Tmix: A Tool for Generating Realistic TCP Application Workloads in ns-2

Michele C. Weigle
Prashanth Adurthi
Clemson University
mcweigle@acm.org, padurth@cs.clemson.edu

Félix Hernández-Campos
Kevin Jeffay
F. Donelson Smith
University of North Carolina at Chapel Hill
{fhernand,jeffay,smithfd}@cs.unc.edu

ABSTRACT

In order to perform realistic network simulations, one needs a traffic generator that is capable of generating realistic synthetic traffic in a closed-loop fashion that "looks like" traffic found on an actual network. We describe such a traffic generation system for the widely used *ns-2* simulator. The system takes as input a packet header trace taken from a network link of interest. The trace is "reverse compiled" into a source-level characterization of each TCP connection present in the trace. The characterization, called a connection vector, is then used as input to an *ns* module called *tmix* that emulates the socket-level behavior of the source application that created the corresponding connection in the trace. This emulation faithfully reproduces the essential pattern of socket reads and writes that the original application performed without knowledge of what the original application actually was. When combined with a network path emulation component we have constructed called *DelayBox*, the resulting traffic generated in the simulation is statistically representative of the traffic measured on the real link. This approach to synthetic traffic generation allows one to automatically repro-duce in *ns* the full range of TCP connections found on an arbitrary link. Thus with our tools, researchers no longer need make arbitrary decisions on how traffic is generated in simulations and can instead easily generate TCP traffic that represents the use of a net-work by the full mix of applications measured on actual network links of interest. The method is evaluated by applying it to packet header traces taken from campus and wide-area networks and comparing the statistical properties of traffic on the measured links with traffic generated by *tmix* in *ns*.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; C.2.3 [**Computer-Communication Networks**]– Network Operations; C.4 [**Performance of Systems**] Modeling Techniques.

General Terms

Measurement, Performance, Experimentation, Verification.

Keywords

Source-level modeling, synthetic traffic generation, *ns*.

1. INTRODUCTION

Networking research has long relied on simulation as the primary vehicle for demonstrating the effectiveness of proposed protocols and mechanisms. Typically, one simulates network hardware and software in software using, for example, the widely used *ns-2* simulator [3]. Experimentation proceeds by simulating the use of the network by a given population of users using applications such as *ftp* or web browsers. Synthetic workload generators are used to inject data into the network according to a model of how the applications or users behave.

This paradigm of simulation follows the philosophy of using source-level descriptions of applications advocated by Floyd and Paxson [12]. The fundamental motivation is that TCP congestion control is an end-to-end *closed-loop* mechanism. In the case of TCP-based applications, TCP's end-to-end congestion control shapes the low-level packet-by-packet traffic processes. For simulating networks used by TCP applications, the generation of network traffic must be accomplished by using models of applications layered over TCP/IP protocol stacks. This is in contrast to an *open-loop* approach in which packets are injected into the simulation according to some model of packet arrival processes (*e.g.*, a Pareto process). The open-loop approach is now largely deprecated as it ignores the essential role of congestion control in shaping packet-level traffic arrival processes. Therefore a critical problem in doing network simulations is generating application-dependent, network-independent workloads that correspond to contemporary models of application or user behavior.

From our experiences performing network simulations, we observe that the networking community lacks contemporary models of application workloads. More precisely, we lack validated tools and methods to go from measurements of network traffic to the generation of synthetic workloads that are statistically representative of the range of applications using the network. Current workload modeling efforts tend to focus on creating *application-specific* workload models such as models of HTTP workloads. The status quo for HTTP is a set of synthetic traffic generators based in large part on the web-browsing measurements of Barford, *et al*. that resulted in the well-known SURGE model and tools [2, 9]. While the results of these studies are widely used today, they were conducted several years ago and were based on measurements of a rather limited set of users. Moreover, they have not been maintained and updated as uses of the web have evolved. Thus, even in the case of the most widely-studied application, there remains no contemporary model of HTTP workloads that accounts for (now) routine uses of the web for applications such as peer-to-peer file sharing and remote email access. The problem is that the development of application-specific workload models is a complex and time-consuming process. Significant effort is required to understand, measure, and model a specific application-layer protocol and this effort must be reinvested each time either the protocol changes or applications change their use of the protocol (*e.g.*, the use of HTTP as a transport protocol for SOAP applications).

Consider the problem of simply measuring application-specific traffic. Typically when constructing workload models, the only means of identifying application-specific traffic in a network is to classify connections by port numbers.[1] For connections that use common reserved ports (*e.g.*, port 80) we can, in theory, infer the application-level protocol in use (HTTP) and, with knowledge of the operation of the application level protocol, construct a source-level model of the workload generated by the application. However, a growing number of applications use port numbers that have not been registered with the IANA. Worse, many applications are configured to use port numbers assigned to other applications (allegedly) as a means of "hiding" their traffic from detection by network administrators or for passing through firewalls. For example, the Internet 2 NetFlow report shows that the percentage of traffic classified as unidentified, in terms of originating application, has increased from 28% in 2002 to 37% in 2006 [19]. However, even if all connections observed on a network could be uniquely associated with an application, constructing workload models requires knowledge of the (sometimes proprietary or "hidden") application-level protocol to deconstruct a connection and understand its behavior. Doing this for the hundreds of applications (or even the top twenty) found on a network is a daunting task.

We advocate a different approach. Our goal is to create an automated method for characterizing the full range of TCP-based applications using a network of interest without any knowledge of which applications are actually present in the network. This characterization should be sufficiently detailed to allow one to statistically reproduce the applications' workload in an *ns* simulation.

The general paradigm we follow is an empirically based method. One first takes one or more packet header traces on a network link and uses the trace(s) to construct a source-level characterization of applications' uses of the network. This source-level workload model is constructed by "reverse compiling" TCP/IP headers into a higher-level, abstract representation that captures the dynamics of both end-user interactions and application-level protocols above the socket layer. Each TCP connection is represented as a *connection vector*. A connection vector, in essence, models how applications use TCP connections as a series of data-unit exchanges between the TCP connection initiator and the connection accepter. The data units we model are not packets or TCP segments but instead correspond to the objects (*e.g.*, files or email messages) or protocol elements (*e.g.*, HTTP GET requests or SMTP HELO messages) as defined by the application and the application protocol. The data units exchanged may be separated by time intervals that represent application processing times or user "think" times. A sequence of such exchanges constitutes the connection's "vector." This model is described in detail in Section 3.

