GENERATING REALISTIC TCP WORKLOADS

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

Abstract

The workload of a network is usually a heterogeneous aggregate of services and applications, driven by a large number of users. This complexity makes it challenging to evaluate the performance of network mechanisms and configurations under realistic conditions. We propose a new methodology for transforming anonymized traces of packet headers into application-neutral models of network traffic. These models are suitable for synthetic traffic generation in simulations and testbeds, preserving the end-to-end nature of network traffic. Our approach provides a tool for studying and tuning the realism of synthetic traffic.

Introduction

Evaluating the performance of network protocols and mechanisms generally requires careful experimentation in simulators and testbed environments. As with any other performance analysis task, a critical element of these experiments is the availability of a realistic workload or set of workloads that can stress the technology in a manner that is representative of the deployment conditions. Despite the increasing availability of measurement results and packet traces from real networks, there is no accepted method for constructing realistic workloads, so networking researchers and performance analysts have to rely on simplistic or incomplete models of network traffic for their experiments.

One essential difference between network workloads and other workloads, such those of storage systems, is the closed feedback loop created by the ubiquitous Transport Control Protocol (TCP). This protocol, responsible for the end-to-end delivery of the vast majority of the traffic on the Internet, reacts to network conditions by retransmitting lost packets and adjusting sending rates to the perceived level of congestion in the network [2]. As a consequence, we cannot simply collect a trace of the packets traversing a network element and replay the trace in our experiments, since the new conditions in the experimental environment would have had an effect on the behavior of TCP that is not present in the packet trace. In other words, replaying a packet trace breaks the feedback loop of TCP. For example, it is incorrect to collect a packet trace in a 1-Gbps link with a mean load of 650 Mbps and use it to evaluate a router servicing an OC-12 (622 Mbps) link. It is incorrect because the replay would not capture the back-off effect of TCP sources as they detect the congestion in the overloaded OC-12 link. The analysis of the results of such an experiment would be completely misleading, because the traffic represents a set of behaviors of TCP sources that can never occur in practice. For example, the rate of queue overflow would be much larger in the experiment than in a real deployment where TCP sources would react to congestion and reduce the aggregate sending rate below the original 650 Mbps (thereby quickly reducing the number of drops). If we try to estimate a metric related to response time, for example by looking at the duration of each TCP connection, we would obviously see virtually no difference between the original trace and the replay. In reality, the decrease in sending rate by the TCP sources in the congested scenario would result in much longer response times. Thus, valid experiments must preserve the feedback loop in TCP. Traffic generation must be based on some form of *closed-loop* process, and not on simple open-loop packet-level replays.

The fundamental idea of closed-loop traffic generation is to characterize the sources of traffic that

drive the behavior of TCP. In this approach, experimentation generally proceeds by simulating the use of the (simulated or real) network by a given population of users using applications such as ftp or web browsers. Synthetic workload generators are therefore 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 [15]. The critical problem in doing network simulations is then 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 applications using the network. We observe that current workload modeling efforts tend to focus on one or a few specific applications.

Consider models for web browsing as a case in point. The status quo today for web workloads is the set of generators that are based on the web-browsing measurements by Barford, Crovella *et al.* [6, 5, 3] that resulted in the well-known SURGE model and tools. These were later refined and extended by Feldman *et al.* in [8]. While the results of these studies are widely used today, both were conducted several years ago and were based on measurements of a rather limited set of users. 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 and no model that accounts for protocol improvements (*e.g.*, the use of persistent connections in HTTP/v1.1) or newer uses of the web for peer-to-peer file sharing and remote email access.

The most important limitation of current source-level modeling approaches is that they construct application-specific workload models. Given the complexity inherent in this approach (*e.g.*, the effort involved in understanding, measuring, and modeling specific application-layer protocols), it is quite understandable that workload models usually consider only one or a small number of applications. However, few (if any) networks today carry traffic from only one or two applications or application classes. Most links carry traffic from hundreds or perhaps thousands of applications in proportions that vary widely from link to link. (In fact simply determining precisely the mix and traffic volume of applications is a difficult problem for reasons discussed later.)

This issue of application mixes is a serious concern for networking researchers. For example, if one wanted to evaluate the amount of buffering in a router, or a TCP protocol enhancement, *etc.* ., it stands to reason they should consider its impact on the applications that consume the majority of bandwidth on the Internet today and that are projected to do so in the future. It would be natural to consider the performance implications of the scheme on web usage (*e.g.*, the impact on throughput or request-response response times), on peer-to-peer applications, streaming media, other non-interactive applications such as mail and news, and on the ensemble of all applications mixed together. As described in greater detail in Section 2, the majority of previous work in workload modeling has focused on the development of source-level models of single applications. Because of this, there are no models for mixes of networked applications. Worse, the use of analytic (distribution-based) models such as those developed by Paxson [14], and Barford, Crovella, *et al.* of specific TCP applications does not scale to developing workload models of application mixes comprised of hundreds of applications. Typically when constructing workload models, the only means of identifying application-specific traffic in a network is to classify connections by port numbers. 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, one problem with this approach for HTTP is that a number of *applications* (*e.g.*, SOAP) are essentially using port 80 as an access mechanism to penetrate firewalls and middleboxes.

