Realistic TCP Traffic Generation in *ns*-2 and *GTNetS*

Prashanth Adurthi and Michele C. Weigle
padurth@cs.clemson.edu, mcweigle@acm.org
Department of Computer Science
Clemson University
Clemson, SC 29634

***Abstract*— Network simulations have become a large part of how researchers evaluate their new proposals for Internet protocols. One of the problems is the lack of good models for how the Internet behaves. This problem is magnified by the fact that the Internet is always changing, with new applications producing unexpected traffic patterns. Recently, a new method for quickly modeling Internet traffic has been developed. The *a-b-t* model, developed at the University of North Carolina, can characterize all of the TCP connections on a particular link without having to know precisely the applications that appear on the link. The *tmix* workload generation tool, which can replay connections described by the *a-b-t* model, was initially developed for use in network testbeds. We have implemented this tool in the *ns*-2 and *GTNetS* simulators. In this paper, we describe the implementation of *tmix* in *ns*-2 and *GTNetS*, compare the two implementations, and show that both implementations can produce traffic that is statistically similar to the original traced traffic.**

I. Introduction

Network simulations have become a large part of how researchers evaluate their new proposals for Internet protocols. One of the often-stated problems is the lack of good models for how the Internet behaves [8], [9]. The Internet is always changing, and new applications are developed that produce unexpected traffic patterns and side-effects. Many of the traffic models that researchers have used in the past, and in fact are still using, were developed in a time before some of the newer high-impact applications were in widespread use (*e.g.*, peer-to-peer file sharing). One of the reasons for this lag is the tremendous effort and amount of time required to develop statistical models of the traffic generated by a particular application. Unfortunately, it often seems that as soon as there is an agreed-upon model for an application traffic type, some new application or new method for using the well-known application (*e.g.*, the use of HTTP for peer-to-peer file-sharing) becomes widely-used.

Recently, a new method for quickly modeling Internet traffic has been developed. The *a-b-t* model [10], [11] can characterize all of the TCP connections on a particular link without having to know precisely the applications that appear on the link. This method models application data units (ADUs) rather than individual packets in order to produce a network-independent, source-level characterization of traffic. The main insight in the *a-b-t* model is that most TCP-based applications operate in a request-response manner. These requests and responses are the ADUs. The size of the ADU, or request, sent by the connection *initiator* is represented by a. The size of the ADU, or response, sent by the connection *acceptor* is represented by b. The time between an initiator receiving a response and sending the next request, or think time, is represented by t_a, and the time between an acceptor receiving a request and sending the response, or server delay, is represented by t_b. Each set of values (a_i, b_i, t_{a_i}, t_{b_i}) is called an epoch, E_i. An entire connection can be represented by a *connection vector*, C_i, which consists of a set of epochs (E_1, E_2, ..., E_n) from the same TCP connection. To allow for applications that do not strictly follow the request-response method, any of the a or b sizes can be 0.

Most connections can be described in the manner outlined above and are called *sequential connections*. Another form of connection, called a *concurrent connection*, allows for deviation from the sequential pattern. Examples of this type of application protocol include HTTP/1.1 (pipelining) and the BitTorrent file-sharing protocol. For a concurrent connection, the t values represent the time between an initiator sending consecutive a-type ADUs or an acceptor sending consecutive b-type ADUs. There is no time dependence in the connection vector between the a-type and b-type ADUs.

The *tmix* workload generation tool [10], [17], which can replay *a-b-t* model connection vectors, was initially developed for use in network testbeds. We have implemented this tool in the *ns-2* [3] and *GTNetS* [14] simulators. A validation of the *ns*-2 implementation was presented in previous work [17]. In this paper, we describe the implementation of *tmix* in *ns*-2 and *GTNetS*, compare the two implementations, and show that both implementations can produce traffic that is statistically similar to the original traced traffic.

Our goal is to provide users of both *ns*-2 and *GTNetS* with a tool that allows them to perform high-quality network simulations that include realistic Internet traffic. All a user needs to replay traffic from a certain link is the file of connection vectors obtained from a trace of that link and the *tmix* tool. We plan to host a repository for a set of connection vectors representing different types of Internet links for users to download. A release of the tool to generate a connection vector file from a `tcpdump` trace will be coming in the near future.

This remainder of this paper is organized as follows. Section II discusses related work including existing traffic generators for *ns*-2 and *GTNetS*. Section III describes the *tmix* connection

vector file format, as well as changes we have made to the format. Section IV discusses the actual implementation of *tmix* in both *ns*-2 and *GTNetS*. Section V discusses our validation of the implementation of *tmix*, and we conclude with a summary and future directions in Section VI.