Collectively, the set of connection vectors derived from a network trace is a representation of the aggregate behavior of all the applications found on the measured network. These connection vectors are input to a trace-driven workload-generating program called *tmix* that "replays" the source-level (socket-level) operations of the original applications. In this manner, *tmix* creates inputs to TCP that are statistically similar to the TCP inputs from the applications that created the original packet trace. *tmix* can generate realistic synthetic TCP traffic in either a network testbed or in an *ns* simulation. Here we focus on the implementation of *tmix* in *ns*.

Our workload modeling approach is presented and validated empirically by comparing synthetically generated traffic in *ns* with trace data obtained from the UNC Internet access link (a Gigabit Ethernet link). Similar validations have been performed using measurements from wide-area links such those in Abilene (Internet-2). For a further comparison, we also include a comparison of the same synthetic traffic generated in our testbed. This is done primarily to suggest that in cases where the *ns* simulation results deviate from reality, that these deviations are due in part to shortcomings of the *ns* simulator itself.

We claim our method of representing TCP connections as connection vectors, and using these vectors to generate synthetic workloads, is a natural and substantial step forward. Connection vectors are easily constructed directly from packet traces from existing network links, without any *a priori* knowledge of the variety or type of applications, or application-level protocols, being measured. With our method and tools, the process of going from network traces to generating a synthetic TCP workload that is statistically representative of that observed on the measured link, can be reduced *from months to hours*. But most importantly, connection vectors derived from a trace can be used to generate realistic synthetic traffic that statistically approximates the traffic on the measured link. Thus with our methods, researchers no longer need make arbitrary decisions concerning, for example, the number of short-lived and long-lived flows to simulate in an experiment. Instead one can simply generate the actual mix of TCP connections found on an actual link. Moreover, standard statistical sampling techniques can be used to generate controlled and meaningful departures from approximations to the measured traffic.

Like all workload modeling efforts, our work on *tmix* is necessarily limited. The careful reader will have no trouble identifying some aspect of TCP application behavior that we either fail to account for or are fundamentally unable to model given our approach. Thus, the core issue is one of parsimony: are we able to capture enough information about applications' uses of TCP to reproduce interesting, important, and new measures of traffic in the synthetically generated traffic, and with what effort are we able to do so? In general, this important discussion is beyond the scope of this paper (but is addressed in detail in [15]). We simply note that this paper reports on the reproduction of a combined set of features of real network traffic that to our knowledge have not been previously demonstrated nor addressed in the literature. For this reason we further believe this work to be an important advance.

An additional limitation of the work present here is the fact that we only consider the faithful reproduction of TCP traffic. Modeling and synthetic generation of UDP traffic is a simpler task and one that is accommodated by a straightforward extension of the methods presented here. The UDP work is not presented here because of space limitations but will be addressed in a future paper.

The remainder of this paper is organized as follows. Section 2 reviews related work in workload characterization and generation. Section 3 describes the methodology for constructing connection vectors from packet-header traces and discusses the limitations of the methods. Section 4 describes the *ns* implementation of the *tmix* tool for synthetic workload

[1] This is largely because user privacy concerns dictate that it is inappropriate to record and analyze packet data beyond the TCP/IP header without the prior approval of users.

generation. Section 5 presents a series of validation experiments using traces from the UNC network. Section 6 gives a summary of the results and discusses directions for further research.

2. RELATED WORK

A compelling case for workload generation as one of the key challenges in Internet modeling and simulation is made by Floyd and Paxson [12]. In particular, they emphasize the importance of generating workloads from source-level models in order to perform closed-loop network simulations. Two important measurement efforts that focused on application-specific models, but which preceded the growth of the web, were conducted by Danzig, *et al.*, [4, 10], and by Paxson [25]. These researchers laid the foundation for TCP workload modeling using empirical data to derive distributions for the key random variables that characterize applications at the source level.

Measurements to characterize web usage have been a very active area of research. Web workload generators in use today are often based on web-browsing measurements by Barford, *et al.* [2, 9] that resulted in the well-known SURGE model and tools. More recent models of web workloads have been published by Smith, *et al.* [27], and Cleveland, *et al.* [5, 8]. Finally, we note that source modeling for multimedia data, especially variable bit-rate video, has been an active area of research [16, 20]. A recent study of Real Audio traffic by Mena and Heidemann [24] provides a source-level view of streaming-media flows. Other approaches to traffic modeling with an emphasis on packet-level traffic are surveyed in [17].

Researchers have used the above HTTP workload models to generate web-like traffic in laboratory testbed networks [2, 8], and in the *ns*-2 network simulator. The web traffic workload built-in to *ns* is the *WebTraf* module based on the work of Feldmann, *et al.* [11] (an extension of the SURGE model). The *nsweb* module [28] is another SURGE extension that includes pipelining and persistent HTTP connections. The *PackMime-HTTP* module is another *ns*-2 generator that allows traffic generation based on a model of HTTP traffic developed at Bell Labs [5]. However, despite this previous work, there is no traffic generator for *ns*-2 that can provide a realistic mix of today's TCP applications, such as web, email, and P2P file sharing.

Our project has goals similar to the SAMAN project [21] at ISI. They have developed tools that convert network measurements to application-level models. They have produced a software tool, RAMP, which takes *tcpdump* trace data from a network and generates a set of CDF (cumulative distribution function) files that model Web and FTP applications. In fact, their set of tools for the Web modeling is based on earlier versions [27] of our trace analysis techniques described here. Our goal is to be able to deal with *all* the TCP-based applications found on a network.

Outside of *ns*, special purpose traffic generators exist for testing servers [7] and routers [26], however, both these works ignore the source-level structure of connections. Additionally, commercial synthetic traffic generation products such as Chariot [6] and IXIA exist, but these generators are typically based on a limited number of application source types. Moreover, it is not clear that any are based on empirical measurements of actual Internet traffic.

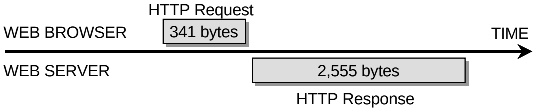

Figure 1: The pattern of ADU exchange in an HTTP 1.0 connection.

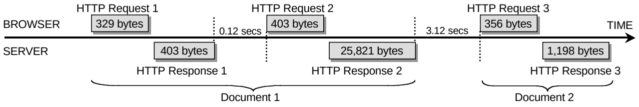

Figure 2: ADU exchanges in an HTTP 1.1 connection.

3. CHARACTERIZING TCP CONNECTIONS[2]

The foundation of our approach to characterizing TCP workloads is the observation that, from the perspective of the network, the vast majority of application-level protocols are based on a few simple patterns of data exchanges within a logical connection between the endpoint processes. Endpoint processes exchange data in units defined by their specific application-level protocol. The sizes of these application-data units (ADUs) depend only on the application protocol and the data objects used in the application and, therefore, are (largely) independent of the sizes of the network-dependent data units employed at the transport layer and below. For example, HTTP requests and responses depend on the sizes of headers defined by the HTTP protocol and the sizes of files referenced but not on the sizes of TCP segments used at the transport layer.

