GENERATION AND VALIDATION OF EMPIRICALLY-DERIVED TCP APPLICATION WORKLOADS

Félix Hernández-Campos

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2006

Approved by:
Advisor: Kevin Jeffay
Reader: F. Donelson Smith
Reader: Ketan Mayer-Patel
Reader: J. Steve Marron
Reader: Andrew Nobel
Reader: Jan Prins

ABSTRACT

FÉLIX HERNÁNDEZ-CAMPOS: Generation and Validation of
Empirically-Derived TCP Application Workloads.
(Under the direction of Kevin Jeffay)

This dissertation proposes and evaluates a new approach for generating realistic traffic in networking experiments. The main problem solved by our approach is generating closed-loop traffic consistent with the behavior of the entire set of applications in modern traffic mixes. Unlike earlier approaches, which described individual applications in terms of the specific semantics of each application, we describe the source behavior driving each connection in a generic manner using the a-b-t model. This model provides an intuitive but detailed way of describing source behavior in terms of connection vectors that capture the sizes and ordering of application data units, the quiet times between them, and whether data exchange is sequential or concurrent. This is consistent with the view of traffic from TCP, which does not concern itself with application semantics.

The a-b-t model also satisfies a crucial property: given a packet header trace collected from an arbitrary Internet link, we can algorithmically infer the source-level behavior driving each connection, and cast it into the notation of the model. The result of packet header processing is a collection of a-b-t connection vectors, which can then be replayed in software simulators and testbed experiments to drive network stacks. Such a replay generates synthetic traffic that fully preserves the feedback loop between the TCP endpoints and the state of the network, which is essential in experiments where network congestion can occur. By construction, this type of traffic generation is fully reproducible, providing a solid foundation for comparative empirical studies.

Our experimental work demonstrates the high quality of the generated traffic, by directly comparing traces from real Internet links and their source-level trace replays for a rich set of metrics. Such comparison requires the careful measurement of network parameters for each connection, and their reproduction together with the corresponding source behavior. Our final contribution consists of two resampling methods for introducing controlled variability in network experiments and for generating closed-loop traffic that accurately matches a target offered load.

ACKNOWLEDGMENTS

First of all, I must thank Kevin Jeffay and Don Smith for their guidance and encouragement throughout my doctoral program. Their patience and friendship have been invaluable all these years. I also thank them, together with other faculty and student members of the Distributed and Real-Time Systems group (DiRT), for building a phenomenal infrastructure for Internet measurement and experimental networking research. DiRT students have greatly contributed to my doctoral experience, most especially Jay Aikat and David Ott.

My committee members and other collaborators have contributed tremendously to my efforts. I am specially in debt with Steve Marron and Andrew Nobel, who have greatly enriched the statistical side of my work. In this regard, being part of SAMSI's "Network Modeling for the Internet" program and of the inter-disciplinary Internet study group at UNC gave me superb opportunities to widen my understanding of Internet research. I must also thank UNC's Department of Computer Science as whole, including faculty, students and staff, for creating an outstanding research and teaching environment. Overall, my years at UNC were an incredibly positive experience.

I thank the National Science Foundation, IBM, Cisco, Intel, Sun Microsystems and others for supporting this work. I am specially grateful to the Computer Measurement Group (CMG) for their doctoral fellowship.

Finally, I thank my family for their support. Their constant example of hard-work, and their respect for intellectual endeavors has motivated me during my entire life. My wife's help with the editing of this manuscript was invaluable, as was her constant encouragement during my graduate studies. More than anybody else, my parents gave me my passion for knowledge, and it is to them that I dedicate this doctoral dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACK	Positive acknowledgment TCP segment
ADU	Application Data Unit
API	Application Programming Interface
AQM	Active Queue Management
BGP	Border Gateway Protocol
BPF	Berkeley Packet Filter
C.I.	Confidence Interval
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
DAG	Data Acquisition and Generation
FIFO	First-In First-Out
FIN	TCP control flag indicating “no more data from sender”
FTP	File Transfer Protocol
GB	Gigabyte
GPS	Global Positioning System
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
I/O	Input/Output
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IRC	Internet Relay Chat
ISP	Internet Service Provider
K-S	Kolmogorov-Smirnov test
KB	Kilobyte
Kpps	Kilo packet per second
LRD	Long-Range Dependence
MB	Megabyte
MIME	Multipurpose Internet Mail Extensions
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit

Mbps	Megabit per second
NNTP	Network News Transfer Protocol
OSTT	One-Side Transit Time
PMA	Passive Measurement and Analysis
Q-Q	Quantile-Quantile
RED	Random Early Detection
RFC	Request For Comments
RST	TCP control flag indicating “connection reset”
RTT	Round-Trip Time
SMTP	Simple Mail Transfer Protocol
SSH	Secure Shell
SYN	Synchronize TCP control segment
SYN-ACK	Positive acknowledgement of SYN segment
TCP	Transport Control Protocol
UDP	User Datagram Protocol
UNC	University of North Carolina at Chapel Hill
URL	Universal Resource Locator

CHAPTER 1

Introduction

As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.

— Albert Einstein (1879–1955)

Humankind cannot stand very much reality.

— T. S. Elliot (1888–1965)

Research in networking has to deal with the extreme complexity of many layers of technology interacting with each other in frequently unexpected ways. As a consequence, there is a broad consensus among researchers that purely theoretical analysis is not enough to demonstrate the effectiveness of network technologies. More often than not, careful experimentation in simulators and network testbeds under controlled conditions is needed to validate new ideas. Every researcher therefore faces, at some point or another, the need to design realistic networking experiments, and synthetic network traffic is a foremost element of these experiments. Synthetic network traffic represents not only the workload of a computer network, but also the direct or indirect target of any optimization. For instance, congestion control research focuses on preserving as much as possible the ability of a network to transfer data in the face of overload. Therefore, evaluating a new congestion control mechanism in a transport protocol such as the Transport Control Protocol (TCP) [Pos81] usually requires constructing experiments in which a number of network hosts exchange data using this protocol in an environment with one or more saturated links. The value of the new mechanism is then expressed as a function of the performance of these data exchanges. For example, the new mechanism may be optimized for achieving a higher overall throughput or a more fair allocation of bandwidth.

A fundamental insight, which provides the main motivation for this dissertation, is that the characteristics of synthetic traffic have a dramatic impact on the outcome of networking experiments. For example, a new mechanism that improves the throughput of bulk, long-lasting file transfers in a congested environment may not improve and may even degrade the response time of the small data exchanges in web traffic. This was precisely the case of Random Early Detection (RED), an Active Queue Management (AQM) mechanism. The original analysis by Floyd and Jacobson [FJ93a] clearly demonstrated the benefits of RED over the basic First-In First-Out (FIFO) queuing mechanism for bulk transfers. In this study, RED queues were exposed to a small number (2–4) of large file transfers. However, a later experimental study by Christiansen *et al.* [CJOS00] showed that this first AQM mechanism degraded the performance of web traffic in highly congested environments. In contrast to the original evaluation, web traffic mostly consists of a very large number of small data transfers, which create a very different workload. The emergence of the web clearly changed the nature of Internet traffic, and made it necessary to revisit existing results obtained under different workloads. The systematic evaluation of network mechanisms must therefore include experiments covering the wide range of traffic characteristics observed on Internet links. It is critical to provide the research community with methods and tools for generating synthetic traffic as representative as possible of this range of characteristics.

The concept of *source-level modeling* introduced by Paxson and Floyd [PF95] constitutes a major influence on this dissertation. These authors advocated for building models of the behavior of Internet applications (*i.e.*, the sources of Internet traffic), and generating traffic in networking experiments by driving network stacks with these application models. The main benefit of this approach is that traffic is generated in a *closed-loop* manner, which fully preserves the fundamental feedback loop between network endpoints and network characteristics. For example, a model of web traffic can be used to generate traffic using TCP/IP network stacks, and the generated traffic will properly react to different levels of congestion in networking experiments. In contrast, *open-loop* traffic generation is associated to models of the packet arrivals on network links, and these models are insensitive to changes in network conditions, and tied to the original conditions under which they were developed. This makes them inappropriate for experimental studies that change these conditions.