A deeper problem with this approach is that 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, in a study of traffic received from two broadband ISPs by AT&T in 2003 [9], the source (application) of 32-48% of the bytes could not be identified. Similarly, the analyses of backbone traffic in Sprint and

Internet2 networks [10, 4] do not identify between 25% and 40% of the bytes depending on the studied link. 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. This is a very time-consuming process, and doing it for hundreds of applications (or even the top twenty) in network traffic is a daunting task.

In this paper we present a new method for constructing statistically sound workload models from network packet traces that captures the richness in the mix of applications using a given link. The general paradigm of workload modeling and generation we follow is an empirically-based method. One first takes (possibly anonymized) packet header traces of traffic found on network links of interest and uses those traces to construct a model of the applications' uses of the network. This first step relies on *application-neutral* modeling of the source-level behavior in TCP connections that can be applied to the entire mix of application in today's Internet traffic. The model of the traffic is then input to a synthetic workload generator that emulates the behavior of the application(s) in the network simulation or laboratory experiment.

Our approach provides us with the ability to replay the application workload from a real network in a simulation or laboratory network and reproduce critical properties of the packet-level traffic from the real network. We claim the method is a natural and substantial step forward: it is simple to describe, interpret, and implement, but flexible enough to accurately capture a wide variety of existing applications *without knowing what those applications are*. With our method and tools, the process of going from packet traces on a network link to generating a synthetic TCP workload that is statistically equivalent to that observed on the measured link can be reduced *from months to hours*.

Application-Neutral Modeling of TCP Sources

The foundation of our approach to modeling 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

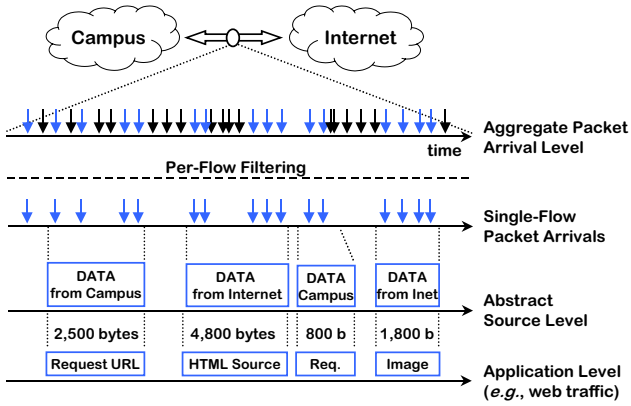

Figure 1: *Network traffic seen at different levels of abstraction.*

the endpoint processes. The idea is illustrated in Figure 1. The traffic on an Internet link can be seen as an aggregate of packets from many different connections. Each connection is driven by an application running on the two end-points. For example, a web connection is used to transport requests for URLs and web objects, such as HTML source code and image files. For TCP applications, arrivals of packets within a flow is a function of the source-behavior of the application and the congestion control and windowing mechanisms of the transport protocol. Each application has a different set of messages and objects that are exchanged between the two end-points. However, there exists a level of abstraction at which all connections are doing nothing more than sending data back and forth and waiting for application events. We believe this *abstract source-level* is the right place for modeling traffic mixes in a manner that is suitable for generating closed-loop traffic.

In our models, the two 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 ap-

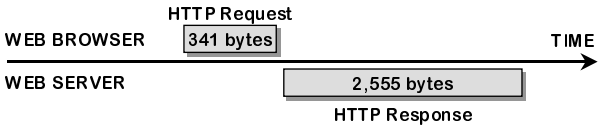

Figure 2: *Pattern of ADU Exchange in an HTTP 1.0 connection.*

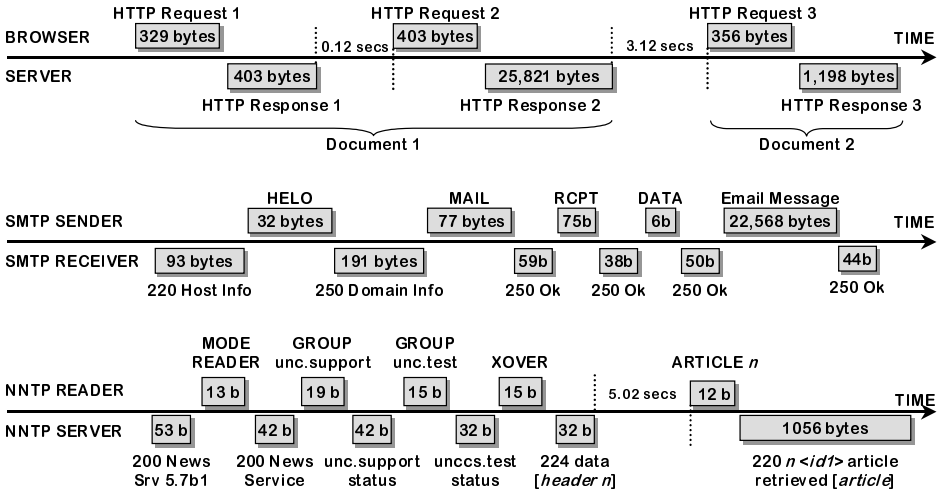

Figure 3: *Pattern of ADU Exchange in three sample connections (from top to bottom: HTTP 1.1, SMTP and NNTP examples).*

plication and, therefore, are (largely) independent of the sizes of the network-dependent data units employed at the transport level and below. For example, the sizes of 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 in Figure 2. 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 (so it is not shown in the figures).