II. Related Work

There are several existing traffic generators either built-in to or externally contributed to *ns*-2. One of the first to be included in *ns*-2 was the *WebTraf* module, based on the work of Feldmann *et al.* [7]. This work models application-specific characteristics of HTTP traffic such as request size, response size, user think time, objects per page, inter-object time, *etc.* Each characteristic is driven by a random variable distribution, the parameters of which were derived from analyzing packet traces.

The *RAMP* tool [12], also included in *ns*-2, can take a `tcpdump` trace and generate cumulative distribution functions (CDFs) that describe FTP transfers and web traffic. These empirical CDFs can then be used to generate traffic.

A newer traffic generator recently added to *ns*-2 is the *PackMime-HTTP* module. This generator is based on a web traffic model developed at Bell Labs [5] which includes persistent HTTP connections. This connection-based model was developed from and validated against a large packet trace collected in 2000.

One externally contributed module is *nsweb* [15]. This is an extension of the well-known SURGE web traffic model [2] and includes both pipelining and persistent connections.

The *GTNetS* simulator has built-in web server and web browser applications based on the Mah web traffic model [13], but to our knowledge, there have been no other TCP traffic generators contributed to *GTNetS*.

Although there are existing web traffic generators for network simulators, there is no tool for generating realistic, application-independent mixes of all TCP traffic on a link, including email, FTP transfers, web browsing, and peer-to-peer file sharing.

III. *Tmix* Connection Vector Format

The *a-b-t* model describes each connection in a trace as a *connection vector*. A connection vector consists of the set of ADUs sent by both the initiator and acceptor sides of the connection and the time delays between each side receiving the data from the other side and sending its own data. In addition to these values, the connection vector also specifies other characteristics of the connection, including connection start time, the window sizes of the initiator and the acceptor, the minimum RTT that the packets experienced, and the loss rate of that connection. A file of connection vectors is read in by the *tmix* tool and used to generate the corresponding traffic.

A. Original Connection Vector Format

Examples of the original format of connection vectors as described in [17] are shown in Figures 1 and 2. Figure 1 shows a sequential connection vector, while Figure 2 shows a concurrent connection vector. Sequential and concurrent

```
SEQ 6851 1 21217 555382  # starts at 6.851 ms
w 64800 6432             # win sz (bytes): init acc
r 1176194                # min RTT (us)
l 0.000000 0.000000      # loss: init->acc acc->init
> 245                    # init sends 245 bytes
t 51257                  # acc waits 51 ms after recv
< 510                    # acc sends 510 bytes
t 6304943                # init waits 6.3 sec after
                         #   send and then sends FIN
```

Fig. 1. Sequential Connection Vector Example. In the comments, we abbreviate the initiator as `init` and the acceptor as `acc`.

```
CONC 1429381 2 2 26876 793318 # starts at 1.4 s
w 65535 5840                  # win sz (bytes)
r 36556                       # min RTT
l 0.000000 0.000000           # loss rate
c> 222                        # init sends 222 bytes
t> 62436302                   # init waits 62 sec
c< 726                        # acc sends 726 bytes
t< 62400173                   # acc waits 62 sec
c> 16                         # init sends 16 bytes
t> 725                        # init waits 725 us
                              #   and then sends FIN
c< 84                         # acc sends 84 bytes
t< 130                        # acc waits 130 us
                              #   and then sends FIN
```

Fig. 2. Concurrent Connection Vector Example. In the comments, we abbreviate the initiator as `init` and the acceptor as `acc`.

connection vectors are differentiated by the starting string in the first line: `SEQ` for a sequential connection and `CONC` for a concurrent connection. The second line in each, starting with `w` gives the window sizes of the initiator and acceptor, respectively, in bytes. The third line starting with `r` gives the minimum RTT in microseconds between the initiator and acceptor. The fourth line, starting with `l`, shows the loss rates involved in each direction of the connection. The remaining lines in the connection vector show the ADU exchanges.

In the sequential connection vectors, the ADUs are shown in increasing order by the times at which they are sent. The lines starting with $>$ show the sizes of the *a*-type ADUs sent by the initiator to the acceptor, and the lines starting with $<$ show the sizes of the *b*-type ADUs sent by the acceptor to the initiator. Note that there is a time dependency in case of sequential connection vectors. One side of the connection is dependent on the other side of the connection sending it an ADU.

In case of sequential connections, the line containing `t` can appear in any of the following four scenarios:

1) After a line beginning with $>$ and before a line beginning with $<$.
2) After a line beginning with $<$ and before a line beginning with $>$.
3) At the end of the connection vector, after a line beginning with $>$.
4) At the end of the connection vector, after a line begin-

ning with <.

Depending on its placement, the semantics associated with the t value change. In case 1, t denotes the amount of time the acceptor has to wait after receiving an ADU from initiator before it can send its next ADU. In case 2, the t denotes the amount of time the initiator has to wait after receiving an ADU from acceptor before it can send its next ADU. In case 3, the t denotes the time the initiator has to wait after sending its last ADU and before closing the connection. In case 4, the t denotes the time that the acceptor has to wait after sending its last ADU and before closing the connection.