The main motivation of our work is to address one important difficulty with source-level modeling. In the past, source-level modeling has been associated with characterizing the behavior of individual applications. While this approach can result in high-quality models, it is a difficult process that requires a large amount of effort. As a consequence, only a small number of models is available, and they are often outdated. This is in sharp contrast to the traffic observed in most Internet links, which is driven by rich *traffic mixes* composed of a large number of applications. Source-level modeling of individual applications does not scale to modern traffic mixes, making it very problematic for networking researchers to conduct representative experiments with closed-loop traffic.

This dissertation presents a new methodology for generating network traffic in testbed experiments and software simulations. We make three main contributions. First, we develop a new source-level model of network traffic, the *a-b-t model*, for describing in a generic and intuitive manner the behavior of the applications driving TCP connections. Given a packet header trace collected at an arbitrary Internet link, we use this model to describe each TCP connection in the trace in terms of data exchanges and quiet times, without any knowledge of the actual semantics of the application. Our algorithms make it possible to efficiently derive empirical characterizations of network traffic, reducing modeling times from months to hours. The same analysis can be used to incorporate network-level parameters, such as round-trip times, to the description of each connection, providing a solid foundation for traffic generation. Second, we propose a traffic generation method, *source-level trace replay*, where traffic is generated by replaying the observed behavior of the applications as sources of traffic. This is therefore a method for generating entire traffic mixes in a *closed-loop* manner. One crucial benefit of our method is that it can be evaluated by directly comparing an original trace and its source-level replay. This makes it possible to systematically study the *realism* of synthetic traffic, in the terms of how well our description of the connections in the original traffic mix reflects the nature of the original traffic. In addition, this kind of comparison provides a means to understand the impact that the different characteristics of a traffic mix have on specific traces and on Internet traffic in general. Third, we propose and study two approaches for introducing variability in the generation process and scaling (up or down) the level of traffic load in the experiments. These operations greatly increase the flexibility of our approach, enabling a wide range of experimental investigations conducted using our traffic generation method.

1.1 Abstract Source-Level Modeling

This dissertation presents a methodology for generating synthetic network traffic that addresses some of the main shortcomings of existing techniques. Figure 1.1 illustrates the levels of detail at which Internet traffic can be studied, providing a good starting point for framing our discussion. We focus on the traffic on a single Internet link, such as the one between the University of North Carolina at Chapel Hill (UNC) and the Internet. We can study the traffic in this link at different levels of detail. The top-most time-line represents traffic observed in the link between UNC and the Internet as a sequence of packet arrivals. This level of detail is known as the aggregate packet arrival level. Here packets from many different connections were interleaved creating a complex arrival process in the network link. In general, TCP traffic accounts for the vast majority of the packets on Internet links (usually between 90%

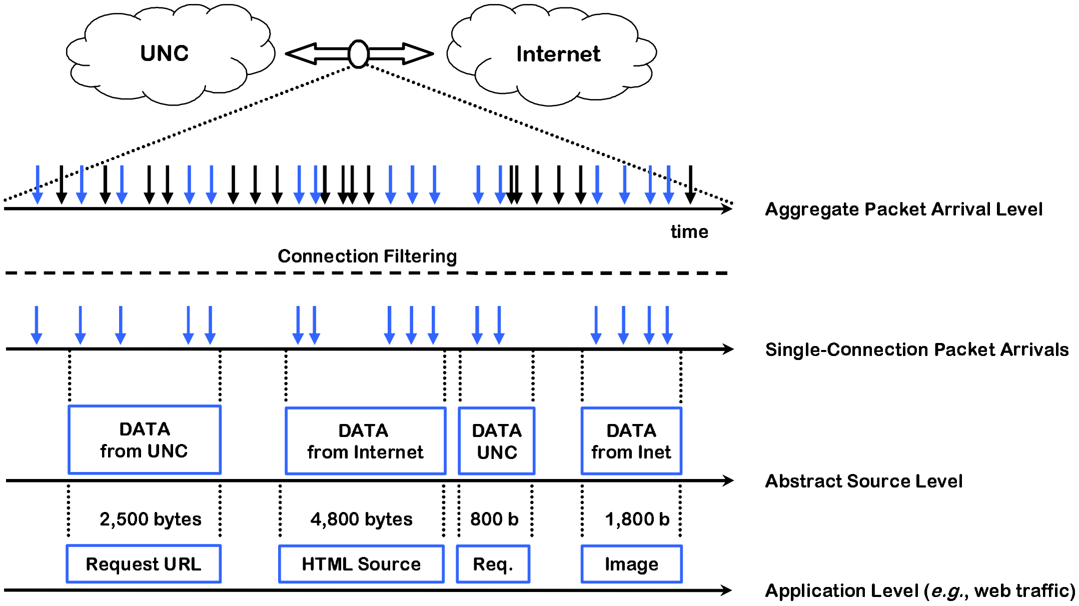


Figure 1.1: Network traffic seen from different levels.

and 95%), which justifies our focus on TCP in this work. The second time-line depicts the packet arrivals that belonged to a single TCP connection. These packets were used to send data back and forth between two network endpoints, one located at UNC, and the other one somewhere on the Internet. The sources of these data are applications running on the endpoints, which rely on the packet switching service provided by the Internet to communicate. Prominent examples of these applications are the World Wide Web, email, file sharing, *etc.* Hundreds of different applications are commonly found on Internet links. The traffic observed at an Internet link is therefore the result of multiplexing the communication of a large number of endpoints driven by a wide range of applications. This dissertation considers the problem of generating traffic in networking experiments that preserves both the aggregate-level and the connection-level properties of traffic observed in a real network link. Note that we restrict ourselves to this most basic form of the problem where only a single link is considered both for observing traffic and for reproducing it in networking experiments. Our findings can certainly be applied to a broader context, *e.g.*, multiple links along a path following the "parking lot topology" [PF95], links in an ISP, *etc.*, but we choose to keep to this problem in its most essential form throughout this dissertation.

As mentioned before, every connection on the Internet is driven by an application exchanging data between two endpoints. It is therefore possible to examine traffic at a higher-level, where the communication is described in terms of *application data units* (ADUs) rather than network packets. This application level is illustrated in the bottom time-line of Figure 1.1, which reveals that the source of the packets in the second time-line was the exchange of data between a web browser and a web server using a TCP connection. The time-line shows a first ADU of 2,500 bytes, which carried a request for an HTML page. The way the data is organized within this ADU and its meaning is given by the specification of the HyperText Transfer Protocol (HTTP) [FGM$^+$97], which standardizes the exchange of data between web browsers and web servers. The time-line shows a second ADU, sent by the web server to the web browser in response to the first ADU. It carried the actual HTML source code of the page requested by the browser. Its size was 4,800 bytes, which included not only the HTML source code but also an appropriate HTTP header. The time-line shows another pair of ADUs that also corresponded to an HTTP request and an HTTP response, which this time carried an image file. Each ADU is associated to one or more packets in the second time-line. The amount of data in these ADUs and its meaning was decided by the application, while the actual number of packets, their sizes, the need for retransmissions, *etc.*, were decided by lower layers (transport and below).

The application level provides the starting point for the traffic modeling and generation methodology developed in this dissertation. Our approach to traffic generation relies on the notion of *source-level modeling*, advocated by Paxson and Floyd [FP01]. Rather than directly generating packets according to some trace or some packet arrival model, source-level modeling involves simulating the behavior of the applications running on the endpoints and allowing lower layers to control the actual exchange of packets. For example, generating traffic with a source-level model of web traffic means to simulate web browsers and web servers according to statistical models of web page sizes, the durations of user think times and other source-level parameters [Mah97, BC98, SHCJO01].