Another common pattern for TCP connections arises from application protocols where there are multiple ADU exchanges between the endpoints of a logical connection. Figure 3 shows three examples of this type of pattern. The top connection diagram shows a persistent HTTP connection, in which three web requests are sent from a browser to a web server and three corresponding web responses (objects) are sent from the server to the browser (for a total of six ADUs exchanged). The first two request/response exchanges correspond to a first document (web page) download, while the last exchange corresponds to a second document download. The diagram also shows two quiet times in this connection. The first one is relatively short (120 milliseconds) while the second one, between the two documents, is much longer (3.12 seconds), so it was probably due to user *user think time* rather than network conditions. Similarly, the SMTP connection in Figure 3 illustrates a sample sequence of data units exchanged by two SMTP servers. Note that in this case most data units are small and correspond to application-level control messages (*e.g.*, the host info message, the initial HELO message, *etc.*) rather than application objects (*e.g.*, the actual email message of 22,568 bytes). The last example is an NNTP connection in which an NNTP reader checks the status of a number of newsgroups and, after a quiet time of 5.02 seconds, requests the content of one article. It is convenient to model this type of pattern as consisting of one or more epochs, where each epoch consists of either 0 or 1 ADU from each endpoint followed by an inter-epoch time interval (or the end of the logical connection). Thus we can model 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.

More formally, we model the source-level behavior in which each TCP connection using a *connec-*

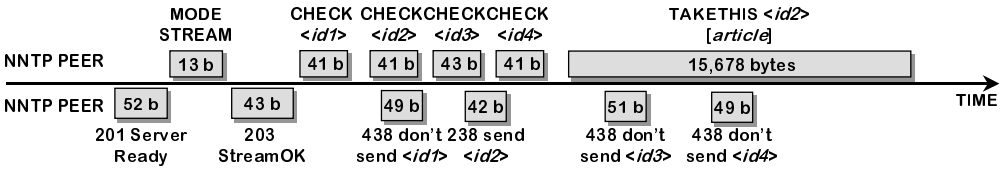

Figure 4: *Pattern of ADU Exchange in data-concurrent NNTP connection (stream mode).*

tion vector $C_i = (e_1, e_2, \ldots, e_n)$ with n epochs. An epoch is a triplet of the form $e_j = (a_j, b_j, t_j)$ that describes the data (a_j, b_j) and quiet time (t_j) parameters of the j'th exchange in a connection. Each a_j captures the amount of data send from the initiator of the connection (*e.g.*, a web browser) to the acceptor (*e.g.*, a web server), and each b_j represents data flowing in the other direction. Using this model, we can succinctly described the connection in Figure 2 as $((341, 2555, \phi))$, where the first ADU, a_1, has a size of 341 bytes, and the second one, b_1 has a size of 2,555 bytes. Similarly, the SMTP connection in Figure 3 can be represented as $((0, 93, 0), (32, 191, 0), (77, 59, 0), (75, 38, 0), (6, 50, 0), (22568, 44, \phi))$. As this last connection vector shows, application protocols sometimes have a single ADU in an epoch (*e.g.*, $a_1 = 0$ for SMTP and $b_1 = 0$ for FTP-data). Given the form of our model, we call it the *a-b-t model* of a connection. Our model captures three essential source-level properties of a TCP connection: data in the "a direction", data in the "b direction" and quiet times t between data units.

A final pattern extends the model to allow for ADU transmissions by the two endpoints to overlap in time (*i.e.*, to be concurrent). Figure 4 shows an NNTP connection between two NNTP peers (servers) in which NNTP's *streaming mode* is used. As shown in the diagram, the article in the last data unit of the initiator side (with a size of 15,678 bytes), coexisted with two data units that were sent from the connector acceptor (two 438 messages). Therefore this connection is said to exhibit *data exchange concurrency*. In contrast, the connections illustrated in Figures 2 and 3 exchanged data units in a sequential fashion. A fundamental difference between these two types of communication is that sequential request/response exchange always take a minimum of one round-trip time. Application designers make use of data concurrency for two primary purposes:

- *Keeping the pipe full*, by making use of requests that overlap with uncompleted responses. This avoids the one round-trip time per epoch price that any request/response exchange must pay, so the connection can be fully utilized.
- *Supporting natural concurrency*, in the sense that some applications do not need to follow the traditional request/response paradigm.

Examples of protocols that attempt to keep the pipe full are the pipelining mode in HTTP, the streaming mode in NNTP, and the BitTorrent and Rsync protocols. Examples of protocols/applications that support concurrency are instant messaging and Gnutella (in which the search messages are simply forwarded to other peers without any response messsage).

For data-concurrent connection, we use a different version of our *a-b-t* model in which the two directions of the connection are modeled independently by two separate connection vectors of the form

$$((a_1, ta_1), (a_2, ta_2), \ldots, (a_{n_a}, ta_{n_a}))$$

and

$$((b_1, tb_1), (b_2, tb_2), \ldots, (b_{n_b}, tb_{n_b}))$$

Using two independent vectors does provides enough detail to capture the two purposes in the last paragraph, since in the first case, one side of the connection completely dominates (so only one of the connection vectors matters), and in the second case, the two sides are completely independent.