The simplest and most common pattern used by TCP applications arises from the client-server model of application structure and consists of a single ADU exchange. For example, given two endpoints, say a web server and browser, we can represent their behavior over time with the simple diagram shown in Figure 1. A browser makes a request to a server that responds with the requested object. Note that the time interval between the request and the response depends on network or end-system properties that are not directly related to (or controlled by) the application.

More generally, a client makes series of k requests of sizes a_1, a_2, ..., a_k, to a server that responds by transmitting k objects of sizes b_1, b_2, ..., b_k, such that the i^{th} request is not made until the $(i - 1)^{st}$ response is received in full. Application protocols that exchange multiple ADUs between endpoints in a single TCP connection include HTTP/1.1, SMTP, FTP-CONTROL, and NNTP. An example of this pattern, a persistent HTTP/1.1 connection, is shown in Figure 2. In addition to the ADU sizes, we also measure a duration that represents the time between an exchange that is most likely to be independent of the network (*e.g.*, human "think times" or other application-dependent elapsed times). Note that we do not record the time interval between the

[2] The material presented Sections 3 and 4 may be the subject of a pending US patent application [30]. However, the *ns* code described herein will be made freely available for non-commercial use.

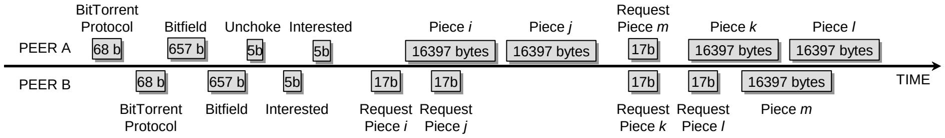

Figure 3: A pattern of concurrent ADU exchanges in a BitTorrent connection.

request and its corresponding response as this time depends on network or end-system properties that are not directly related to (or controlled by) the application.

More formally, we represent a pattern of ADU exchanges as a *connection vector* $C_i = \langle E_{\mathrm{i}}, E_2, ..., E_k \rangle$ consisting of a set of *epochs* $E_i = (a_i, b_i, t_i)$ where a_i is the size of the i^{th} ADU sent from connection initiator to connection acceptor, b_i is the size of the i^{th} ADU sent from connection acceptor to connection initiator, and t_i is the think time between the receipt of the i^{th} "response" ADU and the transmission of the $(i + 1)^{th}$ "request." In addition, for each connection vector *i* we also record its start time T_i. Thus we can view a TCP application as generating a number of time-separated epochs where each epoch is characterized by ADU sizes and an inter-epoch time interval. For example, the HTTP connection in Figure 2 represents a persistent HTTP connection over which a browser sent three requests of 329, 403, and 365 bytes, respectively, and a server responded to each of them with HTTP responses (including object content) of 403, 25821, and 1198 bytes, respectively. The second request was sent by the browser 120 milliseconds after the last segment of the first response was received and the third request was sent 3,120 milliseconds after the second response was received. This connection would represented as the connection vector:

$C_i = \langle (329, 403, 0.12), (403, 25821, 3.12), (356, 1198, 0) \rangle$. Abstractly we say that the connection vector consists of three epochs corresponding to the three HTTP request/response exchanges.

Note that the *a*-*b*-*t* characterization admits the possibility of an application protocol omitting one of the ADUs during an exchange (*e.g.*, epochs of the form $(a_i, 0, t_i)$ and $(0, b_i, t_i)$, are allowed). Single ADU epochs are commonly found in *ftp*-data connections or in streaming media connections.

A final pattern allows for ADU transmissions by the two endpoints to overlap in time (*i.e.*, to be concurrent) as shown in Figure 3. This pattern is not commonly implemented in applications used in the Internet today, but can be used by application-level protocols such as HTTP/1.1, NNTP, and BitTorrent. While uncommon, such concurrent connections often carry a significant fraction of the total bytes seen in a trace (15%-35% of the total bytes in traces we have processed) and hence are critical to model if one wants to generate realistic traffic mixes. The ability to model and reproduce the behavior of concurrent connections is, we believe, unique to our modeling effort.

To represent concurrent ADU exchanges, the actions of each endpoint are considered to operate independently of each other so each endpoint is a separate source generating ADUs that appear as a sequence of epochs following a uni-directional flow pattern (see [14, 15]).

3.1 From packet traces to connection vectors

Modeling TCP connections as a pattern of ADU transmissions provides a unified view of connections that does not depend on the specific applications driving each TCP connection. The first step in the modeling process is to acquire a trace of TCP/IP headers and process the trace to produce a set of connection vectors; one vector for each TCP connection in the trace.

The basic method for determining ADU boundaries and inter-ADU idle times ("think times") is described in detail in [27] and briefly summarized here. Connection vectors for sequential (non-concurrent) connections can be computed from unidirectional traces. The analysis proceeds by examining sequence numbers and acknowledgement numbers in TCP segments. Changes in sequence numbers in TCP segments are used to compute ADU sizes flowing in the direction traced and changes in ACK values are used to infer ADU sizes flowing in the opposite (not traced) direction. In sequential connections, there will be an alternating pattern of advances in the ACK values followed by advances in the data sequence values (or vice versa). This observation is used to construct a rule for inferring the beginning and ending TCP segments of an ADU and the boundary between exchanges. Of course, other events such as FIN, Reset, or idle times greater than a threshold, can mark ADU boundaries as well. Timestamps on the *tcpdump* of segments marking the beginning or end of an ADU are used to compute the inter-epoch times and timestamps on the SYN segments are used to compute connection inter-arrival times. The complexity of this analysis (per connection) is $O(sW)$ where s is the number of segments in the connection and W is the receiver's maximum advertised window size.

For example, consider again the pattern of sequential ADU exchanges illustrated in Figure 2. This example, taken from real measurement data, manifested itself in a packet header trace as 29 TCP segments (including SYNs and FINs). If the connection analysis is applied to this sequence of packet headers it would generate the 3-epoch connection vector C_i, listed above. Note that while the actual sizes and timing of the TCP segments represented in the original packet header trace are network-dependent, the analysis has produced a compact, summary representation that models the source-level behaviors of a web browser and server.[3]

The computation of connection vectors that derive from application protocols which overlap rather than alternate exchanges between endpoints (the pattern in Figure 3) requires a different method than the one described above. We have developed an algorithm that can be used to detect and model instances of concurrent ADU exchanges in a TCP connection. Briefly, the idea is to detect situations in which both end points have unacknowledged data in flight. The algorithm detects concurrent data exchanges between two end points A and B in which there exists at least one pair of non-empty TCP segments p and q such that p is sent from A to B, q is sent from B to A, and the following two inequalities are satisfied: $p.seqno > q.ackno$ and $q.seqno > p.ackno$. If the conversation between A and B is sequential, then for every pair of segments p, q, either p was sent after q reached A, in which case $q.seqno \leq p.ackno$, or q was sent after p reached B, in which case $p.seqno \leq q.ackno$. The classification of concurrent connections requires a bi-directional header trace.

