddDuplexLink` which creates links between nodes. In this example, there are three arguments to `AddDuplexLink`, the `Node` pointer for the opposite link endpoint, the link object itself (`l` in this case), and an *IP Address* for the link endpoint. It is important to note that the `AddDuplexLink` method makes a *copy* of the `Link` object passed as the second parameter, rather than using it directly. Thus a single link object (`l` in this example) can be passed to any number of `AddDuplexLink` calls, and can subsequently be destroyed (when the subroutine exits) without causing problems in the simulation.

Finally notice that the third argument to `AddDuplexLink` in this example is the *IP Address* of the local link endpoint. The argument must be of type `IPAddr_t`, which is defined by *GTNetS* to be `unsigned long`. However, in *GTNetS*, whenever a variable of type `IPAddr_t` is needed, an object of class `IPAddr` may be used instead. Objects of class `IPAddr` require a single argument in the constructor, specifying the desired *IP Address* in the familiar dotted notation. Lines 41 – 43 create the slower speed 10Mb link object connecting the two routers, `r1` and `r2`.

Create the TCP Servers. The *TCP* server objects are created in lines 45 – 51. Lines 46 and 47 create two objects of class `TCPServer`, using the `new` operator. The `new` operator is used for reasons discussed above in the `Node` objects creation paragraph. The constructor for `TCPServer` objects has a single parameter of class `TCP`, which specifies the variant of *TCP* to be used for this server object. Notice that, in this example, an anonymous temporary object `TCPTahoe()` is passed as the argument to the `TCPServer` constructor. The constructor for `TCPServer` makes a copy of the supplied object, rather than using it directly. The anonymous temporary `TCPTahoe` object is destroyed when the constructors at lines 46 and 47 are complete. Lines 48 and 49 assign the `TCPServer` objects to nodes `s1` and `s2` respectively, and bind to port 80. Lines 50 and 51 specify that all *TCP* headers should be logged in the trace file for all received and transmitted packets

```
1  // Simple GTNetS example
2  // George F. Riley, Georgia Tech, Winter 2002
3
4  #include "simulator.h"      // Definitions for the Simulator Object
5  #include "node.h"           // Definitions for the Node Object
6  #include "linkp2p.h"        // Definitions for point-to-point link objects
7  #include "ratetimeparse.h"  // Definitions for Rate and Time objects
8  #include "application-tcpserver.h" // Definitions for TCPServer application
9  #include "application-tcpsend.h"   // Definitions for TCP Sending application
10 #include "tcp-tahoe.h"      // Definitions for TCP Tahoe
11
12 int main()
13 {
14   // Create the simulator object
15   Simulator s;
16
17   // Create and enable IP packet tracing
18   Trace* tr = Trace::Instance(); // Get a pointer to global trace object
19   tr->IPDotted(true);            // Trace IP addresses in dotted notation
20   tr->Open("intro1.txt");        // Create the trace file
21   TCP::LogFlagsText(true);       // Log TCP flags in text mode
22   IPV4::Instance()->SetTrace(Trace::ENABLED);  // Enable IP tracing all nodes
23
24   // Create the nodes
25   Node* c1 = new Node();    // Client node 1
26   Node* c2 = new Node();    // Client node 2
27   Node* r1 = new Node();    // Router node 1
28   Node* r2 = new Node();    // Router node 2
29   Node* s1 = new Node();    // Server node 1
30   Node* s2 = new Node();    // Server node 2
31
32   // Create a link object template, 100Mb bandwidth, 5ms delay
33   Linkp2p l(Rate("100Mb"), Time("5ms"));
34   // Add the links to client and server leaf nodes
35   c1->AddDuplexLink(r1, l, IPAddr("192.168.0.1")); // c1 to r1
36   c2->AddDuplexLink(r1, l, IPAddr("192.168.0.2")); // c2 to r1
37   s1->AddDuplexLink(r2, l, IPAddr("192.168.1.1")); // s1 to r2
38   s2->AddDuplexLink(r2, l, IPAddr("192.168.1.2")); // s2 to r2
39
40   // Create a link object template, 10Mb bandwidth, 100ms delay
41   Linkp2p r(Rate("10Mb"), Time("100ms"));
42   // Add the router to router link
43   r1->AddDuplexLink(r2, r);
44
45   // Create the TCP Servers
46   TCPServer* server1 = new TCPServer(TCPTahoe());
47   TCPServer* server2 = new TCPServer(TCPTahoe());
48   server1->BindAndListen(s1, 80); // Application on s1, port 80
49   server2->BindAndListen(s2, 80); // Application on s2, port 80
50   server1->SetTrace(Trace::ENABLED); // Trace TCP actions at server1
51   server2->SetTrace(Trace::ENABLED); // Trace TCP actions at server2
52
53   // Create the TCP Sending Applications
54   TCPSend* client1 = new TCPSend(TCPTahoe(c1),
55                                  s1->GetIPAddr(), 80,
56                                  Uniform(1000,10000));
57   TCPSend* client2 = new TCPSend(TCPTahoe(c2),
58                                  s2->GetIPAddr(), 80,
59                                  Constant(100000));
60   // Enable TCP trace for all clients
61   client1->SetTrace(Trace::ENABLED);
62   client2->SetTrace(Trace::ENABLED);
63
64   // Set random starting times for the applications
65   Uniform startRv(0.0, 2.0);
66   client1->Start(startRv.Value());
67   client2->Start(startRv.Value());
68
69   s.Progress(1.0);               // Request progress messages
70   s.StopAt(10.0);                // Stop the simulation at time 10.0
71   s.Run();                       // Run the simulation
72   cout << "Simulation Complete" << endl;
73 }
```