Modeling traffic at the source level produces descriptions of traffic that are mostly independent of the underlying protocols and network conditions, so they can be used to drive traffic generation in experiments that modify these same protocols and conditions. For this reason, source-level models are also known as *network-independent model*. For example, the size of an HTML page carried in a TCP connection does not change with the degree of congestion (it always has the same number of characters). Therefore, its size is a network-independent property. Lower-level descriptions of traffic, such as characterizations of packet arrivals, are *network dependent*. For example, the rate at which the packets of a TCP connection arrive decreases as the degree of congestion increases, since TCP uses a congestion control algorithm that decreases the sending rate as the loss rate increases. Also, packet losses force TCP endpoints to perform retransmissions. This means that the transmission of the same amount of data at the source-level (*e.g.*, an HTML page) at different times may require different

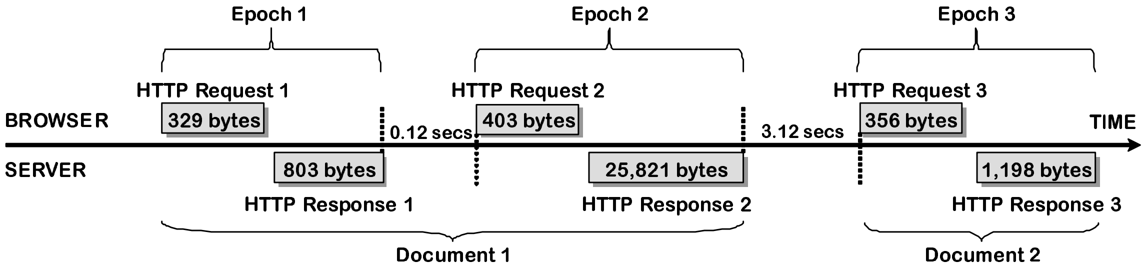


Figure 1.2: An a-b-t diagram illustrating a persistent HTTP connection.

numbers of packets to be transferred, depending on the number of lost packets. A source-level model describes the sizes of ADUs, but not the times at which a connection should lower its sending rate or retransmit a packet. For this reason, the same model can be used to generate traffic under different network conditions, such as low and high levels of congestion. Endpoints generating traffic using these models are able to adapt to each specific set of network conditions in the experiments. This preserves the fundamental feedback loop that exists between endpoints and network conditions. For this reason, this type of traffic generation is said to be *closed-loop*. On the contrary, traffic generated according to lower level models is necessarily *open-loop*. For example, *tcpreplay* [tcpb] can be used to reenact the sending of every packet recorded in a trace, which results in open-loop traffic that is insensitive to the underlying network conditions. This traffic is inappropriate for experiments where network conditions are important, such as the evaluation of congestion control mechanisms.

In the past, source-level modeling has been considered a synonym of application modeling, so researchers have developed a number of application-specific models including models for web traffic, file transferring and other individual applications. This approach is good if one is interested in the traffic generated by a single application (or by a handful of applications). However, if one is interested in realistic traffic mixes, application-specific traffic modeling has some important shortcomings. The first problem is that application specific modeling does not scale well to the large number of applications that form contemporary traffic mixes. For example, the weekly traffic report from Internet2 [Con04] collects separate statistics for more than 80 different applications that make up Internet2 traffic. Using existing technology, it is simply too time-consuming to develop and populate individual models for each application. Moreover, even if we had the resources to examine the behavior of all applications, many applications use proprietary protocols, so painstaking reverse engineering is needed to understand and model their behavior. In addition, Internet traffic evolves quickly, since new applications and improved versions of the existing ones appear very frequently.

This dissertation proposes a more general solution to the source-level modeling and the traffic generation problems. We develop an *abstract model* of network data exchange wherein each connection is described independently of the semantics of the application initiating the connection. This idea is illustrated in the third time-line of Figure 1.1. Here the communication is described in generic terms, simply as a sequence of ADU exchanges between the two endpoints of the TCP connection, without attaching any meaning to the ADUs. Other generic characteristics of traffic include the direction in which the ADUs are sent, from the connection initiator or from the connection acceptor, and the duration of quiet times between ADUs, which are due to user behavior and processing times. These characteristics can generally be used to describe the behavior of any specific application. For example, the ADUs of web traffic are HTTP requests and responses, while the inter-ADU times are user think times and server processing times. The crucial observation is that the sizes of ADUs and the times between them can be measured from the packet traces of two connections *without* knowledge of the behavior of the application driving the connection. This makes it possible to construct a source-level description of the entire set of connections observed in a measured link, instead of only the connections driven by one or a few well-known applications. Any trace of packets traversing a network link can be transformed into an abstract source-level trace, without examining the payload of the packets and without instrumenting the endpoints.

Our approach to source-level modeling results in an abstract representation of a TCP connection using a notation that we call an *a-b-t connection vector*. We also refer to this idea as the *a-b-t model*, in the sense that it provides a mental model for understanding network traffic at the source level, rather than in the sense of a mathematical or statistical model[1]. The term a-b-t is descriptive of the basic building blocks of this model: *a-type* ADUs (a's), which are sent from the connection initiator to the connection acceptor, *b-type* ADUs (b's), which flow in the opposite direction, and quiet times (t's), during which no data segments are exchanged. We will make use of these terms to describe the source-level behavior of TCP connections throughout this dissertation.

Our a-b-t model has a sequential version and a concurrent version. The sequential version applies to connections where the endpoints follow a strict order in their exchange of ADUs. In this version, a TCP connection is described by a vector of epochs $(e_1, e_2, \ldots, e_n)$. Each epoch has the form $e_j = (a_j, ta_j, b_j, tb_j)$, where a_j is the size of an ADU sent from the connection initiator to the connection acceptor, b_j is the size of an ADU sent in the opposite direction, and ta_j and tb_j are inter-ADU quite

[1] Our a-b-t model provides however a good foundation for developing mathematical and statistical models of traffic at the source-level. This dissertation consistently follows a non-parametric approach to traffic modeling. The only exception is the Poisson Resampling method presented in Chapter 7, for which we also offer a more powerful non-parametric alternative, block resampling.

times (during which the endpoints are idle). We call this representation of source-level behavior a *sequential connection vector*. For example, the connection illustrated in Figure 1.2 is represented as

$$((329, 0, 403, 0.12), (403, 0, 25821, 3.12), (356, 0, 1198, 15.3))$$

using the sequential a-b-t model. This connection has three epochs, each carrying one HTTP request/response pair. The first epoch has an ADU a_1 of size 329 bytes, which was sent from the connection initiator (a web browser) to the connection acceptor (a web server), and an ADU b_1 of size 803 bytes, which was sent in the opposite direction. We also observe some quiet times between the ADUs, such tb_2, which had a duration of 3.12 seconds. While Figure 1.2 includes labels for HTTP requests, responses and documents, our a-b-t notation is completely generic.

We consider this TCP connection sequential because only one endpoint sent data to the other one at any point in the lifetime of the connection. It is important to iterate that an ADU is *not* a TCP segment (*i.e.*, TCP packet), but an application message that is independent of its actual network representation as a link-level packet. As such, an ADU can be of arbitrary size, like the smaller $a_1 = 329$ bytes and the larger $b_2 = 25,821$ bytes in the previous example. The transferring of a_1 would usually involve a single TCP segment, but it is also possible that this segment gets duplicated, or lost and then retransmitted. In this case, the TCP endpoint sending a_1 would result in the generation of two or more segments carrying this ADU. Our notation would still describe this part of the TCP connection as a single 329-byte ADU, and not as the sequence of TCP segments used to transfer the data. Similarly, transferring $b_2 = 25,821$ bytes requires a minimum of 18 TCP segments in a path without loss and with a regular Maximum Segment Size (MSS) of 1,460 bytes (the one derived from Ethernet's Maximum Transmission Unit (MTU) of 1,500 bytes, after subtracting 20 bytes for the IP header and 20 bytes for the TCP header). It may require many more segments in a lossy environment, or in a path with a lower MTU. However, these details are irrelevant at the abstract source level, where b_2 captures the need of one of the endpoints to send 25,821 bytes of data, and this need is independent of the way in which the data is transferred by the network. Our modeling is therefore network-independent, which makes it suitable for generating closed-loop traffic.