[3] In general, the connection vector representation of TCP connections is 50-100 times smaller than a packet header trace.

4. WORKLOAD GENERATION FROM CONNECTION VECTORS

The *tmix* traffic generation tool takes as input a set of connection vectors. If the connection vectors come directly from a trace on a network link then, as described next, the *tmix* tool will faithfully reproduce the traffic observed on the link. This reproduction, called "replay," is the basis for the validation procedure described in Section 5 where *tmix* is shown to generate TCP traffic that is statistically representative of the measured traffic from which the connection vectors derive. However, in addition to replay, a wide-range of alternative workload generation scenarios are also possible. For example, one can derive distributions of values for the key random variables that characterize applications at the source level (*e.g.*, distributions of ADU sizes, time values, number of epochs, *etc*.). These can be used to populate analytic or empirical models of the workload in much the same way as has been done for application-specific models (*e.g.*, as in the SURGE model for web browsing).

Alternatively, if one wanted to model a "representative" workload for an entire network, traces from several links in the network could be processed to produce their connection vectors and these vectors could be pooled into a library of TCP connection vectors. From this library, random samples could be drawn to create a new trace that would model the aggregate workload. To generate this workload in a simulation, one could assign start times for each TCP connection according to some model of connection arrivals (perhaps derived from the original packet traces).

A third possibility is to apply the methods of *semi-experiments* introduced in [18] at the application level instead of at the packet level. For example, one could replace the recorded start times for TCP connections with start times randomly selected from a given distribution of inter-arrival times (*e.g.*, Weibull) in order to study the effects of changes in the connection arrival process on a simulation. Other interesting transforms to consider include replacing the recorded ADU sizes with sizes drawn from analytic distributions (*e.g.*, LogNormal) with different parameter settings. One might replace all multi-epoch connections with single-epoch connections where the new *a* and *b* values are the sums of the original *a* and *b* values and the *t* values are eliminated (this is similar to using NetFlow data to model TCP connections). All such transforms provide researchers with a powerful new tool to use in simulations for studying the effects of workload characteristics in networks. The simple structure of a connection vector makes it a flexible tool for a broad range of approaches to synthetic workload generation.

4.1 The *tmix* workload generation tool

tmix takes a set of connection vectors and replays them to generate synthetic TCP traffic. The vectors can be the exact connection vectors found in a trace or an artificial set generated via any of the methods described above.

At a high-level, *tmix* will initiate TCP connections at times taken from the T_i and, for each connection, send and receive data as specified in C_i. For an example, consider the replay of an *a-b-t* trace containing the connection vector described in Section 3 corresponding to the persistent HTTP connection in Figure 2:

C_i = <(329, 403, 0.12), (403, 25821, 3.12), (356, 1198, 0)>.

- At time T_i the *tmix* connection initiator establishes a new TCP connection to the *tmix* connection acceptor.
- The initiator writes 329 bytes to its socket and reads 403 bytes.
- Conversely, the connection acceptor reads 329 bytes from its socket and writes 403 bytes.
- After the initiator has read the 403 bytes, it sleeps for 120 milliseconds and then writes 403 bytes and reads 25,821 bytes.
- The acceptor reads 403 bytes and writes 25,821 bytes.
- After sleeping for 3,120 milliseconds, the third exchange of data units is handled in the same way and the TCP connection is terminated.

The following is the representation of the connection vector used by *tmix*:

```
SEQ 102345 3      // Sequential connection
w 16384 16384     // Window size
r 13450           // Minimum RTT
> 329             // Epoch 1
< 402
t 12000
> 403             // Epoch 2
< 25821
t 312000
> 356             // Epoch 3
< 1198
```

The input specifies that a sequential connection should be started at time 102,345 milliseconds and that the connection will consist of 3 epochs. Each endpoint has a maximum TCP receiver window of 16,384 bytes. The base (minimum) RTT of the connection is 13.450 milliseconds. The window sizes and RTTs are optional. If present they are used with an additional *ns* module we have developed that implements per flow delays and losses. This component, called *DelayBox* [29], is required to emulate certain network path properties to enable us to validate traffic synthetically generated in *ns* against the actual measured traffic. (The connection's window size and minimum RTT are extracted from the trace data using the techniques described in [1].)

Lines beginning with '>' indicate the number of bytes sent from initiator to acceptor (*i.e.,* the *a* size), and those beginning with '<' indicate the number of bytes sent from acceptor to initiator (*i.e.,* the *b* size). The lines beginning with 't' indicate the amount of time (in microseconds) between b_i and a_{i+1}.

Below we show an example connection vector describing a concurrent connection with 2 epochs in each direction:

```
CONC 3737620 2 2    // Concurrent connection
w 65535 64240       // Window size
r 166883            // Minimum RTT
c> 91
t> 2514493
c< 111
t< 2538395
c> 55
t> 1985074
c< 118
t< 7516197
```

Lines beginning with 'c>' indicate the number of bytes sent from initiator to acceptor (*i.e.,* the *a* size), and those beginning with 'c<' indicate the number of bytes sent from acceptor to initiator (*i.e.,* the *b* size). Lines beginning with 't>' indicate the amount of time (in microseconds) between a_i and a_{i+1}. Likewise, lines beginning with 't<' indicate the amount of time between b_i and b_{i+1}. Note that unlike the case with sequential connector vectors, in concurrent connection vectors there is no implied ordering between the transmission of *a*'s and *b*'s. In this representation, the notation is interpreted as specifying a sequence for each direction of the connection independently. Thus, for example, although the connection vector above lists an *a* data unit of 91 bytes before a *b* data unit of 111 bytes, this first *a* and first *b* will actually be transmitted concurrently.

4.2 Implementation of *tmix* in *ns*

The *a*-*b*-*t* connection vector model includes parameters for end systems (connection start time, *a* sizes, *b* sizes, and *t* durations) as well as the optional path parameters (minimum RTTs, loss rates). We split the implementation of *tmix* along these lines. The module *tmix-end* includes the implementation of the end system emulation, and the module *tmix-net* includes the implementation of the network path emulation.