For concurrent connection vectors, lines starting with c> indicate the bytes sent by the initiator (*a*-type ADUs), and lines starting with c< indicate the bytes sent by the acceptor (*b*-type ADUs). Lines starting with t> indicate the time the initiator waits before sending the next ADU (or sending the FIN, if the last ADU has been sent). Likewise with lines beginning with t< and the acceptor. Note that there is no time dependence between the initiator and acceptor in case of a concurrent connection vector. The waiting times are between consecutive sends and are not dependent upon receiving an ADU from the other side.

B. Alternate Connection Vector Format

The multiple possible interpretations of the t value with sequential connections requires us to keep a record of what has been read before the t value is read. This makes parsing the connection vector file tedious. Also, because of the difference between the sequential and concurrent connection semantics, they have to be dealt separately while programming the module. To avoid this and to make programming the module easier, we have modified the connection vector format into an alternate format.

The basic idea behind converting the original connection vector format is that in the case of sequential connection vectors, there really exist two times associated with the initiator or acceptor while sending an ADU to the other side:

1) The amount of time the initiator/acceptor has to wait before sending the next ADU after sending its previous ADU (*send_wait_time*).
2) The amount of time the initiator/acceptor has to wait before sending the next ADU after receiving an ADU from the other side (*recv_wait_time*).

Note that only one of the above two values is used by an initiator/acceptor while sending its ADU to the other side, *i.e.*, the initiator/acceptor schedules sending its next ADU with respect to the event of receiving a ADU from the other side or with respect to the event of sending a previous ADU. The initiator/acceptor does not use both of these values at the same time, so in the new format one of these values is always set to 0. Also note that at the beginning of the connection, the side sending the first ADU will have both *send_wait_time* and *recv_wait_time* set to 0. In case of the t values appearing at the end of a connection vector in the original format, we introduce a dummy ADU with size 0 to represent the FIN that will be sent by the initiator/acceptor that sends the last ADU.

```
S 6851 1 21217 555382     # starts at 6.851 ms
w 64800 6432              # win sz (bytes)
r 1176194                 # min RTT
l 0.000000 0.000000       # loss rate
I 0 0 245                 # init sends 245 bytes
A 0 51257 510             # acc waits 51.257 ms after
                          #   recv then sends 510 bytes
A 6304943 0 0             # acc waits 6.3 sec after
                          #   send then sends FIN
```

Fig. 3. Alternate Sequential Connection Vector Example. In the comments, we abbreviate the initiator as init and the acceptor as acc.

```
C 1429381 2 2 26876 793318  # starts at 1.4 s
w 65535 5840                # win sz (bytes)
r 36556                     # min RTT
l 0.000000 0.000000         # loss rate
I 0 0 222                   # init sends 222 bytes
A 0 0 726                   # acc sends 726 bytes
I 62436302 0 16             # init waits 62 sec and
                            #   then sends 16 bytes
A 62400173 0 84             # acc waits 62 sec and
                            #   then sends 84 bytes
I 725 0 0                   # init waits 725 us
                            #   and then sends FIN
A 130 0 0                   # acc waits 130 us
                            #   and then sends FIN
```

Fig. 4. Alternate Concurrent Connection Vector Example. In the comments, we abbreviate the initiator as init and the acceptor as acc.

This same alternate representation can be used for concurrent connection vectors also. But because there is no time dependence between the sides of the connection, each side schedules sending its next ADU with respect to time at which it sent its last ADU. Therefore, in case of concurrent connection vectors, *recv_wait_time* is not applicable and is always set to 0. Also, in case of concurrent connection vectors, both the sides start sending their messages at the same time.

We keep the header lines of the connection vectors (those containing the start time, window size, RTT, and loss rates) in the same format as the original, except that we replace SEQ with S and CONC with C. Lines beginning with I denote actions for the initiator, and lines beginning with A show actions for the acceptor. The remaining format of these lines is send_wait_time recv_wait_time bytes. Figures 3 and 4 show the alternate connection vector format corresponding to the sequential and concurrent connections shown in Figures 1 and 2, respectively.

With this new connection vector format, sequential and concurrent connections can be handled uniformly. Parsing the file is made easier as the problem with the t value in the original format is eliminated. Instead of changing the original tool that creates connection vectors from tcpdump traces, we wrote a simple Perl script to convert the original *tmix* connection vector format to this new format.

IV. Implementation of *tmix*

The implementation of *tmix* in both *ns*-2 and *GTNetS* is divided into two parts: *tmix-end* and *tmix-net*. The module *tmix-end* implements the work of the end systems in sending ADUs into the network, and the *tmix-net* module implements the network part of the traffic generation (*i.e.*, delays and losses).