From Packet Traces to *a-b-t* Traces

Modeling TCP connections as a pattern of ADU transmissions provides a unified view of Internet information exchanges that does not depend on the specific applications driving each TCP connection. The first step in the modeling process is to acquire

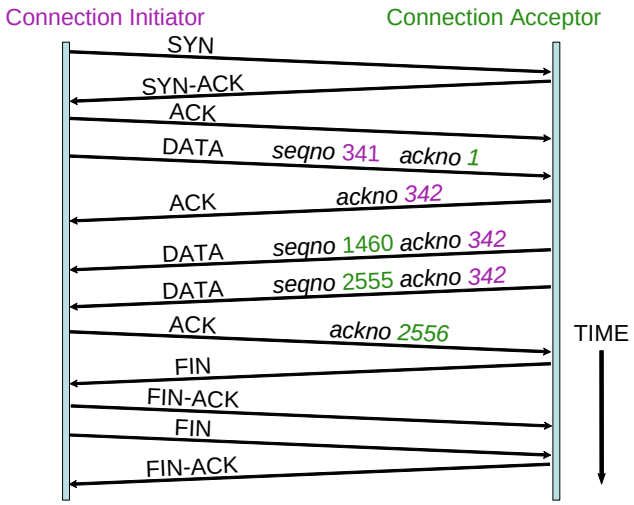

Figure 5: *The exchange in Figure 2 seen at the packet level (lossless example).*

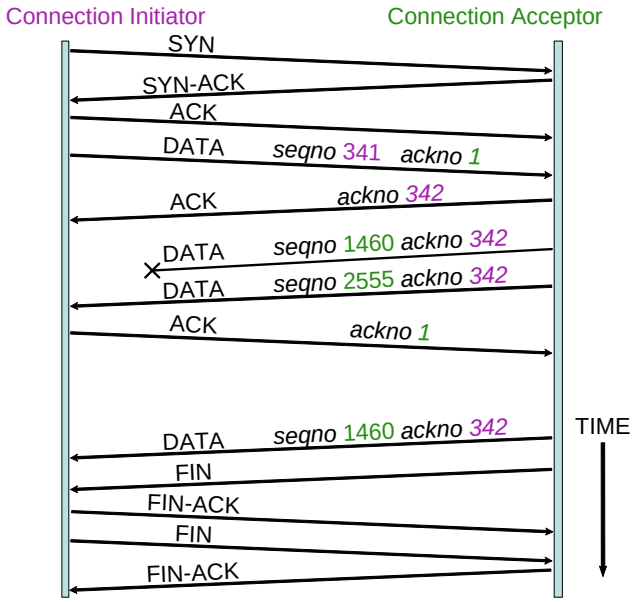

Figure 6: *The exchange in Figure 2 seen at the packet level (lossy example).*

empirical data and process that data to produce the *a-b-t* model. The basic data acquisition technique that we use is passive tracing of TCP/IP packet-headers to capture bidirectional traces. The sample set of packets in a TCP connection is shown in Figure 5 (the source level representation of this connection was shown in Figure 2). The sequence numbers (seqnos) in each packet provide enough information to compute ADU sizes. In the example, we observe a first data packet sent from the initiator to the acceptor with a sequence number for the last byte in the segment of 341. In response to this data packet, first a pure acknowledgment packet (with ackno 342) is sent from the acceptor, followed by two data packets (with the same acknowledgment number). This change in the directionality of data transmission makes it possible to established a boundary between the first data unit a_1, which was transported using a single packet and had a size of 341 bytes, and the second data unit b_1, which was transported using two packets and had a size of 2,555 bytes.

Figure 6 shows a more complicated example, in which the first data packet of the data unit sent from the acceptor is lost somewhere in the network, forcing the acceptor end point to retransmit this packet some time later. Depending on the location of the monitor (before or after the point of loss), the analyzed packet header trace may or may not include the first instance of the packet with end sequence number 1460. If this packet is present in the trace, the analysis program must detect that the third packet is a retransmission, so it should be ignored and the size of the data should be 2,555 bytes. If the lost packet is not present in the trace, the analysis must detect the reordering of packets using their sequence number and also output a size for b_1 of 2,555 bytes.

We have developed a measurement algorithm that can measure application-data unit sizes in the presence of arbitrary reordering and loss. For sequential connections, our starting point is the observation that, despite the possibility of loss and reordering, data in a TCP connection has a unique logical ordering (otherwise, the connection would not be transmitting useful data). For example, the data in the retransmitted segment in Figure 6 logically preceeds the data in the previous data packet, since the sequence number of the retransmitted packet is lower. Using this observation, we can define a total ordering of packets in any TCP connection, and develop an algorithm that will be able to determine ADU sizes using this ordering. Our algorithm, which does not rely on hashing but only on insertion in an ordered list of segments, has a complexity $O(n*W)$, where n is the number of packets and W is the window size of a TCP connection. The algorithm proceeds by reading packets in arrival order and inserting a summary of each packet into a data structure that captures the logical data order-

ing in a TCP connection. Due to the possibility of reordering (that in TCP is constrained to the size of the TCP window), this insertion step takes $O(W)$ time. The algorithm also makes use of the timestamps reported by the trace acquisition tool to compute the inter-epoch quiet times. The timestamp of the SYN packet is used to compute the connection start time, T_i, relative to the beginning of the trace.