Because we want to generate network traffic as realistically as possible, we base *tmix* on the Full-TCP model. Full-TCP provides TCP connection startup and teardown, variable packet sizes, and full-duplex connections. The connection vectors produced from the *a*-*b*-*t* model generate two-way traffic, with initiators and acceptors on both sides of the network. By default, *tmix* models a single direction with initiators on one side of the network and acceptors on the other. Users who wish to generate two-way traffic should use two separate *tmix* modules (as we have done in the validation) to create two-way traffic.

4.2.1 tmix-end

The *tmix-end* module controls all the activities of a set of initiators and acceptors. Upon startup, the *tmix-end* module reads the connection vectors from a specified file. Two lists are then created. One list holds each connection's ID and start time, and the other list (indexed by connection ID) holds the remaining parameters for each connection (type of connection (sequential or concurrent), initiator window size, acceptor window size, connection start time, list of *a* sizes, list of *b* sizes, list of *t* times).

Once the connection vector file has been processed, new connections are started according to the connection start time list. 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.

4.2.1.1 Sequential Connections

The initiator uses the connection's *a* sizes and inter-request times, while the acceptor uses the connection's *b* sizes and server delays. The initiator sends *a*-type ADUs in the order specified and with the appropriate gap. The inter- *a*-type ADU gap specifies the time the initiator waits between receiving a response and sending a new *a*-type ADU. Once the acceptor receives an *a*-type ADU, it waits for the time specified in the next server delay and then sends the next *b*-type ADU in the list. After the acceptor has sent its final response, it sends a FIN to close the connection.

4.2.1.2 Concurrent Connections

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 the inter- *a*-type ADU delay time before sending a new *a*-type ADU). The acceptor sends its *b*-type ADUs according to its schedule (waiting the inter- *b*-type delay 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.

4.2.2 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 nodes used by *tmix-end*. Upon startup, the *tmix-net* module reads each connection's ID, source, destination, RTT, and loss rate 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. The *tmix-net* module is symmetric, in that both a data and ACK from the same connection will be delayed the same amount. Because of this, the delay value in the table is the connection's RTT/2. *tmix-net* uses the aforementioned *DelayBox* component to implement per-flow delays.

Using *tmix*, each packet in a flow is delayed the same amount before being passed on to the next *ns* node. 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. 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 *ns* forwarding node. When a FIN is received for a connection, its entry is deleted from the table.

5. VALIDATION EXPERIMENTS

In this section we describe experiments designed to validate our approach to workload characterization and generation. Our experimental procedure is based on the following steps:

- Acquire a TCP/IP header trace from an Internet link.
- Filter this Internet link trace to obtain a *sub-trace* consisting of all the packets from all the TCP connections to be included in the workload generation. For the experiments described in this paper, a sub-trace includes all packets from TCP connections where the SYN or SYN+ACK was present in the trace (so we could explicitly identify the initiator of the connection) and the connection was terminated by FIN or RST. This eliminates only those connections that were in progress when the packet trace began and ended. In the remainder of the paper, phrases like "UNC trace" will refer to the sub-trace derived according to the above description. We also refer to these sub-traces as the "original" traces.
- Derive a trace, $\mathcal{T}$, of *a*-*b*-*t* connection vectors from the sub-trace packet headers using the process described in Section 3.2
- Use $\mathcal{T}$ to generate the workload in an *ns*-2 simulation with the *tmix* generator described in section 4.2.
- Capture the packet trace from the simulation. In the remainder of this paper, phrases "UNC simulation" will refer to the packet traffic captured from the *ns*-2 simulation.
- Compare various properties of the traffic in the original trace with the simulation trace.

We report the results from applying this approach to TCP/IP header traces from a 1 Gbps Ethernet link connecting the campus of the University of North Carolina at Chapel Hill (UNC) with the

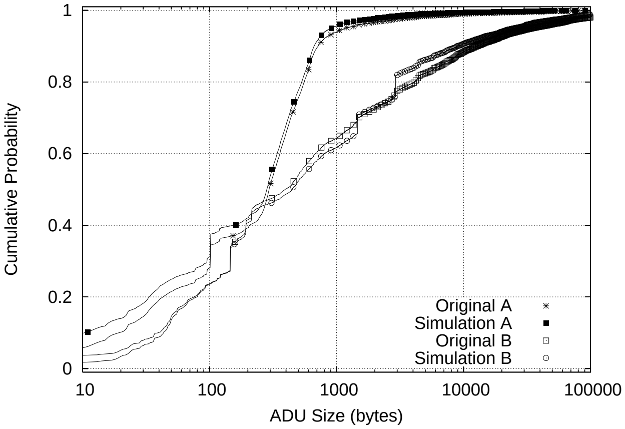

Figure 4: ADU sizes from the original trace and the simulation. Body of the distribution.

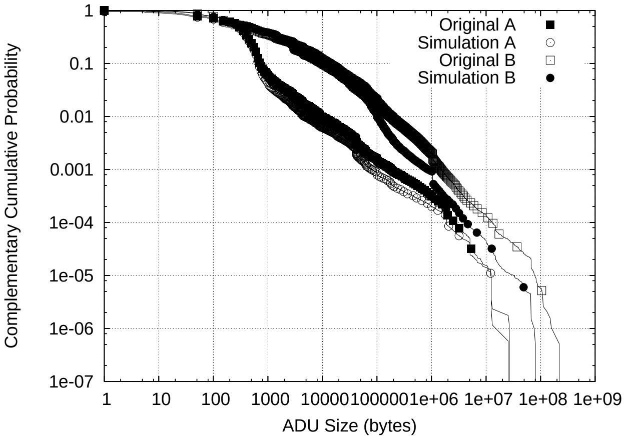

Figure 5: ADU sizes from the original trace and the simulation. Tail of the distribution.

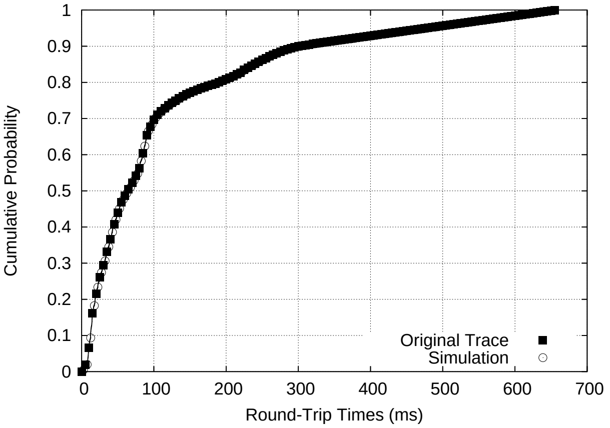

Figure 6: Empirical and simulation RTT distributions.