In *ns*-2, *tmix-end* is based on the implementation of *PackMime-HTTP* [4], [5], and *tmix-net* is based on the implementation of *DelayBox* [5], [16]. Both *PackMime-HTTP* and *DelayBox* are now included as a standard part of *ns*-2, and further documentation is available in the *ns*-2 Manual [6].

We will first describe the general implementations of *tmix-end* and *tmix-net* (which are also described in our previous work [17]), and then follow with differences between *ns*-2 and *GTNetS* and how they affect the *tmix* implementations. Additional details of both implementations are available in [1].

A. tmix-end

The *tmix-end* module controls all of the activities of a set of initiators and acceptors. There are two nodes associated with each *tmix-end* instance – one representing a set of initiators and another representing a set of acceptors. These nodes are typically on different sides of the network from each other.

tmix-end reads the description of connections from a specified connection vector file, the format of which was described earlier. For each connection described in the file, *tmix-end* maintains a list of ADUs and their associated times (*i.e.*, the *send_wait_time* and the *recv_wait_time*) for the initiator and a separate list for the acceptor. These two lists are associated with a single connection. This connection is also described by the start time of the connection, a unique ID present in the connection vector file, and the maximum TCP window sizes for the initiator and acceptor. Once the initiator and acceptor have a TCP connection established, transfer of ADUs proceeds according to the connection vector. Once all the connections described in the file have completed, *tmix-end* ends the simulation.

New connections are started according to the connection start time. The *tmix-end* module sets the appropriate window sizes for both the initiator and acceptor. The remaining operation of *tmix-end* depends upon whether the connection vector describes a sequential or concurrent connection. In sequential connections, the initiator sends *a*-type ADUs ("requests") that the acceptor "responds to" with a *b*-type ADU. No pipelining of requests is used. In concurrent connections, the initiator uses pipelining to send multiple *a*-type ADUs without waiting for a response from the acceptor.

For sequential connections, the initiator uses the connection's *a* sizes, while the acceptor uses the connections *b* sizes. In most cases, both the initiator and acceptor use the *recv_wait_time*. This time represents the think time for the initiator and the server delay time for the acceptor. The initiator first sends an *a*-type ADU. Once the acceptor receives an *a*-type ADU, it waits for the time specified in *recv_wait_time* and then sends the next *b*-type ADU in the list. When the acceptor receives the *b*-type ADU, it waits for the *recv_wait_time* and then sends the next *a*-type ADU. If the size of the next ADU to send is 0, the node sends a FIN to close the connection.

In the case of concurrent connections, each side (acceptor or initiator) is scheduled independently. The initiator sends its *a*-type ADUs according to the schedule given, waiting for its *send_wait_time* before sending a new *a*-type ADU. The acceptor sends its *b*-type ADUs according to its schedule, waiting for its *send_wait_time* before sending a new *b*-type ADU. The acceptor no longer waits to receive an *a*-type ADU before sending a response, and the initiator no longer waits to receive a *b*-type ADU before sending the next *a*-type ADU. The side sending the last ADU will send the FIN to close the connection.

B. tmix-net

The *tmix-net* module allows a user to create per-flow delays and losses. The delay and loss rates for each connection are contained in the connection vector. A *tmix-net* node should be placed in the network between the initiator and acceptor nodes used by *tmix-end*.

Upon startup, the *tmix-net* module reads each connection's ID, source, destination, RTT, and loss rate from the connection vector file into a table. When a packet is received, *tmix-net* looks up the connection ID, source, and destination in the table to find the appropriate delay and loss values. Any variations in delays between packets in the same flow are due only to network effects. This allows each TCP connection in the experiment to have a different minimum RTT. There is a separate queue for each connection, so the connection's packets stay in order while they are being delayed. Packets from different connections may be forwarded in a different order than they were received based on their delay values. Each packet in a flow is delayed the same amount before being passed on to the next node. As in *DelayBox*, once packets are delayed for their specified time, they are passed up to the single network-level queue for the node. This allows each packet to experience additional queuing delays and possible queue overflows, just as in a regular forwarding node. When a FIN is received for a connection, its entry is deleted from the table.

Note that the *tmix-net* module is symmetric, in that both data and ACKs from the same connection will be delayed the same amount. Because of this, the delay value in the *tmix-net* table is one-half of the RTT given in the connection vector.

C. Differences Between ns-2 and GTNetS

The implementations of *tmix* in *ns*-2 and *GTNetS* are similar, but mainly differ in the way the simulators handle the layers of the network protocol stack.