The brief description of our measurement algorithm in the previous paragraph does not include any of the many details that must be addressed to perform this type of measurement accurately. For example, Figure 5 and 6 shows "beautified" sequence numbers in order to make the example easier to understand. In reality, the starting sequence numbers of a TCP connection are randomized, so we have to use the sequence numbers of the SYN and FIN packets to determine the boundaries of the connection. Furthermore, the algorithm must detect sequence number wrap-arounds, since TCP sequence numbers are represented using only 32-bits. Another source of difficulties that must be addressed by the measurement algorithm is aborted TCP connections and implementation errors that make TCP behave in non-standard ways.

The logical data ordering mentioned above is not present in data-concurrent connections, such as the one shown in Figure 4. For example, the packet that carried the last b-type data unit may have been sent roughly at the same time as another packet carrying some of the data of the a-type data unit sent from the other side. Sequence numbers will show that these two packet do not acknowledge each other, so it cannot be determined whether the data in one of them was supposed to logically come before the data in the other packet. This observation makes it possible to detect data concurrency.

Formally, the algorithm considers a connection to be concurrent when there exists at least one pair of non-empty TCP packets p and q such that p is sent from the initiator to the acceptor, q is sent from the acceptor to the initiator, and the following two inequalities are satisfied:

$$p.seqno > q.ackno$$

and

$$q.seqno > p.ackno$$

If the conversation between the initiator and the acceptor is sequential, then for every pair of segments p and q, either p was sent after q reached the initiator, in which case $q.seqno = p.ackno$, or q was sent after p reached the acceptor, in which case $p.seqno = q.ackno$. Thus, every non-concurrent connection will be classified as such by our algorithm. Situations in which all the segments in potentially concurrent data exchanges are sent sequentially (purely by chance in sparse, *i.e.*, non-backlogged, connections) are not detected by our algorithm and the connection is modeled as non-concurrent. Note that we detect *concurrent exchanges of data* and not just concurrent exchange of packets in which a data packet and an acknowledgment packet are sent concurrently. In the latter case, the logical ordering of data inside the connection is never broken. Similarly, the simultaneous close mechanism in TCP (in which two FIN packets are sent concurrently) is not considered data concurrency by our algorithm.

Workload Modeling and Generation from *a-b-t* Traces

Once an *a-b-t* trace $\mathcal{T}$ has been obtained, it may be used for workload modeling and generation in a variety of ways. If the goal is to simply reproduce the workload represented by a single packet trace, then one may simply "replay" $\mathcal{T}$ at the socket API with the same sequence of start times that preserves both the order and initiation time of the TCP connections. This is the trace-driven approach we use in this paper. A tool to generate workloads using this source-level "replay" approach is described below.

Another straightforward modeling approach is to derive the distributions for the key random variables that characterize applications at the source level (*e.g.*, distributions of ADU sizes, time values, number of epochs, *etc.*) from values recorded in $\mathcal{T}$. These distributions 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.*, the SURGE model for web browsing). However, the simple structure of the *a-b-t* trace makes it a flexible tool for a broader range of modeling and generation approaches. For example, 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 *a-b-t* representation and 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 connec-

tion according to some model of connection arrivals (perhaps derived from the original packet traces).

Another form of modeling one could use is strongly related to the methods of semi-experiments introduced in [11] but applied at the application level instead of 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 [7]) in order to study the effects of changes in the connection arrival process. Other interesting transforms to consider include replacing the recorded ADU sizes with sizes drawn from analytic distributions (*e.g.*, log-normal) 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. An open question, which we are addressing on our current work, is whether the different transformations of the original trace are still representative.

Trace-driven replay

A workload generator driven by an *a-b-t* trace $\mathcal{T} = \{(T_i, C_i)\}$ will initiate each TCP connection at time T_i, and send and receive data based on the corresponding connection vector C_i, which models the sources using that connection. We assume that the environment in which the program runs has an interface to the transport layer (*e.g.*, sockets) that can be used to initiate the (real or simulated) transmission of application data. For example, in the ns-2 network simulator, workload generating code accesses the transport layer via Agents. TCP Agents closely mirror the implementation of stream sockets in actual operating systems. Workload-generating applications can send data to, and receive data from, Agents in much the same manner as they would with sockets.

Workload generators in laboratory or testbed networks can use the socket interface in real operating systems to send streams of bytes. The results presented in this paper were obtained using a new workload generating tool, `tmix`, which implements the trace replay method in a FreeBSD environment (see Figure 7). Two instances of this program, each running on a machine at the edge of a network, can

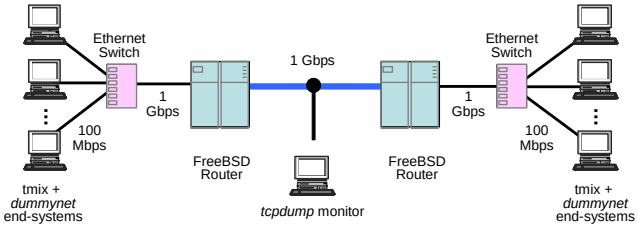

Figure 7: *Experimental network setup (simplified).*

replay an arbitrary *a-b-t* trace by establishing one TCP connection for each connection vector C_i in the trace, with one instance of the program playing the role of the connection initiator and the other program the connection acceptor. To begin, the connection initiator performs a number of socket writes in order to send the number of bytes specified in the first data unit a_1. The other end point will read as many bytes as specified in the data unit a_1. In addition, this first data unit, is used to synchronize the two instances of `tmix`, by including a 32-bit connection vector id in the first four bytes of first data unit. Since this id is part of the content of the first data unit, the acceptor can uniquely identify the connection vector that is to be replayed in this new connection. If a_1 is less than 4 bytes in length, the connection initiator will open the connection using a special port number designated for connections for which the id is provided by the connection acceptor. This approach guarantees that the two `tmix` instances always remain properly synchronized (*i.e.*, they agree on the C_i they replay within each TCP connection) even if connection establishment segments are lost or reordered.