for these *TCP* endpoints.

Create the TCP Clients. The *TCP* client applications are created in lines 54 – 62. The objects of class `TCPSend` are created using the `new` operator, as previously discussed. `TCPSend` constructors have four parameters, as follows. First is a temporary object of class `TCP` (or any subclass of `TCP`) that specifies the *TCP* variation to be used for this *TCP* client. Notice that in this case, the constructor for the temporary *TCP* object specifies the node object that is associated with the corresponding *TCP* endpoint (`c1` in line 54, and `c2` in line 57). The second and third arguments are the *IP Address* and port number of the *TCP* server to which a connection is made. The second argument in this example illustrates the `GetIPAddr` method for `Node` objects, which returns the *IP Address* of the node (or the first *IP Address* if the node has more than one). The last parameter is a temporary object of class `Random` (or any subclass of `Random`), which specifies a random variable that determines how much data to send to the server. The first client specifies a `Uniform` random variable at line 56 which returns a uniform value between 1000 and 10,000. The second client specifies a `Constant` random variable, that returns the constant value 100,000. Lines 60 – 62 enable tracing of the *TCP* headers for all packets sent and received by these endpoints.

Start the Client Applications. Lines 64 – 67 tell the simulator to create the connections between the clients and servers. Line 65 defines a statically allocated `Uniform` random variable that will return random values uniformly in the interval [0.0, 2.0). Lines 66 and 67 use the `Start` method common to all *TCP* applications that specifies when the application should create the connection and begin sending the data.

Start the Simulation. Line 69 uses the `Progress` method of class `Simulator` to request a message be printed on `stdout` every 1.0 seconds of simulation time, indicating the simulation is progressing in time. Line 70 calls the `Run` method of `Simulator`, to run the simulation. The `Run` method does not exit until the simulation completes.