While most TCP connections are driven by applications that follow a sequential pattern of ADU exchanges, we can also find cases in which the two endpoints send data to each other at the same time. This is illustrated in Figure 1.3 using a BitTorrent [Coh03] connection, where we can see ADUs whose transmission overlaps in time (*i.e.*, the ADUs are exchanged concurrently). This pattern is certainly less

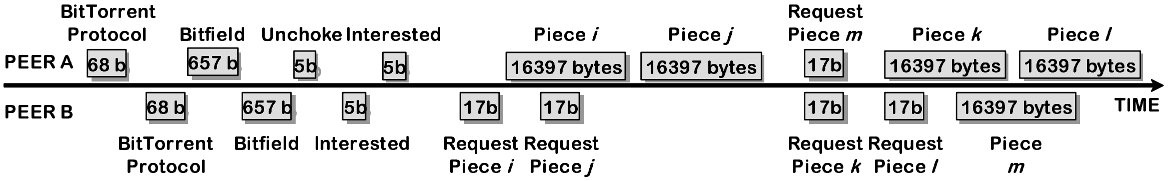


Figure 1.3: A diagram illustrating the interaction between two BitTorrent peers.

common that the sequential one, but it is supported in important protocols like HTTP/1.1 (pipelining), NNTP (streaming mode) and BitTorrent. Our analysis shows that while the fraction of connections with concurrent data exchanges is usually small, (17.4%), such concurrent connections often carry a significant fraction (15%-35%) of the total bytes seen in a trace, and hence modeling these connections is critical if one wants to generate realistic traffic mixes.

To represent concurrent ADU exchanges, the actions of each endpoint are considered to occur independently of each other. Thus each endpoint is a separate source generating ADUs that appear as a sequence of epochs following a unidirectional flow pattern. Formally, this means that we represent each connection as a pair (α, β) of connection vectors of the form

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

and

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

where a_i and b_i are sizes of ADUs sent from the initiator and from the acceptor of the TCP connection respectively, and ta_i and tb_i are quiet times between the ADUs. We call this representation of source-level behavior a *concurrent connection vector*. Unlike the sequential version of the a-b-t model, this representation does not capture any causality between the two directions of a TCP connection. As a consequence, traffic generated according to this version of the model usually exhibits a substantial number of concurrent data exchanges.

The a-b-t model provides a simple yet expressive way of describing source-level behavior in a generic manner that is not tied to the details of any application. In addition, this non-parametric model was designed to incorporate quantities (ADU sizes, ADU directionality, and inter-ADU quiet time duration) that can be extracted from packet header traces in a efficient, accurate manner. We can easily imagine more complex and expressive models of TCP connections for which no efficient data acquisition algorithm

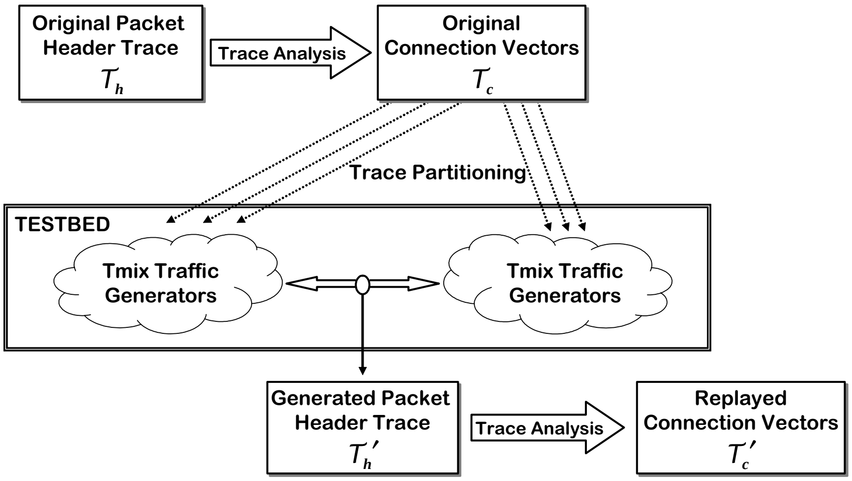


Figure 1.4: Overview of Source-level Trace Replay.

exists, or models that deal with characteristics of source-level behavior that cannot be extracted purely from packet headers. In the case of the a-b-t model, we have developed a data acquisition algorithm that relies on TCP sequence numbers for measuring ADU sizes, and on the packet arrival timestamps obtained during trace collection to determine inter-ADU quite times. Our algorithm constructs a data structure in which TCP segments are ordered according to their *logical data order*, *i.e.*, the order in which data must be delivered to the application layer of the receiving endpoint. In reconstructing this logical order for each connection, we have developed methods for dealing with network pathologies such as arbitrary segment reordering, duplication and retransmission. Furthermore, when the data segments in a TCP connection cannot be ordered according to the logical data order, we can classify the connection as concurrent with certainty. Our data structure supports both sequential (*i.e.*, bidirectional) and concurrent (*i.e.*, unidirectional) ordering, making it possible to extract ADU sizes and quiet times with a single pass over the segments of a TCP connection found in a trace. The analysis can be performed in $O(sW)$ time, where s is the number of data segments in the connection and W is the maximum size of the TCP window (which bounds the maximum amount of reordering).

1.2 Source-Level Trace Replay

Our abstract source-level modeling of TCP connection provides a solid foundation for generation traffic mixes in simulators and network testbeds. We propose to generate traffic using *source-level trace replay*, as illustrated in Figure 1.4. Given a packet header trace $\mathcal{T}_h$ collected from some Internet link, we first use our data acquisition algorithm to analyze the trace and describe its content as a collection of connection vectors $\mathcal{T}_c = \{(T_i, C_i)\}$, where T_i is the relative start time of the i-th TCP connection, and C_i is the sequential or concurrent connection vector corresponding to this connection. The basic approach for generating traffic according to $\mathcal{T}_c$ is to replay every connection vector C_i. Each connection vector C_i is replayed by starting a TCP connection precisely at C_i's relative start time T_i, and transmitting the measured sequence of ADUs (a_j and b_j) separated in time by the inter-ADU measured quiet times (ta_i and tb_i). In this dissertation, we evaluate a specific implementation of this approach for FreeBSD network testbeds, where traffic is generated using a tool we developed called *tmix* .

The goal of the direct source-level trace replay of $\mathcal{T}_c$ is to reproduce the source-level characteristics of the traffic in the original link, generating the traffic in a closed-loop fashion. Closed-loop traffic generation implies the need to simulate the behavior of applications, using regular network stacks to actually translate source-level behavior into network traffic. In particular, our experiments use an implementation which relies on the standard socket interface to reproduce the data exchanges in each connection vector. Generating traffic in this manner is closed-loop in the sense that it preserves the feedback mechanism in TCP, which adapts its behavior to changes in network conditions, such as loss and receiver saturation. In contrast, packet-level trace replay, the direct reproduction of $\mathcal{T}_h$, is an open-loop traffic generation method in the sense that TCP control algorithms are not used during the generation, and hence the traffic does not adapt to network conditions.

The evaluation of our methodology consists of comparing the original trace $\mathcal{T}_h$ and the synthetic trace $\mathcal{T}_h'$ obtained from the source-level trace replay. Validating our traffic generation method consists of transforming $\mathcal{T}_h'$ into a set of connection vectors $\mathcal{T}_c'$, using the same method used to transform $\mathcal{T}_h$ into $\mathcal{T}_c$. We then compare the resulting set of connection vectors $\mathcal{T}_c'$ with the original $\mathcal{T}_c$. In principle, they should be identical, since $\mathcal{T}_c$ represents the invariant source-level characteristics of $\mathcal{T}_h$. There are however some differences that are explained by the nature of the model and our measurement methods.

The direct comparison of $\mathcal{T}_h$ and $\mathcal{T}_h'$ also provides a way to study the accuracy of our approach in terms of how well traffic is described by the a-b-t model. This is however a subtle exercise. The actual replay of $\mathcal{T}_c$, which creates $\mathcal{T}_h'$, necessarily requires the selection of a a set of network-level parameters, such as round-trip times and TCP receiver window sizes, for each TCP connection in the source-level trace replay. The exact set of generated TCP segments and their arrival times is a direct function of these parameters. As a consequence, if we conduct a source-level trace replay using arbitrary network-level parameters, we obtain a $\mathcal{T}_h'$ with little resemblance to the original $\mathcal{T}_h$. The replayed a-b-t connection vectors may be a perfect description of the source behavior driving the original connections, but the generated packet-level trace $\mathcal{T}_h'$ would *still* be very different from the original $\mathcal{T}_h$. To address this difficulty, our replay incorporates network-level parameters individually derived from each connection in $\mathcal{T}_h$. We have also incorporated methods for measuring three important network-level parameters (round-trip time, TCP receiver window size and loss rate) into our analysis and generation procedure. While this set of parameters is by no means complete, it does include the main parameters that affect the average throughput of a TCP connection found in a trace. This enables us to generate traffic in a closed-loop manner that approximates measured traces very closely.