For an example, consider the replay of an *a-b-t* trace containing the connection vector, $C_i = ((329, 403, 0.12), (403, 25821, 3.12), (356, 1198, \phi)$ that corresponds to the TCP connection shown on the top of Figure 3. At time T_i the `tmix` connection initiator establishes a new TCP connection to the `tmix` connection acceptor. The initiator then 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 sequential replay of connection vectors at pre-

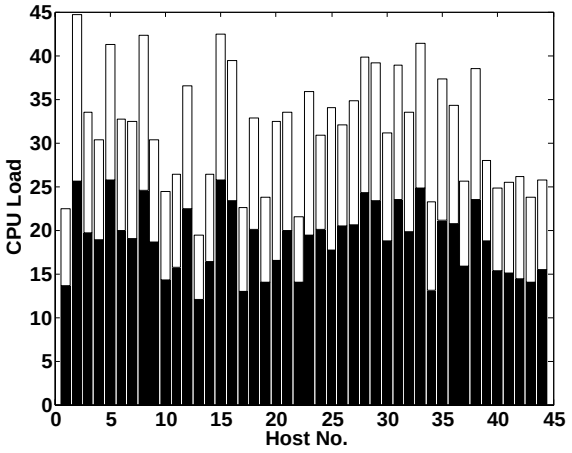

Figure 8: *CPU loads during experiment (average in black and maximum in white).*

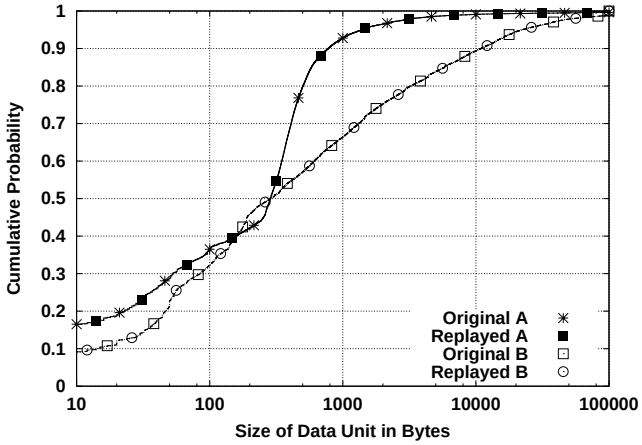

Figure 9: *ADU sizes for Abilene Trace (body).*

scribed start times raises a number of implementation issues and challenges. The scalability issue is particularly important for laboratory environments where a relatively small set of hosts (on the order of 100) is being used to generate traffic corresponding to a much larger number of active connections (on the order of 10,000). The first step in trace replay is to divide a trace into non-overlapping, interleaved subtraces. During workload generation, the connections within a particular subtrace are implemented by a single host, so the number of subtraces is equal to the number of available hosts. The detailed selection of the subtraces will depend on the load balancing strategy, and the speed of the host machines. Our experience with the experiments reported in this paper showed that a simple round-robin assignment of connection vectors to machines performed well.

Case Study: Replay of Abilene Backbone Traffic

In this section we examine the results of performing a *source-level trace replay* of the traffic in a 2-hour trace from the Abilene backbone (Internet2). This trace is publicly available from the trace repository at the National Laboratory of Applied Networking Research (NLANR) [13]. The trace is bidirectional and was collected on the OC-48 link between Indianapolis and Cleveland in August, 2002, using a DAG monitor [16].

The first step of our approach, after acquiring a packet header trace, is to obtain a subtrace consisting of all the packets from all the TCP connections to be included in the workload generation. For the experiment in this paper, the subtrace 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 a FIN or a RST packet. This eliminates only those connections that were in progress when the packet trace began and ended. In the remainder of the paper the phrase *Abilene trace* will refer to the subtrace derived according to the above description. We also refer to this subtrace as the Abilene *original* trace, $\mathcal{T}_h$. The trace measured 158.2 million packets flowing from Cleveland to Indianapolis (128.5 GB of IP data), and 160.5 million packets flowing in the opposite direction (125.9 GB of IP data), that were part of 2.44 million TCP connections.