4. SCALABILITY EXPERIMENTS

Since the *Georgia Tech Network Simulator* was designed to support large–scale experiments, we set out to determine how large of a network topology we could create and successfully simulate. Using our account on the Pittsburgh Supercomputer Center (PSC), we created a scalable simulation based on the *Campus Network* (CN) topology defined by Nicol[9], as shown in figure 3. Each CN consists of 538 nodes, including 504 leaf client nodes and 4 server nodes. An arbitrary number of campus networks can be connected together with high–speed links between the gateway router nodes of each campus. For data flows in these experiments, each leaf on a campus network randomly selects a server on an adjacent CN, and establishes a TCP connection with it. Each connection sends a total of 500,000 bytes and terminates. By distributing a number of campus networks on each of a number of *GTNetS* instances, the total scale of the network and amount of data processed can grow arbitrarily large.

Some preliminary results of our scalability experiments are given in table 1. The Pittsburgh Supercomputer Center system consists of 750 systems, each with four HP Alpha CPU's and 4Gb of main memory. The systems are connected via a Quadrix high–speed interconnect. We assigned four simulation processes on each PSC system (one per CPU), and varied the number of systems from 1 to 32, giving a total number of simulation processes varying from 4 to 128. As a baseline, we also ran the single CPU case. For these results, each simulation instance modeled a total of seven campus networks, for a total 3,766 nodes and 3,556 flows per instance. Clearly, as the number of CPUs assigned to the distributed increases, the total topology size and number of flows modeled increases linearly. As can be seen from the table, we have to date completed a simulation of 482,048 nodes and 455,168 TCP flows, which processed more than 4 billion simulated packet transmission events. Each of these simulations completed in less than 15 minutes, including all initialization overhead. Further, the table shows that *GTNetS* shows good scalability in terms of performance as the number of simulation instances increases. The simulation running on 128 CPUs does take longer than the single CPU case (869 seconds versus 562 seconds), but recall that it is modeling 128 times as many flows and packets. The difference in execution time is due to the time synchronization and message passing overhead between the simulation instances in the distributed simulation environment.

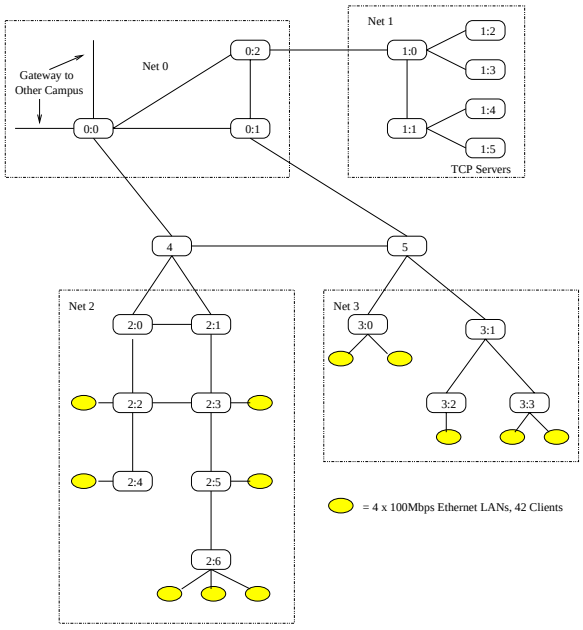

Figure 3: Campus Network Topology

A pencil and paper analysis of the memory usage shows that, within the memory constraints of a single system at PSC, we should be able to support four processes of 20 CN's per processor on a single PSC system. Given this, we fully expect simulation topologies exceeding several million nodes to be completed in the near future.

We point out that, while we did in fact set out to design and implement an efficient simulation environment, our objective was not to create a tool where performance is the only important metric. The performance of a network simulator is a function of a number of variables, and design tradeoffs. For example, *GTNetS* has substantial detail in the layer 2 IEEE 802-3 model, including preparation and processing of a layer 2 PDU as packets are transmitted and received. This results in extra per–packet event processing overhead, as compared to *ns2* and *SSFNet* which have no such models. Further, *GTNetS* checks for local network routes and broadcasts in the layer 3 processing, as well as looking up the layer 4 processor in a protocol graph, which again results in some extra per–packet overhead. Design decisions such as this were made to insure the simulator works much like real networks work, and to facilitate future addition of protocols such as Address Resolution Protocol (ARP)[10].