Incorporating network-level properties is important, but it is critical to understand the main shortcoming of this approach. The goal of our work is not to make the generated traffic $\mathcal{T}_h'$ identical to the original traffic $\mathcal{T}_h$, which could be accomplished with a simple packet-level replay. As mentioned before, packet-level replays generate traffic that does not adapt to changes in network conditions, resulting in open-loop traffic. Our goal is to develop a closed-loop traffic generation method based on a detailed characterization of source behavior. Traffic generated in a closed-loop manner can adapt to different network conditions, which are intrinsic when evaluating different network mechanisms. Our comparison of $\mathcal{T}_h$ and $\mathcal{T}_h'$ is only a means to understand the quality of traffic generation method, where quality is considered to be higher as the original trace is more closely approximated. If enough parameters of the original traffic are accurately measured and incorporated into the traffic generation experiment, we expect to observe a great similarity between $\mathcal{T}_h$ and $\mathcal{T}_h'$. On the contrary, if we are missing some important parameters, we expect to observe substantial differences between traces.

By construction, traffic generated using source-level trace replay can *never* be identical to the original traffic. The statistical properties of original packet header traces are the result of multiplexing a large number of connections onto a single link, and these connections traverse a large number of different paths with a variety of network conditions. It is simply not possible to fully characterize this environment and reproduce it in a laboratory testbed or in a simulation. This is both because of the limitations of passive inference from packet headers, and because of the stochastic nature of network traffic. Source-level trace replay can never incorporate every factor that shaped $\mathcal{T}_h$, and therefore differences between $\mathcal{T}_h$ and $\mathcal{T}_h'$ are unavoidable. Still, finding a *close* match between an original trace and its replay, even if they are not identical, constitutes strong evidence of the accuracy of the a-b-t model and the data acquisition and generation methods we have developed. It also demonstrates the feasibility of generating realistic network traffic in a closed-loop manner that resembles a rich traffic mix.

1.3 Trace Resampling and Load Scaling

As long as the network setup of a simulation or testbed experiment remains unchanged, the source-level trace replay of a connection vector trace $\mathcal{T}_c = \{(T_i, C_i)\}$ always results in traffic that is similar to the original trace. Every replay contains the same number of TCP connections behaving according to the same connection vector specification and starting at the same times. Only tiny variations are introduced on the end-systems by changes in clock synchronization, operating system scheduling and interrupt handling, and at switches and routers by the stochastic nature of packet multiplexing. Source-level trace replay has therefore two desirable properties:

- The quality of the synthetic traffic can be evaluated by *directly comparing* synthetic and original traffic. This makes it possible to study the accuracy of the analysis methods and the generation system with complete freedom, using any metric that can be derived from real traffic. In contrast, more abstract methods based on parametric models of traffic are inherently stochastic and therefore more difficult to evaluate. For such methods, it is less obvious whether the observed difference between the traffic generated using the parametric model and the original traffic from which the model derives should be admitted.
- The generation of the synthetic traffic is fully *reproducible*. A researcher can expose a collection of network protocols and mechanisms to exactly the same closed-loop traffic, which provides the right foundation for fair comparative studies. In contrast, stochastic variation in the traffic generated using parametric models is often difficult to control. For example, experiments with models that rely on heavy-tailed distributions converge very slowly to comparable conditions, as discussed by Crovella and Lipsky [CL97].

While these properties are important, the practice of experimental networking often requires to introduce controlled variability in the generated traffic for exploring a wider range of scenarios. This motivates the development of methods that manipulate $\mathcal{T}_c$ in order to generate different traffic that still *resembles* the original one. Furthermore, developing a statistically sound way of manipulating $\mathcal{T}_c$ is essential for generating traffic with different levels of offered load. This manipulation to match a target offered load is a very common need in experimental networking research. This is because the performance of a network mechanism or protocol is often affected by the amount of traffic to which it is exposed. Therefore, rigorous experimental studies frequently require to generate a complete range of target loads.

In this dissertation, we propose two flexible methods for introducing variability in traffic generation experiments. In both cases, the set of connection vectors in $\mathcal{T}_c$ is randomly resampled, resulting in a new set $\mathcal{T}_c'$ that preserves the aggregate source-level characteristics of the original traffic. In our first method, *Poisson Resampling*, we construct a new connection vector trace $\mathcal{T}_c'$ by randomly resampling connections from $\mathcal{T}_c$, and assigning them exponentially distributed inter-arrival times. As a result, connections in $\mathcal{T}_c'$ arrive according to a Poisson process. In the second method, *Block Resampling*, we resample blocks (groups) of connections rather than individual connections. This method results in a more realistic connection arrival process, which matches the substantial burstiness observed in real traces. In more technical terms, Block Resampling preserves the moderate long-range dependence found in real connection arrival processes, while Poisson Resampling results in a short-range dependent connection arrivals process. This difference is demonstrated in our experimental evaluation of the two methods. In addition, the evaluation shows that the duration of the resampling block creates a trade-off between shorter blocks (which increase the number of distinct resamplings) and long-range dependence (which disappears for short blocks). Our analysis demonstrates that block durations between 1 and 5 minutes offer the best compromise.

Researchers often need to conduct a set of experiments with a range of different traffic loads. When using a traditional source-level model, *e.g.*, a model of web traffic, researchers have to first conduct a preliminary experimental study to determine how the parameters of the model, *e.g.*, the number of user equivalents, affect the generated load [CJOS00, LAJS03, KcLH$^+$02]. This is usually known as the *calibration* of traffic generator. Our resampling methods eliminate this common need for calibrating traffic generators, since the resampling process can be controlled to match a specific target load (*i.e.*, generated load is known a priori). In the case of Poisson Resampling, this is accomplished by changing the mean arrival rate of connections. In the case of Block Resampling, offered load is manipulated using block thinning (*i.e.*, subsampling) and block thickening (*i.e.*, combining blocks). Our work reveals that load scaling cannot be based simply on controlling the number of connections. Such an approach frequently results in offered loads that are far from the target, because the number of connections in a resample is not strongly correlated with the offered load represented by these connections. We address this difficulty by developing *byte-driven* versions of Poisson Resampling and Block Resampling, which scale load using a running count of the total data in the resampled trace $\mathcal{T}_c'$. Unlike the number of connections, the total amount of data in $\mathcal{T}_c'$ is strongly correlated to traffic load offered by $\mathcal{T}_c'$. Our experiments confirm that byte-driven resampling is highly accurate, eliminating the common need for calibrating traffic generators.

1.4 Thesis Statement

This dissertation considers the following thesis:

1. An abstract source-level model can describe in detail the entire set of TCP application behaviors observed in real networks.

2. Descriptions of abstract source-level behavior can be empirically derived from packet header traces in an efficient, accurate manner.

3. Traffic generation based on this abstract source-level modeling results in synthetic traffic that is realistic and suitable for experimental networking research.

4. The abstract source-level model of a trace can be manipulated to introduce statistically valid variability in the generated traffic and also to accurately match a target offered load while preserving application characteristics.

1.5 Contributions

We highlight the following contributions from this dissertation:

- We develop the concept of abstract source-level modeling and the a-b-t notation for describing the source-level behavior of entire traffic mixes. We identify a fundamental dichotomy in source-level behavior between connections that exchange data sequentially and connections that exchange data concurrently. Our a-b-t notation includes a sequential version and a concurrent version that makes it possible to appropriately describe these two types of behaviors.

- We formulate a formal test of concurrency that can be applied to the packet headers of any TCP connection, and that does not suffer from false positives. This enables us to accurately classify connections as sequential or concurrent. We show that only a small fraction of TCP connections (less than 4% in our traces) exchange data concurrently, but that these TCP connections account for a substantial fraction (up to 32%) of the total traffic.

- We present an efficient algorithm for transforming a packet header trace into a collection of sequential and concurrent a-b-t connection vectors. Given a TCP connection for which we observe s segments and that has a maximum receiver window size of W, the asymptotic cost of our algorithm is $O(sW)$. We demonstrate that this algorithm is accurate using traffic generated from synthetic applications (*i.e.*, with known characteristics).