In both *ns*-2 and *GTNetS*, *tmix-end* is implemented as two `Application` objects, one for the initiators and one for the acceptors. In *ns*-2, an `Application` must be bound to an `Agent` (representing the transport layer), which is then bound to a `Node` (representing the physical end system). In *GTNetS*, when an `Application` is created, its entire network stack is created and automatically bound to it. This `Application`

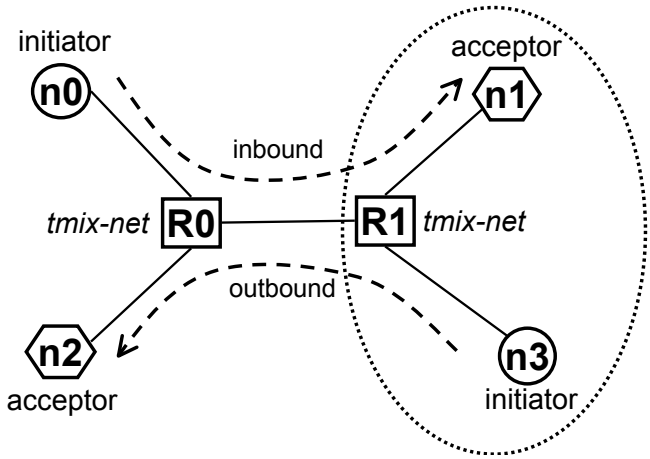

Fig. 5. Simulation Topology With Four Nodes Associated with Two *tmix-end* Modules and Two *tmix-net* Nodes. Nodes `n0` and `n1` are associated with the inbound *tmix-end* module, and nodes `n2` and `n3` are associated with the outbound *tmix-end* module.

is then bound to a `Node`. `Agents` in *ns*-2 correspond to the `Layer4` protocol objects in *GTNetS*.

In the *ns*-2 implementation of *tmix*, we chose to use Full-TCP as the `Agent` because we want to generate network traffic as realistically as possible. Full-TCP provides TCP connection startup and teardown, variable packet sizes, and full-duplex connections. There is an option in *tmix-end* to choose the particular type of Full-TCP `Agent` (Tahoe, Reno, NewReno, or SACK). In *GTNetS*, the TCP-based `Layer4` objects are similar to the Full-TCP `Agents` in *ns*-2. The TCP variant can be specified when a new `Application` is created.

In *ns*-2, we used a maximum segment size (MSS) of 1460 bytes. In *GTNetS*, we encountered issues with buffering of short packets when the MSS was set to 1460 bytes. These issues were eliminated when using the default MSS of 512 bytes.

D. Examples

In Figure 5 we show the topology used in our validation experiments and upon which our examples are based. We use two *tmix-end* modules so that we can represent two-way traffic, with initiators and acceptors on both sides of the network. Recall that a single simulation node acting as an initiator or an acceptor is actually representing a set of end systems. The terms *inbound* and *outbound* refer to the direction of the connections being initiated with respect to the area inside the dotted line in Figure 5.

Figure 6 gives an example of the *ns*-2 TCL commands needed to setup *tmix*. First, a new `TmixEnd` object is created. Then, the initiator and acceptor nodes are assigned, and an ID is assigned to distinguish between each `TmixEnd` instance. Finally, the connection vector file is assigned. To setup *tmix-net*, we create a new `TmixNet` object and assign the connection vector file, initiator node, and acceptor node. By default, *tmix-net* will use the loss rate specified in the connection vector

```
# Setup tmix-end
set tmixIn [new TmixEnd]
$tmixIn set-init $n(0);    # name $n(0) as initiator
$tmixIn set-acc $n(1);     # name $n(1) as acceptor
$tmixIn set-ID 1
$tmixIn set-cvfile "inbound.cvec"

set tmixOut [new TmixEnd]
$tmixOut set-init $n(3);    # name $n(3) as initiator
$tmixOut set-acc $n(2);     # name $n(2) as acceptor
$tmixOut set-ID 2
$tmixOut set-cvfile "outbound.cvec"

# Setup tmix-net
set tmixNetIn [$ns TmixNet]
$tmixNetIn set-flowfile "inbound.cvec" \
                  [$n(0) id] [$n(1) id]
$tmixNetIn set-lossless

set tmixNetOut [$ns TmixNet]
$tmixNetOut set-flowfile "outbound.cvec" \
                    [$n(3) id] [$n(2) id]
$tmixNetOut set-lossless

# Start tmix
$ns at 0.0 "$tmixIn start"
$ns at 0.0 "$tmixOut start"
```

Fig. 6. Example ns Commands to Run *tmix*

file. For a lossless simulation where no losses are enforced by *tmix-net*, the `set-lossless` option should be given.

Figure 7 gives an example of the *GTNetS* commands needed to setup *tmix*. The setup of *tmix* in both *ns*-2 and *GTNetS* is very similar, except that the *ns*-2 script is written in TCL and *GTNetS* is in C++.

Full examples, as well as the source code for the *tmix* implementations in *ns*-2 and *GTNetS*, will be available from our webpage in the near future.