Preliminary experiments comparing *ns2* and *GTNetS* show that the simulation initialization time is substantially faster in *GTNetS*,

Number Systems	Number Processors	Simulated Nodes	Simulated Flows	Execution Time	Simulated Pkts
1	1	3766	3556	562 sec	36955410
1	4	15064	14224	697 sec	146014864
2	8	30128	28448	723 sec	291981440
4	16	60256	56896	769 sec	583689248
8	32	120512	113792	753 sec	1167925760
16	64	241024	227584	787 sec	2332761088
32	128	482048	455168	869 sec	4662689280

Table 1: Scalability Experiments from PSC

roughly an order of magnitude faster. This difference is clearly due to the *ns2* design decision to use the interpreted Tcl language to define the simulation, with the resulting ease of use at the expense of slower performance. Once the simulation begins processing events, *ns2* in fact runs somewhat faster than *GTNetS*, due to the extra processing details in the *GTNetS* layer 2 and 3 models. We have not yet done a performance study comparing *GTNetS* with *SSFNet*, but we expect that the performance will be comparable.

5. SUMMARY

The *Georgia Tech Network Simulator* is a full featured network simulation environment that can be used for experimental networking research on moderate to large–scale topologies. The design of *GTNetS* is such that it is easy to learn and use. The object oriented methodology in the design is such that it can be easily extended to support new variations on existing networking methodologies. The simulator is efficient, both in the initialization overhead, and during the actual processing of simulation events. This is evident from the 480,000 node topology simulation at PSC completing in less than 15 minutes.

The scalability experiments presented show nearly a half–million node topology can be simulated on a moderately sized network of workstations (32 workstations, 4 CPU's each). Using more of the available resources at PSC (there are a total of 3,000 procesesors on 750 systems), we fully expect to be demonstrating simulations of millions of nodes in the near future.

As previously mentioned, a tool designed for use by the networking research community will never be complete. New reqirements for network protocols and methods will certainly be needed as new such methods are invented. We are continuing work on our simulator, with students presently working on more wireless routing protocol (AODV and NixVector), and wired routing protocols (BGP and EIGRP).

Our objective is to provide a tool that can be another option in the set of tools available to networking researchers to study network behavior in a simulation environment. We hope that it will be of benefit to the community at large.

6. REFERENCES

[1] The Gnutella protocol specification. Software on-line: http://www.gnutella.com, 2002. Gnutella.

[2] The gwebcache specification. Software on-line: http://www.gnucleus.com/gwebcache/specs.html, 2002. Gnucleus.

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

[4] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE Transactions on Networking*, 1(4):397–413, August 1993.

[5] IEEE. Ieee standard 802-11 wireless lan medium access control (mac) and physical layer (phy) specification. *Institute of Electrical and Electronic Engineers*, 1997.

[6] IEEE. 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*, 2000.

[7] B. A. Mah. An empirical model of http network traffic. In *Proceedings of IEEE INFOCOMM*, pages 592–600, 1997.

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

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

[10] D. Plummer. Internet RFC826: Ethernet address resolution protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. Network Working Group, Nov 1982.

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

[12] G. F. Riley. The georgia tech network simulator. Software on-line: http://www.ece.gatech.edu/research/labs/MANIACS/gtnets.htm, 2003.

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

[14] G. F. Riley, M. H. Ammar, and E. W. Zegura. Efficient routing using nix-vectors. In *2001 IEEE Workshop on High Performance Switching and Routing*, May 2001.

[15] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. 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)*, October 1999.

[16] A. Varga. The OMNeT++ distrete event simulation system. Software on-line: http://whale.hit.bme.hu/omnetpp/, 1999.

[17] A. Varga. Using the omnet++ discrete event simulation system in education. *IEEE Transactions on Education*, 42(4), Nov 1999.

[18] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE Infocom 96*, 1996.