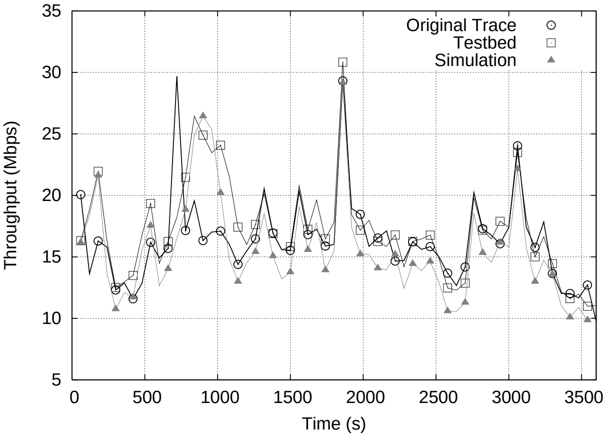

Figure 7: UNC throughput with empirical RTTs and window sizes, outbound.

router of its ISP. The UNC access-link trace is a one-hour bi-directional trace acquired using standard tools (*e.g.*, *tcpdump*).

The *ns*-2 topology used for the validation is a classical "dumbbell" network (a 1 Gbps network connecting two clusters of emulated end systems). While this network topology is in no way representative of the network traced (the UNC campus), by using the *tmix-net* component to emulate per flow RTTs, window sizes, and losses (using measurement data acquired from the trace along with the connection vectors), we are in fact able to closely reproduce in the *ns* simulation important measures of network performance observed on the UNC network.

5.1 Verification of source-dependent properties

Our methodology makes strong emphasis on accurately modeling the way in which TCP connections are used by the sources. Our underlying assumption is that these source-level dependent properties are mostly independent of the specific network-dependent properties of the link in which they are measured. If this is true, we should be able to use our model for generating traffic in testbeds and simulators and safely change some network mechanism (*e.g.*, the TCP flavor or the queuing mechanism) but maintain the same TCP application workload.

We validate this idea in two ways. The first, reported in [15], is to instrument a set of TCP applications to record actual ADU sizes and inter-ADU idle times. A header trace of the execution of these applications is taken and the *a*'s, *b*'s, and *t*'s we compute from this trace are compared against the data measured by the applications. As shown in [15], our heuristics for computing connection vectors are surprisingly accurate.

The second validation is to study the source-dependent properties of a packet header trace from a real link, and compare with the source-level properties of a packet header trace collected from a *tmix* simulation. If the source-dependent properties remain unchanged, we would prove that these properties (and the way we use them to generate traffic) represent a *fixed point* that does not depend on the underlying network mechanisms.

Figure 4 compares the bodies of the distributions of *a* and *b* data unit sizes from the UNC trace with the UNC simulation, while Figure 5 compares the tails of the same distributions. One interesting feature of these distributions was that the distribution of *a* sizes was considerably lighter in the body of the distribution than the distribution of *b* sizes. This confirms our expectation that *a* units were more likely to be small because they are usually requests (*e.g.*, in HTTP) and the *b* units (the responses) are more likely to be larger. The distributions appeared to be consistent with a heavy-tailed distribution.

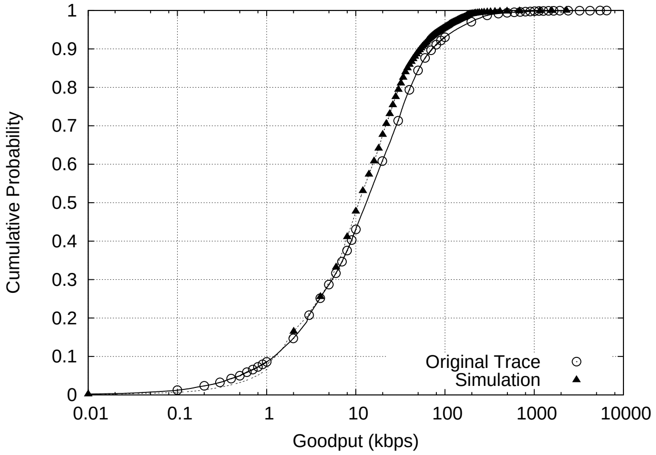

Figure 8: Cumulative distribution of goodput per connection.

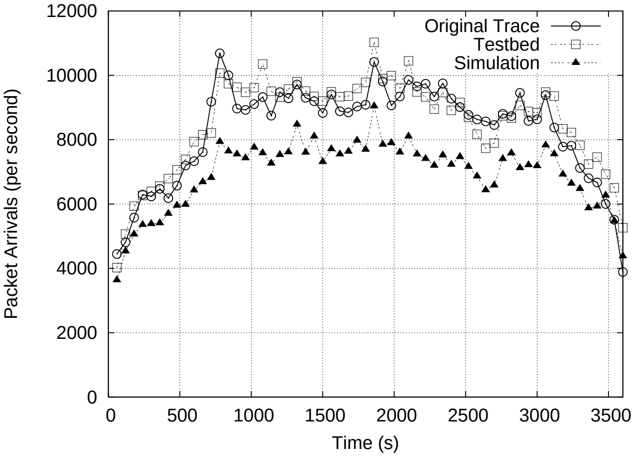

Figure 10: Time series of packet arrivals.

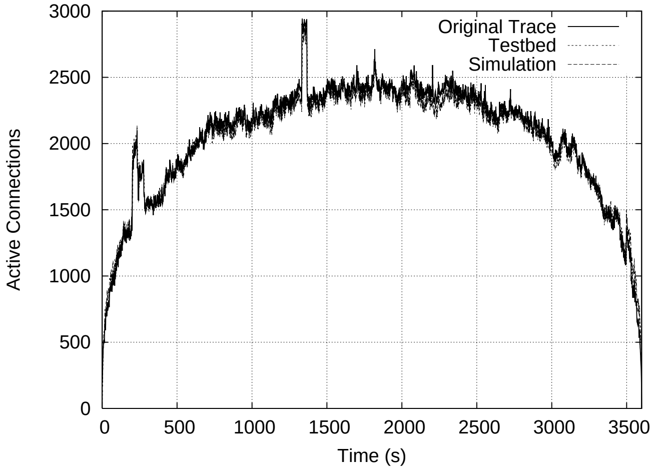

Figure 9: Time series of active connections.