The second step of our approach is to derive an *a-b-t* trace, $\mathcal{T}_c$, from the subtrace of packet headers using the process described in the third section of this paper. The result is a collection of *a-b-t* connection vectors and their start times. We then use $\mathcal{T}_c$ to generated traffic in a laboratory with the trace-driven generator `tmix` described in the previous section. Instances of the `tmix` generator are used to replay the a-b-t connection vectors in a closed-loop manner. During the experiment, we can use a pair of monitor machines running `tcpdump` to collect a new packet header trace $\mathcal{T}_h^*$, that we will refer to as the Abilene *replay* trace. We can then compare the var-

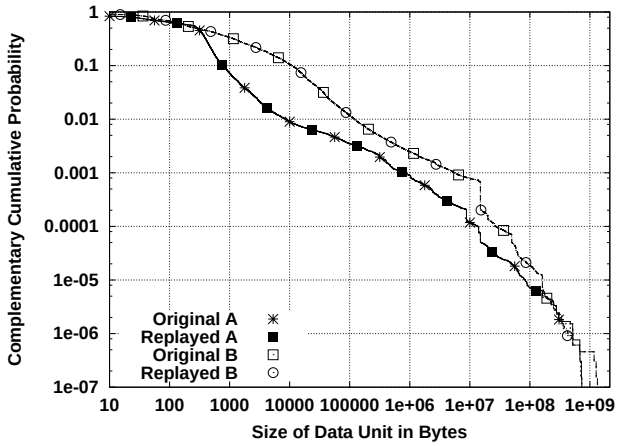

Figure 10: *ADU sizes for Abilene Trace (tail).*

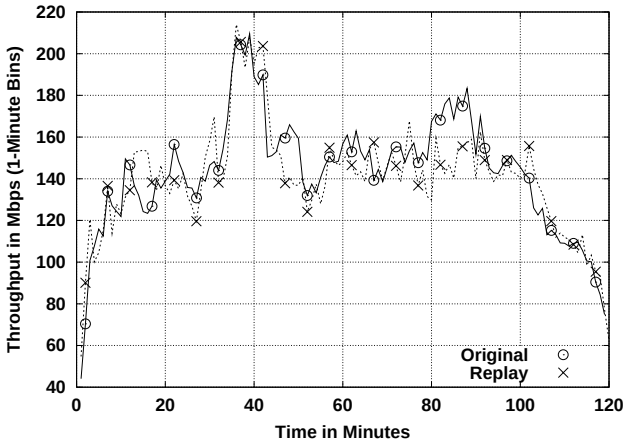

Figure 11: *Throughput (1-minute-bins) on the Cleveland to Indianapolis direction.*

ious properties of $\mathcal{T}_h$ and $\mathcal{T}_h^*$, and study the differences between original and synthetic traffic.

The replay was performed in the laboratory configuration shown in Figure 7. The network consists of approximately 50 Intel-processor machines running FreeBSD 4.5. Forty-four of these machines (22 on each side of the configuration) execute the trace-driven workload generator, `tmix`. The generating machines have 100 Mbps Ethernet interfaces and are attached to switched VLANs on Gigabit Ethernet switches. At the core of this network are two 1.4 GHz Intel-processor server-class machines (PCI-X busses) acting as routers (IP-forwarding enabled) with drop-tail FIFO queues. The router machines have 1 Gbps interfaces to the Ethernet switches and a point-to-point Gigabit Ethernet between the routers. For all the experiments reported here, there is no congestion on any router or switch interface and no losses were recorded at these interfaces. We also verified that it is unlikely that there were any CPU (see Figure 8) or other resource constraints on generators.

So that we can emulate TCP connections that traverse a longer network path than the one in our lab, we use a locally-modified version of `dummynet` [17] to configure artificial in-bound and out-bound packet delays on the workload generating machines. These delays allow us to emulate different round-trip times on each TCP connection (thus giving per-flow delays). Our version of `dummynet` delays all packets from each flow by the same randomly-chosen delay for that flow. In many of the experiments reported in this section, the distribution of RTT values across all TCP connections is an important parameter and the values used are described for each experiment. The version of TCP used in these experiments is New Reno without selective acknowledgments. [1]

Sample Comparison

The comparison of the Abilene original and replay traces can provide interesting insights on the nature of synthetic traffic and help validate the traffic generation approach. An obvious way of comparing the traces is to study the connection vectors in the original trace (*i.e.*, those derived from $\mathcal{T}_h$) and those of the replay trace (*i.e.*, those derived from $\mathcal{T}_h^*$ using the same measurement tool). Figure 9 compares the body of the distributions of a and b data unit sizes, while Figure 10 compares the tails of the same distributions. One interesting feature of these distributions is that the distribution of a sizes is considerably lighter in the body of the distribution than the distribution of b sizes. This confirms our expectation that a units are more likely to be small because they are usually requests (*e.g.*, as in HTTP) and the b units (the responses) are more likely to be larger. The tail of the distributions appear to be consistent with a heavy-tailed distribution. Note also that the distributions measured from the replay are almost identical to the original ones.

A similar analysis of the distribution of the number

[1] Other parameters: TCP was configured to use an *ssthresh* of 4 MB, RFC 1323 was disabled, delayed ACKs (up to 100 milliseconds) were enabled, ECN was disabled, send space was 32K and the receiver maximum window was 17,520 bytes.