- We develop source-level trace replay, a closed-loop traffic generation method that uses a-b-t connection vectors as a non-parametric model of network traffic. One key benefit of this approach is the possibility of directly comparing original and generated traffic, which we use to evaluate the "realism" of our traffic generation approach. This comparison requires us to incorporate some network-level parameters (round-trip times, maximum receiver window sizes, and possibly loss rates) into the traffic generation. These parameters can be measured from packet header traces. We pay special attention to passive round-trip time estimation in our data acquisition, developing the concept of One-Side Transit Time and studying the impact of delayed acknowledgments on passive round-trip time estimation.

- We implement our traffic generation method in a network testbed, developing a new distributed traffic generation tool, *tmix* . We use this implementation to study the results of a large collection of trace replay experiments, evaluating the need for detailed source-level modeling and the impact of losses on measured network traffic. Our results demonstrate that detailed source-level modeling is often required for accurately approximating real traffic, which demonstrates that source-level behavior is a major factor shaping Internet traffic. The most substantial differences are observed for the number of active connections and the number of packet arrivals per unit of time. Byte arrivals per unit of time and long-range dependence do not improve so consistently with the use of detailed source-level modeling. We also show that losses had only a secondary effect in our traces, but they are not negligible when comparing original and generated traffic.

- We present two trace resampling algorithms which can be used to derive new traces from an existing one, preserving its statistical characteristics at the source-level. Our comparison of the two methods reveals that the observed long-range dependence in connection arrivals has no apparent impact on the long-range dependence of packet and byte arrivals.

- We demonstrate the need for byte-driven rather than connection-driven resampling in order to accurately scale offered loads, and develop byte-driven versions of our two resampling methods. This approach eliminates the need for the experimental calibration of traffic generators (which study the relationship between the parameters of the generator and the offered traffic load).

- Our entire methodology makes it possible to conduct networking experiments with closed-loop synthetic traffic derived from real traces in an automated manner. This eliminates the need for painstaking parametric modeling.

1.6 Overview

Chapter 2 presents a review of the state-of-the-art in synthetic traffic generation. We first expand our discussion of packet-level traffic generation and data acquisition, and then examine source-level traffic generation more in depth. We review the literature on application-specific modeling, discussing models of web traffic and other applications, and also consider several approaches for generating traffic driven by more than one application. We also discuss existing methods for controlling the traffic load created in networking experiments. The chapter finally considers some research efforts addressing implementation issues.

Chapter 3 discusses abstract source-level modeling, presenting several examples of real applications and how their behavior can be described using our a-b-t notation. We also present our measurement algorithm for transforming a packet header trace into a collection of sequential and concurrent a-b-t connection vectors. The chapter also includes a validation of the measurement method using synthetic applications, and a measurement study that examines the statistical properties of the a-b-t connection vectors extracted from five real traces.

Chapter 4 focuses on network-level measurement. We first describe our methods for measuring round-trip times, window sizes and loss rates, and an evaluation of their accuracy. While this set of parameters is by no means complete, it does include the main parameters that affect the average throughput of a TCP connection found in a trace. The second part of Chapter 4 describes the network-level metrics that we consider in the evaluation of our traffic generation method: packet and byte throughput time series, their marginal distributions, wavelet spectra, Hurst parameter estimates and time series of active connections.

Chapter 5 describes source-level trace replay and our implementation in a network testbed. We present a validation of this implementation using the source-level trace replays of five traces. For each trace, we study the a-b-t connection vectors extracted from the original traces and those found in replays with and without packet losses at the network links. The results demonstrate the accuracy of our approach, and also uncover some difficulties, which are in some cases inherent to the a-b-t model and its passive method of data acquisition.

Chapter 6 examines the results of several source-level trace replay experiments. Our analysis compares original traces and their source-level trace replays using the rich set of metrics introduced in Chapter 4, revealing a remarkably close approximation. This study also includes a comparison of traffic generated with the a-b-t model and with a simplified version that “disables” source-level modeling, which is shown to perform well for some metrics and poorly for others. As in the previous chapter, we also consider experiments with and without artificial losses, showing that loss did not have a dominant impact on the characteristics of the original traffic. In general, our results provide a strong justification of our source-level modeling approach, demonstrating that the closed-loop replay of a-b-t connection vectors closely resembles real traffic.

Chapter 7 presents our two resampling methods, Poisson Resampling and Block Resampling. These methods enable the researcher to introduce controlled variability in source-level trace replay experiments, without sacrificing reproducibility. In addition, we consider the problem of load scaling, *i.e.*, how to control the resampling process to obtain a new trace with a target offered load. Our work demonstrates that this task can be accomplished by keeping track of the total number of data bytes in the resampled trace, but not by keeping track of the number of connections. Our scaling methods eliminate the common need for running a preliminary study to calibrate the traffic generator.

Chapter 8 presents our conclusions and discusses future work.

CHAPTER 2

Related Work

A scientific theory should be as simple as possible, but no simpler.

— Albert Einstein (1879–1955)

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— Bertrand Russell (1872–1970)

This chapter presents an overview of the research literature relevant for realistic traffic generation. We consider two types of works. First, we discuss the body of literature that developed the concepts and techniques currently in use for generating synthetic traffic in simulations and testbed experiments. Second, we examine the Internet measurement literature that informs the discussion of what is meant by "realistic" traffic generation. Intuitively, synthetic traffic resembling Internet traffic can only be realistic if derived from measurements conducted from real network links. We could argue that any Internet measurement paper helps to gain a better understanding of the nature of the Internet and its traffic, being therefore relevant for realistic traffic generation. However, the sheer size of the Internet measurement literature makes a complete overview impractical, so we will restrict ourselves to the main works that had a direct impact on Internet traffic generation. It is also interesting to note that the most recent trend in the field of traffic generation is precisely to combine traffic measurement and generation into a single, coherent approach [HCJS$^+$01, LH02, SB04, HCSJ04].

Traffic generation for experimental networking research was identified as one of the key challenges in Internet modeling and simulation by Paxson and Floyd [PF95] in 1995. Interestingly, Floyd and Kohler [FK03] made a similar point in 2003, and argued that it was still difficult to conduct experiments with representative, validated synthetic traffic. While traffic measurement and Internet measurement in general have become increasingly popular in recent years, most studies are exploratory and provide little foundation to build traffic generators. This chapter provides an overview of the major works in the field of Internet traffic generation, considering first packet-level traffic generation and then source-level traffic generation. Other aspects of traffic generation, such as load scaling, incorporating network-dependencies and implementation issues are discussed at the end of the chapter.

2.1 Packet-Level Traffic Generation

In this dissertation we restrict the question of generating realistic traffic to a single link. This is the most essential form of the traffic generation problem. It does not seem possible to tackle the problem of generating traffic for multiple links, say the backbone of an ISP, if single-link traffic generation is not fully understood.

The simplest way of generating realistic traffic on a single link is to inject packets into the network according to the characteristics of the packets observed traversing a real link. We will use the term *packet-level traffic generation* to refer to this approach. Packet-level traffic generation can mean either performing a *packet-level replay*, *i.e.*, reproducing the exact arrivals and sizes of every observed packet, or injecting packets in such a manner as to preserve some set of statistical properties considered fundamental, or relevant for a specific experiment. Packet-level replay, which has been implemented in tools like *tcpreplay* [tcpb], is a straightforward technique that is useful for certain types of experiments where configuration of the network is not expected to affect the generated traffic. In other words, whenever it is reasonable to generate traffic that is invariant of (*i.e.*, unresponsive to) the experimental conditions, then packet-level replay is an effective means for generating synthetic traffic. For example, packet-level replays of traces collected from the Internet have been used to evaluate cache replacement policies in routing tables [Jai90, Fel88, GcC02]. In this type of experiments, different cache replacement policies are compared by feeding the lookup cache of a routing engine with a packet trace and computing the achieved hit ratio. Also, studies that require malicious traffic generation can often make use of packet-level replay [SYB04, RDFS04]. Malicious traffic (*e.g.*, a SYN flood) is frequently not responsive to network conditions (and their degradation).