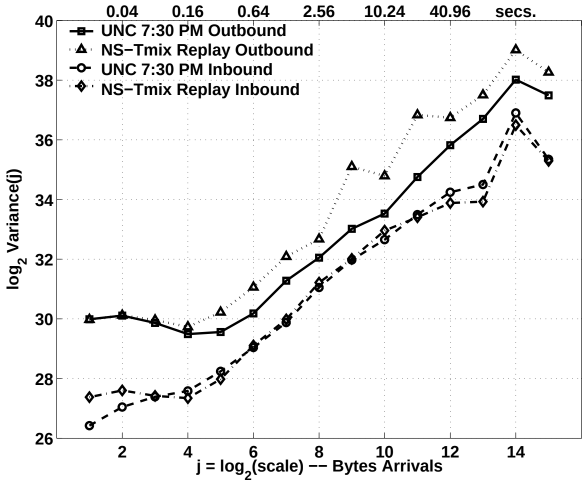

Figure 11: Logscale diagram for inbound and outbound directions.

5.2 Validation against real link traffic

The most demanding validation experiment that we could devise was to use the workload models derived from the UNC trace with the goal of reproducing certain essential characteristics of the original packet traces when the workloads are simulated in *ns*-2. The question being explored is: to what extent can we reproduce the traffic found on real network links in a simulation?

Our metrics for evaluating the fidelity of reproduction between the real and simulated packet traffic include the following:

- The link load or throughput – the number of bits per second (including protocol headers) transmitted on a link. This metric would be used by experimenters to gauge the level of congestion on simulated links. Note that because we can replay the applications' use of TCP connections at both endpoints, we are able to generate the packet-level traffic flowing in both directions of the link concurrently.
- The distribution of goodput (application-level throughput) across flows. This metric is a more sophisticated measure of link throughput as it speaks to how aggregate link throughput is realized. Moreover, reproduction of goodput implies that *tmix*'s emulation of applications reproduces both network and application-layer measures of throughput.
- The time series of active connections per bin size. Accurately reproducing the number of active connections is important for experimenting with network services and mechanisms that maintain per flow state.
- The statistical properties of the time series of counts of arriving packets and bytes on a link in an interval of time (*e.g.*, Hurst parameter estimates, wavelet spectra). The breakthrough results in studying these time series over the past several years have identified *self-similarity* and *long-range dependence* as fundamental features of network traffic that should be preserved in synthetic traffic.

To reproduce traffic from a real link in a laboratory network, we must consider a second set of factors that are *network-dependent* but are, to a first approximation, independent of the applications using the network. These are further classified as being determined at the TCP *endpoints* or along the *path* between endpoints. The primary network-dependent endpoint factors we consider are the TCP sender and receiver window sizes (*ns* does not simulate receiver window sizes, but does implement a sender maximum window) and the maximum segment size (MSS). Other network-dependent endpoint factors one might consider (and that we have experimented with but do not provide data here for) include the TCP variant used (*e.g.*, Reno, NewReno, SACK, *etc*.). The network-dependent path factors we consider in this paper are the distributions of per-flow round-trip times and loss rates. We are working on incorporating per-flow bottleneck bandwidths.

Figure 6 shows the distribution of estimated RTT per flow in the original traces compared with the achieved replay RTT distribution. There was a good match between the RTTs in the real network and their approximation in the simulation.

The bytes transmitted on the UNC link in 1-minute intervals are shown in Figure 7. The simulation appears to track the fluctuations in load reasonably well. However, overall there is a slight trend for the simulation to generate less throughput than was observed on the UNC network. To test whether this effect is an artifact of the *tmix* implementation or whether it is an artifact of *ns*, we also show the performance of a comparable implementation of *tmix* on a laboratory network testbed. (The testbed, described in more detail in [22], is also structured as a dumbbell network with a 1 Gbps link in the center. Technology similar to that described here is used to emulate minimum per-flow delays so as to ensure all flows experience a different minimum round-trip time.)

We interpret the fact that the testbed implementation of *tmix* more closely approximates the original UNC throughput as an indication that in principle *tmix* can sufficiently reproduce the aggregate throughput of TCP connections. Moreover, while a complete discussion is beyond the scope of this paper, we have ample evidence to suggest that the essential problem here lies in the *ns* software itself.

Figure 8 shows the CDF of goodput achieved per connection, and we see that the simulation follows the original trace rather well. Thus while the aggregate throughput is less than perfect, on a per flow basis, application-layer throughput is closely approximated.

Figure 9 shows the time series of active connections. In this case there is an excellent match between the original trace, the *ns* simulation, and the testbed data. The bell-shaped nature of the curve is an artifact of startup and termination effects inherent in the original packet header trace (see [15]).

For evaluating how well we reproduced the packet- and byte-arrival time series, we used the methods (and MATLAB software) developed by Abry and Veitch [14] to study the wavelet spectrum of the time series. The output of this tool is a log-scale diagram that provides a visualization of the scale-dependent variability in the data. Briefly, the logscale diagram plots the $\log_2$ of the (estimated) variance of the Daubechies wavelet coefficients for the time series against the $\log_2$ of the scale (j) used in computing the coefficients. The wavelet coefficients are computed for scales up to 2^{16}. Since the scale effectively sets the time scale at which the wavelet analysis is applied there is a direct relationship between scale and time intervals (see the top labels of the following plots). For processes that exhibit long-range dependence, the logscale diagram will exhibit a region in which there is an approximately linear relationship with slope > 0 between scale and variance. An estimate of the Hurst parameter along with confidence intervals on the estimate can be obtained from the slope of this line $H = (slope+1)/2$. For more information than this (grossly oversimplified) summary, see [14].

Figure 10 shows the time series of packet arrivals. While the testbed implementation faithfully reproduces the packet arrival process, the *ns* simulation does not. We again interpret this as a shortcoming of the *ns* simulator itself. In particular, the packet arrival process in *ns* is negatively affected by *ns*'s implementation of delayed-ACKs.

Figure 11 shows the logscale diagram for both directions of the UNC trace. Both the original and the simulation show strong scaling starting around 500 milliseconds, so the simulation substantially reproduces the long-range dependence of the traffic. The strength of these scaling in the inbound direction, as estimated by the Hurst parameters, was $H = 0.95$ for the original (the confidence interval was between 0.93 and 0.98) and $H = 0.94$ for the simulation (C.I. = [0.91, 0.96]).

The results presented above show that is possible to use workload models and reproduce with reasonable fidelity the traffic from access links like UNC in an *ns* simulation. This also validates the workload modeling and generation approach.

6. SUMMARY AND CONCLUSIONS

Simulation is the dominant method for evaluating most networking technologies. However, it is well known that the quality of a simulation is only as good as the quality of its inputs. An overlooked aspect of simulation methodology is the problem of generating realistic synthetic workloads to drive a simulation or laboratory experiment. We have developed an empirically-based approach to workload generation. Starting from a trace of TCP/IP headers on a production network, a model is constructed for all the TCP connections observed in the network. The model, a set of *a-b-t* connection vectors, can be used in the workload generator *tmix* to replay the connections and reproduce the application-level behaviors observed on the original network. Moreover, by combining set of connection vectors or sub-sampling vectors according to any number of heuristics, a wide range of meaningful departures from reality are possible. This enables researchers to perform "what if" experiments where the synthetic traffic can be manipulated in controlled ways.