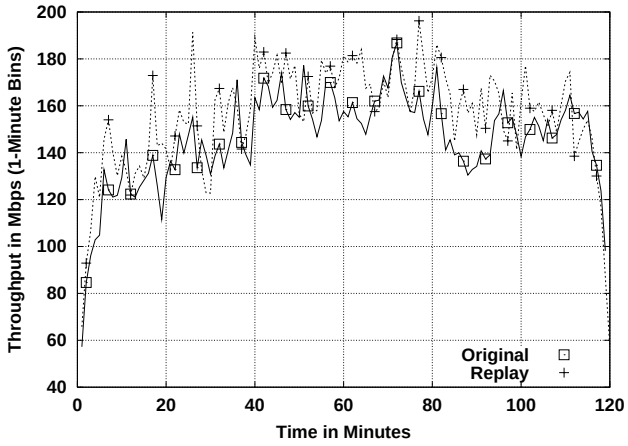

Figure 12: *Throughput (1-minute-bins) on the Indianapolis to Cleveland direction.*

of epochs (not shown) reveals that 59% of the connections had a single epoch and that only 5% of the connections had more than 10 epochs. Data transmission connections with more than one epoch tends to be much more sparse in time than those with a single epoch, so these connections stress TCP in a way that is very different from that of a bulk transfer. The analysis of the inter-epoch times revels that 35% of the connections have quiet times longer than 10 seconds. This also contributes to making the arrival of packet in a TCP connections more sparse and make congestion control dynamics less important. As in the previous comparison of data unit sizes, the replay distributions are almost identical to those of the original trace, demonstrating the feasibility of generating TCP workloads using the *a-b-t* model and the accuracy of the replay tool (`tmix`).

In our work, we have also compared original and replayed traffic at the network level (rather than at the source-level as in the previous paragraphs). The most obvious analysis at this level is a plot of the time-series of throughput values for the duration of the experiment. Figures 11 and 12 show this type of comparison for throughput in the original and the replay of the Abilene trace. For the Cleveland to Indianapolis path, the replay appears to track the fluctuations in load reasonably well. Notice for example how the replay is able to reproduce the sustained burst between 35 and 45 minutes. Similarly, the other direction of the replay is close to the original, although the replay is somewhat more bursty. (Note the sharp peaks in the first 30 minutes of the replay.)

We have studied other properties of the original and the generated traffic such as:

- The number of simultaneously active connections per unit interval (any mechanism that requires per-flow state, such as NetFlow monitoring [12], is strongly affected by this property).
- The distribution of packet sizes.
- The distribution of flow durations.
- The distribution of flow rates.
- The arrival process of TCP flows.
- The arrival process of TCP packets and its long-range dependence (we followed the wavelet analysis method described in [1] for this part of our study).

Our results (not presented here for brevity) show that the source-level trace replay approach can accurately reproduce the properties of real traffic. In some cases, it is important to derive distributions of some network parameters (round-trip times, receiver window sizes, and access capacities) to achieve accurate reproduction. Accordingly, we have extended our measurement tools to extract these distributions from the original trace of packet headers.

Summary and Conclusion

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. In networking, an overlooked aspect of simulation methodology is the problem of generating realistic synthetic workloads. We have developed an empirically-based approach to synthetic 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 workload generators (such as `tmix`) to replay the connections and reproduce the application-level behaviors observed on the original network.

We believe this approach to source-level modeling, and the `tmix` generator, are contributions to the art of network evaluations, because of their ability to automatically generate valid workload models representing all of the TCP applications present in a network

with no a priori knowledge of their existence or identity. Our work therefore serves to demonstrate that researchers and performance analysts 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 and laboratory or testbed experiments.

Acknowledgments

We gratefully acknowledge the use of traces from the NLANR Measurement and Network Analysis Group (NLANR/MNA) which is supported by the National Science Foundation cooperative agreement nos. ANI-0129677 (2002) and ANI-9807479 (1998). This work was supported in parts by the National Science Foundation (grants CCR-0208924, ANI-0323648, and EIA-0303590), Cisco Systems Inc., and the IBM Corporation. Félix Hernández-Campos was partially supported by a fellowship from the Computer Measurement Group (CMG).

References

[1] P. Abry and D. Veitch. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, 44(1):2–15, 1998.

[2] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, April 1999.

[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. of the ACM SIGMETRICS*, pages 151–160, 1998.

[4] Internet 2 Consortium. Internet2 Netflow Weekly Report, `http://netflow.internet2.edu/weekly/20040621`, June 2004.

[5] Mark E. Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[6] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of www client-based traces. Technical report, Boston University, 1995.

[7] A. Feldmann. Characteristics of TCP connection arrivals. In K. Park and W. Willinger, editors, *Self-Similar Network Traffic and Performance Evaluation*. Wiley, New York, 2000.

[8] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *SIGCOMM*, pages 301–313, 1999.

[9] Alexandre Gerber, Joseph Houle, Han Nguyen, Matthew Roughan, and Subhabrata Sen. P2P: The gorilla in the cable. In *National Cable & Telecommunications Association(NCTA) 2003 National Show*, Chicago, IL, June 2003. 8-11.

[10] Sprint Advanced Technology Laboratory (IP Group). IP monitoring project: Data management system, 2004. `http://ipmon.sprintlabs.com`.

[11] N. Hohn, D. Veitch, and P. Abry. Does fractal scaling at the ip level depend on tcp flow arrival processes? In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2002.

[12] Cisco Systems Inc. Cisco IOS Software Netflow, `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[13] NLANR Measurement and Network Analysis Group. Trace IPLS-CLEV-20020814-090000-0 (Abilene-I data set). `http://pma.nlanr.net/Traces/long/ipls1.html`.

[14] Vern Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, 1994.

[15] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[16] The DAG Project. `http://dag.cs.waikato.ac.nz`.

[17] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.