Before conducting an experiment in which traffic is generated using packet-level replay, researchers must obtain one or more traces of the arrivals of packets to a network link. These traces are collected using a packet “sniffer” to monitor the traffic traversing some given link. This packet capturing can be performed with and without hardware support. The most prominent example of software-only capture is the Berkeley Packet Filter (BPF) system [MJ93, tcpa]. BPF includes a packet capturing library, *libpcap*, and a command-line interface and trace analysis tool, *tcpdump*. BPF relies on the promiscuous mode of network interfaces to observe packets traversing a network link and to create a trace of them in the "pcap" format. Due to privacy and size considerations, most traces only include the protocol headers (IP and TCP/UDP) of each packet and a timestamp of the packet's arrival. Monitoring high-speed links with a software-only system is problematic, given that traffic has to be forwarded from the network interface to the monitoring software using the system bus. The system bus may not be fast enough for this task depending on the load on the monitored link. High loads can result in "dropped" packets that are absent from the collected trace. Furthermore, the extra forwarding from the wire to the monitoring program, which usually involves buffering in the network interface and in operating system layers, makes timestamps rather inaccurate. In the case of BPF, timestamping inaccuracies of a few hundreds of microseconds are quite common. In order to overcome these difficulties, researchers often make use of specialized hardware that can extract headers and provide timestamps without the intervention of the operating system. This is of course far more expensive, but it dramatically improves timestamp accuracy and increases the volume of traffic that can be collected without drops. The DAG platform [Pro, GMP97, MDG01] is a good example of this approach, and it is widely used in network measurement projects. The timestamping accuracy of DAG traces is on the order of nanoseconds. Multiple DAG cards, possibly at different locations, can also be synchronized using an external clock signal, such as the one from the Global Positioning System (GPS). Besides collecting their own traces, researchers can also make use of public repositories of pcap and DAG traces, such as the Internet Traffic Archive [Int] and the PMA project at NLANR [nlab].

While packet-level replay is conceptually simple, it involves a number of engineering challenges. First, traffic generators usually rely on operating systems layers and abstractions, such as raw sockets, to perform the packet-level replay. Most operating systems provide no guarantee on the exact delay between the time of packet injection by the traffic generator and the time at which the packet leaves the network interface. Servicing interrupts, scheduling processes, *etc.*, can introduce arbitrary delays, which make the arrival process of the packet replay differ from the original and intended arrival process. This inaccuracy may or may not be significant for a given experiment. Another challenge is the replay of traces collected in high-speed links. The rate of packet arrivals in a trace can be far higher than the rate at which a single host can generate packets. For example, the speed at which a commodity PC can inject packets into the network is primarily limited by the speed of its bus and the bandwidth of its network interface. As a consequence, replying a high rate trace often requires an experimenter to partition the trace into subtraces that have to be replayed using a collection of hosts. In this case, it is important to carefully synchronize the replay of these hosts. This is generally a difficult task, since the synchronization has to be done using the network itself, which introduces variable I/O delays. Clock drift is also a concern with common PC clocks.

Ye *et al.* [YVIB05] discussed packet-level replay of high rate traces, focusing on OC-48, and how to evaluate the accuracy of the replay. They proposed *flow-based splitting* to construct a partition of the original trace that can be accurately replayed by an ensemble of traffic generators. This addresses the challenge of replaying a trace using multiple traffic generators without reordering the packets within a flow. In contrast, round-robin assignment of packets to traffic generators, called *choice of N* in this work, results in packets belonging to the same flow generated by different traffic generators. As a consequence, the generated traffic exhibits substantial packet reordering. This reordering is due to the difficulty of maintaining the generators perfectly synchronized with commodity hardware, so one generator can easily get ahead of another and modify the order of packets within a flow. Ye *et al.* also discussed the difficulties created by buffering on the network cards, which modifies the properties of the packet arrival process at fine scales. An alternative to the approach in Ye *et al.* is to rely on specialized hardware. Most DAG cards support packet-level replay, bypassing the network stack. However, no information is available on how accurately the generated traffic preserves the properties of original packet arrival process.

Packet-level replay has two important shortcomings: it is inflexible and it is open-loop. Given that a packet-level replay is the exact reproduction of a collected trace, both in terms of packet arrival times and packet content, there is no way to introduce variability in the experiments other than acquiring a collection of traces and using a different trace in different runs of the experiments. This makes packet replay inflexible, since the researcher has to limit his experiments to the available traces and their characteristics. The "right" traces may not be available or may be difficult to collect. Even conducting experiments that study simple questions can be cumbersome. For example, a researcher that intends to test a cache replacement policy under heavy loads must find traces with high packet arrival rates, which may or may not be available. Similarly, evaluating a queuing mechanism under a range of (open-loop) loads requires one to find traces covering this range of loads, and may involve mixing traces from different locations, which could cast doubt on the realism of the resulting traffic and thus on the conclusions of the evaluation.

More flexible traffic generation can be achieved by generating packets according to a set of statistical properties derived from real measurements. The challenge then is to determine which properties of traffic are most important to reproduce so that the synthetic generated traffic makes the experiments "realistic enough." For example, Internet traffic has been found to be very bursty, showing very frequent changes in throughput (both for packets and bytes per unit of time). Therefore, most experiments should make use of synthetic traffic that preserves this observed burstiness. Leland *et al.* [LTWW93] observed that this burstiness can be studied using the framework provided by *statistical self-similarity*. At a high-level, self-similarity means that traffic is equally bursty, *i.e.*, equal variance in arrival times, across a wide range of time scales. This is similar to the geometric self-similarity that fractals exhibit. Mathematically, statistical self-similarity manifests itself as *long-range dependence*, a sub-exponential decay of the autocorrelation of a time-series with scale. This is in sharp contrast to Poisson modeling and its short-range dependence, which implies an exponential decay of the autocorrelation with scale. Therefore, it is generally difficult to accept experimental results where synthetic traffic does not exhibit some degree of self-similarity. Accordingly, some experiments may simply rely on some method for generating a self-similar process [Pax97] and inject packets into the experiments according to this process. Studies on queuing dynamics, *e.g.*, [ENW96], made use of this traffic generation approach.

Other experiments with a more stringent need for realism may also attempt to reproduce other known properties of traffic. For example, a realistic distribution of IP addresses is essential for experiments in which route caching performance is evaluated. To accomplish this, packet-level traffic generation can be combined with a statistical model of packet arrival and a model of address structure. As one example, Aida and Abe [AA01] proposed a generative model based on the finding that the popularity of addresses follows a *powerlaw* (a heavy-tailed distribution with a hyperbolic shape). In contrast, Kohler *et al.* [KLPS02] focused on the hierarchical structure of addresses and prefixes, which is shown to be well-described by a multi-fractal model. Both studies could be used to enrich packet-level traffic generation.

2.2 Source-Level Traffic Generation

While packet-level traffic generation based on a set of statistical properties is convenient for the experimenter, and attractive from a mathematical point of view, it fails to preserve an essential property of Internet traffic. As Floyd and Paxson [PF95] point out, packet-level traffic generation is *open-loop*, in the sense that it does not preserve the feedback loop that exists between the sources of the traffic (the endpoints) and the network. This feedback loop comes from the fact that endpoints react to network conditions, and this reaction itself can change these conditions, and therefore trigger further changes in the behavior of the endpoints. For example, TCP traffic reacts to congestion by lowering its sending rate, which in turn decreases congestion. A trace of packet arrivals collected at some given link is therefore specific to the characteristics of this link, the time of the tracing paths of the connections that traversed it, *etc.* Therefore, any changes that the experimenter makes to the experimental conditions make the packet-level traffic invalid since the traffic generation process is insensitive to these changes (unlike real Internet traffic). For example, packet-level replay of TCP traffic does not react to congestion in any manner.

The solution is to *model the sources* of traffic, *i.e.*, to model the network behavior of the applications running on the endpoints that communicate using network flows. Source-level models are then used to drive network stacks which do implement flow and congestion control mechanisms, and therefore react to changes in network conditions as real Internet endpoints do. As a result, the generated traffic is *closed-loop*, which is far more realistic for a wide range of experiments.

The simplest source-level model is the *infinite source model*. The starting point of the infinite source model is the availability of an infinite amount of data to be communicated from one endpoint to another. Generating traffic according to this model means that a traffic generator opens one or more transport connections, and constantly provides them with data to be transferred. This means that, for each connection, one of the endpoints is constantly writing (sending data packets) while the other endpoint is constantly reading (receiving data packets). The sources are never the bottleneck in this model. The only process that limits the rate at which the endpoints transmit data is the network, broadly defined to include any mechanism below the sources, such as TCP's maximum receiver window.