V. Validation

As the validation of *tmix* in *ns*-2 was demonstrated in previous work [17], we use this section to show that the implementation of *tmix* in *GTNetS* is similar to that in *ns*-2. The connection vectors used in this validation were collected from a trace of a 1 Gbps Ethernet link connecting the campus of the University of North Carolina (UNC) with the router of its ISP. Outbound connections are those where the initiator was located at UNC and the acceptor was located somewhere else, while inbound connections are those where the acceptor was located at UNC. Because of popular download sites on campus such as `www.ibiblio.org`, UNC is a net producer of traffic, meaning that there is often more inbound traffic (acceptors at UNC) than outbound traffic.

In our validation, we consider the following metrics:

- *Packet Arrivals* - time series of the number of packets arriving on the link per second
- *Throughput* - number of bits per second transmitted on the link

```
// Setup tmix-end
TmixEnd* tmixEndIn = new TmixEnd();
tmixEndIn->Set_init(n0);
tmixEndIn->Set_acc(n1);
tmixEndIn->Set_cvfile("inbound.cvec");
tmixEndIn->Set_id(1);

TmixEnd* tmixEndOut = new TmixEnd();
tmixEndOut->Set_init(n3);
tmixEndOut->Set_acc(n2);
tmixEndOut->Set_cvfile("outbound.cvec");
tmixEndOut->Set_id(2);

// Setup tmix-net
TmixNet* tmixNetIn = new TmixNet(r0);
tmixNetIn->Set_flowfile ("inbound.cvec", \
                  IPAddr("192.168.1.0"), \
                  IPAddr("192.168.1.1"));
tmixNetIn->Set_lossless();

TmixNet* tmixNetOut = new TmixNet(r1);
tmixNetOut->Set_flowfile ("outbound.cvec", \
                     IPAddr("192.168.1.3"), \
                     IPAddr("192.168.1.2"));
tmixNetOut->Set_lossless();

// Start tmix
tmixEndIn->Start(0.0);
tmixEndOut->Start(0.0);
```

Fig. 7. Example GTNetS Commands to Run *tmix*

- *Active Connections* - time series of the number of active connections per second

For each of these metrics, we show the statistics calculated from the original trace data, from the *ns*-2 *tmix* traces, and from the *GTNetS tmix* traces.

As mentioned earlier, our simulation topology is shown in Figure 5. We use two *tmix-end* modules to simulate initiators and acceptors on both sides of the network. Each link has a link speed of 1 Gbps and a delay of 100 μs. These parameters were set to minimize their effect on the simulation. The RTTs will be specified in the connection vectors.

A. Packet Arrivals

Figure 8 shows the number of packet arrivals per second in the inbound direction, and Figure 9 shows the number of packet arrivals per second in the outbound direction. The *ns-2* line is slightly below the original trace in both figures because of two main issues. First, we found that with Full-TCP, even when the Nagle algorithm was turned off, small packets were still buffered together before being sent. For example, if an initiator was schedule to send a 600-byte ADU at time 1.53 seconds, followed 20 ms later by a 1400-byte ADU, only one 1460-byte packet would be sent at time 1.55 seconds. This, then, reduces the number of packets seen on the link. The second issue we found was with delayed ACKs. If delayed ACKs were not used, there were a larger number of packets for *ns-2* than with the original trace (*i.e.*, these were extra ACKs). Depending on the exact delay timer used (100 ms