We believe this approach to source-level modeling, and the *tmix* generator, are significant contributions to network evaluations, specifically because of their ability to automatically generate valid workload models representing all TCP applications in a network with no *a priori* knowledge about them. Our work therefore serves to demonstrate that researchers need not make arbitrary decisions when performing simulations such as deciding the number of flows to generate or the mix of "long-lived" versus "short-lived" flows. Given an easily acquired TCP/IP header trace, it is straightforward to populate a workload generator and instantiate a generation environment capable of reproducing a broad spectrum of interesting and important features of network traffic. For this reason, we believe this work holds the potential to improve the level of realism in network simulations.

7. ACKNOWLEDGMENTS

We would like to thank Shobana Natesan Sampath and Venkata Vasireddi of Clemson University for their help in the coding of *tmix* in *ns*.

This work was supported in parts by the National Science Foundation (grants CCR-0208924, EIA-0303590, and ANI-0323648), Cisco Systems Inc., and the IBM Corporation.

8. REFERENCES

[1] J. Aikat, J. Kaur, F.D. Smith, and K. Jeffay, Variability in TCP Round-trip Times, *Proc. ACM SIGCOMM Internet Measurement Conference*, Miami Beach, FL, Oct. 2003, pp. 279-284.

[2] P. Barford and M. E. Crovella, A Performance Evaluation of HyperText Transfer Protocols, *Proc. ACM SIGMETRICS*, Atlanta, GA, May 1999, pp. 188-197.

[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] R. Caceres, P. Danzig, S. Jamin, and D. Mitzel, Characteristics of Wide-Area TCP/IP Conversations, *Proc. ACM SIGCOMM*, Zurich, Switzerland, Sept. 1991, pp. 101-112.

[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, *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004, pp. 1547-1558.

[6] Chariot Performance Evaluation Platform, NetIQ Software Inc, `http://www.netiq.com/products/chr/`.

[7] Y.-C. Cheng, U. Hölzle, N. Cardwell, S. Savage, and G.M. Voelker, Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying, *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2004, pp. 87-98.

[8] W.S. Cleveland, D. Lin, and D.X. Sun, IP Packet Generation: Statistical Models for TCP Start Times Based on Connection-rate Superposition, *Proc. ACM SIGMETRICS*, Santa Clara, CA, June 2000, pp. 166-177.

[9] M. Crovella, and A. Bestavros, Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes, *IEEE/ACM Transactions on Networking*, 5(6):835–846, Dec. 1997.

[10] P. Danzig, S. Jamin, R. Caceres, D. Mitzel, and D. Estrin, An Empirical Workload Model for Driving Wide-Area TCP/IP Network Simulations, *Internetworking: Research and Experience*, 3(1):1–26, 1992.

[11] A. Feldmann, P. Huang, A.C. Gilbert, and W. Willinger, Dynamics of IP traffic: A study of the role of variability and the impact of control, *Proc. ACM SIGCOMM*, Cambridge, MA, Aug. 1999, pp. 301-313.

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

[13] F. Hernández-Campos, A.B. Nobel, F.D. Smith, and K. Jeffay, Understanding Patterns of TCP Connection Usage with Statistical Clustering, *Proc. IEEE MASCOTS*, Atlanta, GA, Sept. 2005, pp. 35-44.

[14] F. Hernández-Campos, F.D. Smith, K. Jeffay, Generating Realistic TCP Workloads, *Proc. Computer Measurement Group Intl. Conf.*, Las Vegas, NV, Dec 2004, pp. 273-284.

[15] F. Hernández-Campos, *Generation and Validation of Empirically-Derived TCP Application Workloads*, Ph.D. Dissertation, Dept. of Computer Science, UNC Chapel Hill, 2006.

[16] D. Heyman, and T.V. Lakshman, Source Models for VBR Broadcast Video Traffic, *IEEE/ACM Transactions on Networking*, 4(1):37–46, Feb. 1996.

[17] H. Hlavacs, G. Kostsis, and C. Steinkellner, *Traffic Source Modeling*, Technical Report TR-99101, Institute of Applied Computer Science and Information Systems, University of Vienna, 1999.

[18] N. Hohn, D. Veitch, and P. Abry, Does Fractal Scaling at the IP Level Depend on TCP Flow Arrival Processes?, *Proc. ACM SIGCOMM Internet Measurement Workshop,* Marseille, France, pp. 63-68, Nov. 2002.

[19] `http://netflow.internet2.edu/`.

[20] E.W. Knightly, and H. Zhang, D-BIBD: An Accurate Traffic Model for Providing QoS Guarantees to VBR Traffic, *IEEE/ACM Transactions on Networking*, 5(2):219–231, Apr. 1997.

[21] 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.

[22] L. Le, J. Aikat, K. Jeffay, F.D. Smith, The Effects of Active Queue Management on Web Performance, *Proc. ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003, pp. 265-276.

[23] B. Mah, An Empirical Model of HTTP Network Traffic, *Proc. IEEE INFOCOM,* Apr. 1997, pp. 592-600.

[24] A. Mena and J. Heidemann, An Empirical Study of Real Audio Traffic, *Proc. IEEE INFOCOM*, Tel-Aviv, Israel, Mar. 2000, pp. 101-110.

[25] V. Paxson. Empirically Derived Analytic Models of Wide-Area TCP Connections, *IEEE/ACM Transactions on Networking*, 2(4):316-36, Aug. 1994.

[26] J. Sommers and P. Barford, Self-Configuring Network Traffic Generation, *Proc. ACM IMC 2004*, Taormina, Italy, October 2004, pp. 68-81.

[27] F.D. Smith, F. Hernández-Campos, and K. Jeffay. What TCP/IP Protocol Headers Can Tell Us About the Web, *Proc. ACM SIGMETRICS*, Cambridge, MA, June 2001, pp. 245-256.

[28] J. Wallerich, NSWEB - A HTTP/1.1 Extension to the NS-2 Network Simulator, *http://www.net.informatik.tu-muenchen.de/~jw/nsweb/,* 2004.

[29] M.C. Weigle, *DelayBox: Per-flow Delay and Loss in ns*, in "The ns manual," K. Fall, K. Varadhan, eds, *http://www.isi.edu/nsnam/ns/doc/.*

[30] USA Patent Application, 20060083231, Methods, Systems, and Computer Program Products for Modeling and Simulating Application-Level Traffic Characteristics in a Network Based on Transport and Network Layer Header Information, April 2006.