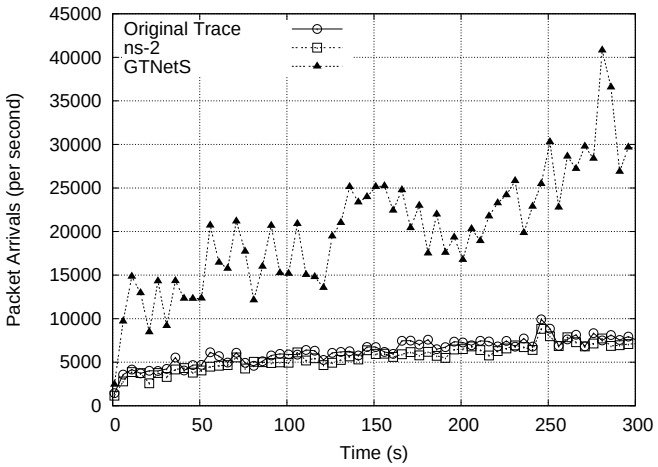

Fig. 8. Packet Arrivals Inbound

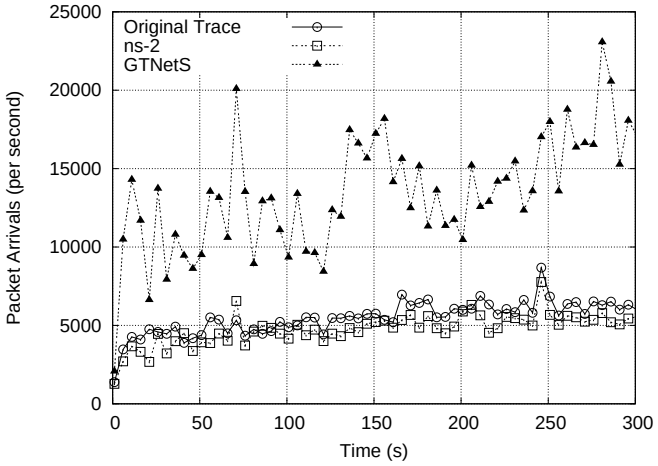

Fig. 9. Packet Arrivals Outbound

in these experiments), there were different numbers of packet arrivals. Our conjecture is that the delayed ACK mechanism in Full-TCP in *ns-2* does not exactly mimic a real TCP stack's delayed ACK mechanism. In addition, there may have been some number of flows in the original trace that did not use delayed ACKs at all.

The *GTNetS* line is much higher than both the original and *ns-2* line. This is because we had to use a MSS of 512 bytes in *GTNetS*, which increased the number of packets used to send larger ADUs. We plan to look into the issues caused in *GTNetS* when the MSS was increased to 1460 bytes.

B. Throughput

Figure 10 shows the throughput observed in the inbound direction, and Figure 11 shows the throughput in the outbound direction. Both *ns*-2 and *GTNetS* follow the original trace throughput quite well. Both simulation lines are close to the original trace. Differences between the *ns*-2 and the *GTNetS* lines may have to do with the different MSS used and the slightly different ways that *tmix-net* was implemented. We will continue to investigate these differences.

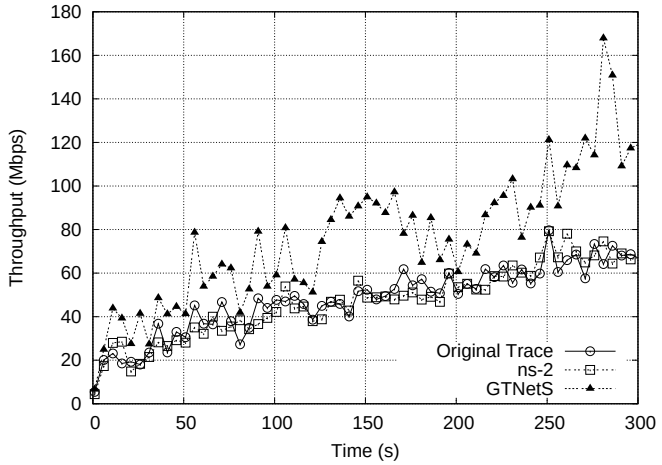

Fig. 10. Throughput Inbound

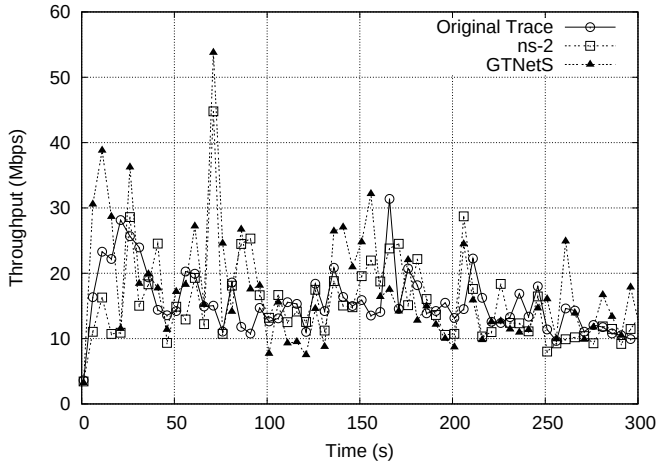

Fig. 11. Throughput Outbound

C. Active Connections

Figure 12 shows the number of active connections per second. The number of active connections provides an indication of the congestion in the network. Both the *ns*-2 and *GTNetS* simulations in this figure were run in the lossy mode (where the loss rates in the connection vector were simulated), and both have similar numbers of active connections as the original trace. If instead the simulations had been run in lossless mode, there would be fewer active connections than in the original trace because more connections would have completed in a shorter amount of time (due to the absence of loss).

D. Running Time

One thing that surprised us was the difference in running times. We expected that the *GTNetS* simulation would run faster since it is implemented solely in C++ rather than a mix of C++ and OTcl, but we found that the *ns*-2 implementation was faster. On a 3.4 GHz Intel Pentium 4 machine with 2 GB of memory, the 300-second *ns-2* simulation completed in about 18 minutes, while the *GTNetS* simulation took just over an hour to complete.

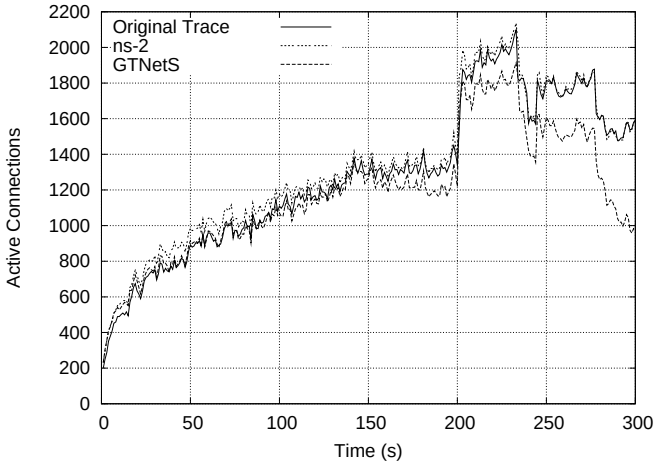

Fig. 12. Active Connections

VI. Conclusions

Network simulations are a useful tool for researchers studying Internet protocols and mechanisms. A major problem with simulations is the lack of realistic models of Internet traffic. The *a-b-t* model and accompanying *tmix* traffic generator can provide for a rapid transition from tracing network links to replaying, in a network-independent manner, the source-level characteristics of the measured traffic. We have implemented the *tmix* generator in the *ns*-2 and *GTNetS* network simulators and have shown that both simulators can produce traffic that is statistically similar to that observed on a real Internet link.

Future work in regards to the *a-b-t* model, as described in [17], includes characterizing UDP traffic, to provide a richer model of Internet links. We will continue to work on the implementations of *tmix* in *ns-2* and *GTNetS* to address some of the issues discussed earlier and to reduce the running time and memory requirements. In addition, we plan to host a repository of connection vector files traced from various links that researchers can download and use in their simulations. We hope that this work will provide network researchers with a method for conducting more realistic experiments than were possible before.

Acknowledgments

We thank George Riley of Georgia Tech for many helpful conversations about *GTNetS* and for contributing code to the *tmix-net* portion of the *GTNetS* implementation. We also thank Shobana Natesan Sampath and Venkata Vasireddi of Clemson University for help in coding *tmix* in *ns*-2. Finally, we thank Felix Hernandez-Campos of the University of North Carolina for collecting the traces and producing the *tmix* connection vectors for the links we used in the validation study.

References

[1] P. Adurthi. Generating *tmix*-based TCP application workloads in ns-2 and GTNetS. Master's thesis, Clemson University, 2006.

[2] P. Barford and M. E. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, pages 151–160, 1998.

[3] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[4] J. Cao, W. S. Cleveland, Y. Gao, K. Jeffay, F. D. Smith, and M. C. Weigle. PackMime-HTTP: Synthetic web traffic generation in ns-2, 2004. Software available at `http://dirt.cs.unc.edu/packmime/`.

[5] J. Cao, W. S. Cleveland, Y. Gao, K. Jeffay, F. D. Smith, and M. C. Weigle. Stochastic models for generating synthetic HTTP source traffic. In *Proceedings of IEEE INFOCOM*, Mar. 2004.

[6] K. Fall and K. Varadhan, editors. *ns Manual*. `http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf`, 2006.

[7] A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proceedings of ACM SIGCOMM*, Sept. 1999.

[8] S. Floyd and E. Kohler. Internet research needs better models. In *Proceedings of the Workshop on Hot Topics in Networks (HOTNETS)*, Princeton, New Jersey, Oct. 2002.

[9] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, Aug. 2001.

[10] F. Hernández-Campos. *Generation and Validation of Empiricially-Derived TCP Application Workloads*. PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, 2006.

[11] F. Hernández-Campos, F. D. Smith, and K. Jeffay. Generating realistic TCP workloads. In *Proceedings of the Computer Measurement Group International Conference (CMG)*, Las Vegas, NV, Dec. 2004.

[12] K.-C. Lan and J. Heidemann. Rapid model parameterization from traffic measurements. *ACM Transactions on Modeling and Computer Simulation*, 12(3):201–229, July 2002.

[13] B. A. Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE INFOCOM*, pages 592–600, 1997.

[14] G. F. Riley. The Georgia Tech network simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools)*, Aug. 2003.

[15] J. Wallerich. NSWEB - A HTTP/1.1 extension to the ns-2 network simulator. `http://www.net.informatik.tu-muenchen.de/~jw/nsweb`, 2004.

[16] M. C. Weigle. DelayBox: Per-flow loss and delay in ns-2, 2004. Software available at `http://dirt.cs.unc.edu/delaybox/`.

[17] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith. Tmix: A tool for generating realistic TCP application workloads in ns-2. *ACM SIGCOMM Computer Communication Review*, 2006. to appear.