The infinite source model is very attractive for several reasons, which make it rather popular in both theoretical and experimental studies [FJ93b, KHR02, AKM04, SBDR05]. First, the infinite source model has no parameters and hence it is easy to understand and amenable to formal analysis. It was, for example, the foundation for the work on the mathematical analysis of steady-state TCP throughput [PFTK98, BHCKS04]. Second, its underlying assumption is that the largest flows on the network, which account for the majority of the packets and the bytes, "look like" infinite sources. For example, an infinite source provides a convenient approximation to a multi-gigabyte file download using FTP. Third, infinite sources are well-behaved, in the sense that, if driving TCP connections, they try to consume as much bandwidth as possible. They also result in the ideal case for bandwidth sharing. This makes them useful for experiments in the area of congestion control, since infinite sources can easily congest network links.

Despite their convenience, infinite sources are unrealistic and do not provide a solid foundation for networking experiments, or even for understanding the behavior and performance of the Internet. The pioneering work by Cáceres *et al.* [CDJM91], published as early as 1991, provided a first insight into the substantial difference between infinite sources and real application traffic. These authors examined packet header traces from three sites (the University of California at Berkeley, the University of Southern California, and Bellcore in New Jersey) using the concept of application-level *conversations*. An application-level conversation was defined as the set of packets exchanged between two network endpoints. These conversations could include one or more "associations" (TCP connections and UDP streams). A general problem when studying traffic for extended periods is the need to separate traffic into independent units of activity, which in this case correspond to conversations. Endpoints may exchange traffic regularly, say every day, but that does not mean that they are engaged in the same conversation for days. Danzig *et al.* separated conversations between the same endpoints by identifying long periods without any traffic exchange, which are generally referred to as *idle times* or *quiet times* in the literature. In their study, they used a threshold of 20 minutes to differentiate between two conversations. The authors examined conversations from 13 different applications, characterizing them with the help of empirical cumulative distribution functions (empirical CDFs). The results include empirical CDFs for the number of bytes in each conversation, the directionality of the flow of data (*i.e.*, whether the two endpoints sent a similar amount of data), the distribution of packet sizes, the popularity of different networks, *etc.* Danzig and Jamin [DJ91] used these distributions in their traffic generation tool, *tcplib*. The results from this work are further discussed in Section 2.2.2.

Cáceres *et al.* pointed out a number of substantial differences between their results and the assumptions of earlier works. First, the majority of connections carried very small amounts of data, less than 10 KB in 75-90% of the cases. This is true for both interactive applications (*e.g.*, telnet and rlogin) and bulk transfer applications (*e.g.*, FTP, SMTP). This is in sharp contrast to the infinite availability of data to be transferred assumed in the infinite source model. The dynamics of such short data transfers are completely different from those of infinite sources, which for example have time to fully employ congestion control mechanisms. The second difference was that traffic from most applications was shown to be strongly bidirectional, and it included at least one request/response phase, *i.e.*, an alteration in the role of the endpoints as senders of data. The infinite source model is inherently unidirectional, with one of the endpoints always acting as the sender, and the other endpoint always acting as the receiver. Third, the authors observed a wide range of packet sizes, and a large fraction of the data packets were small, even for bulk applications. Data packets from an infinite source are necessarily full size, since there is by definition enough data to completely fill new packets.

These measurement results highlighted a substantial difference between infinite sources and real traffic, and later experimental studies demonstrated the perils of using traffic from infinite sources in the evaluating of network mechanisms. Joo *et al.* [JRF$^+$99, JRF$^+$01] demonstrated that infinite TCP sources tend to become synchronized, so they increase or decrease their transmission rate at the same time. This pattern is completely absent from more realistic experiments in which the majority of the sources have small and diverse amounts of data to send. As a result, loss patterns, queue lengths and other characteristics are strikingly different when more realistic synthetic traffic is used. Joo *et al.* also studied the difference between open-loop and closed-loop traffic generation.

The area of active queue management has provided several illustrations of the misleading results obtained with the unrealistic infinite sources. The first AQM scheme, RED, was presented by Floyd and Jacobson in [FJ93b], and evaluated using infinite sources. Their results showed that RED significantly outperformed FIFO, the usual router queuing mechanism. Later work by Christiansen *et al.* [CJOS00] demonstrated that RED offers very little benefit, if any, when exposed to more realistic traffic where sources are not infinite. In particular, they used a model of web-like traffic, which is discussed later in this chapter.

Paxson's analysis [Pax94] of packet header traces from seven different network links provided further support for the conclusions of Cáceres *et al.* In addition, Paxson considered the parsimonious modeling of traffic from different applications. He characterized four prominent applications, telnet, NNTP, SMTP and FTP, using analytic models to fit the empirical distributions. Analytic models are more commonly known as parametric models in the statistical literature, and correspond to classical distributions, such as the Pareto distribution, that can be fully characterized with a mathematical expression and only one or a few parameters. As Paxson pointed out, the use of analytic models results in a concise description of network applications that can be easily communicated and compared, and are often mathematically tractable. His methodology has had a lasting influence in application-level modeling. He clearly demonstrated that analytic fits (*i.e.*, parametric models) of the observed distributions can closely approximate the characteristics of real applications. However, it is important to remember that traffic is not necessarily more realistic when generated by analytic models as opposed to empirical models. Empirical CDFs, derived from network measurement of sufficient size, provide a perfectly valid foundation for traffic generators. Furthermore, finding analytic fits of complex random variables that do not match well-known statistical distributions is a daunting task.

2.2.1 Web Traffic Modeling

Modeling web traffic has received substantial attention since the sudden emergence of the World Wide Web in the mid-nineties. Arlitt and Williamson [AW95] proposed an early model for generating web traffic[1], based on packet header traces collected at the University of Saskatchewan. The model was centered around the concept of a conversation, as proposed by Cáceres *et al.* [CDJM91]. In this case, a conversation was the set of connections observed between a web browser and a web server. These authors were the first to consider questions such as the distribution of the number of bytes in requests and responses, the arrival rates of connections, *etc.* In general, the proposed model has parameters that are quite different from those of later works. For example, an Erlang model of response sizes was used, which is in sharp contrast to the heavy-tailness observed by other authors. While Arlitt and Williamson did not provide any details on the statistical methods they employed, it is likely that the small sample size (less than 10,000 TCP connections) made it difficult to develop a more statistically representative model.

One of the major efforts in the area of web traffic modeling oriented toward traffic generation took place at Boston University. Cunha *et al.* [CBC95] examined client traces collected by instrumenting browsers at the Department of Computer Science. Unlike the packet header traces used in Arlitt and Williamson, client traces include application information such as the exact URL of each web object requested and downloaded in each TCP connection. The authors made use of this information to study page and server popularity, which are relevant for web caching studies. In addition, the authors proposed the use of powerlaws for constructing a parametric model of web traffic. They relied on the Pareto distribution for modeling the sizes of downloaded objects, and the parameterless Zipf's law for modeling the popularity of specific pages. Crovella and Bestavros [CB96] used these findings to explain the long-range dependence observed in the packet arrivals of web traffic. Their explanation was derived from earlier work by Willinger *et al.* [WTSW97], which showed that the multiplexing of heavy-tailed ON/OFF sources results in long-range dependent traffic. Crovella and Bestavros demonstrated that the underlying distributions of web object sizes, the effects of caching and user preference in file transferring, the effect of user "think time", and the superimposition of many web transfers precisely creates the multiplexing process hypothesized by Willinger *et al.*

Crovella and Bestavros also showed that the explanation behind the suitability of powerlaws for describing the sizes of web objects is that the sizes of files are well described by powerlaws. This refined

[1] To be more specific, Arlitt and Williamson proposed a model of "Mosaic" traffic. Mosaic was the first web browser.

previous studies of file-system characteristics (*e.g.*, [BHK$^+$91]), which observed long-tailed distributions of file sizes (but did not propose powerlaw models).

Powerlaw modeling has had a lasting impact on traffic modeling, which is natural given that the transfer of files is one of the most common uses of many application protocols. Countless studies have confirmed the usefulness of powerlaws for modeling application traffic. The eloquent term "mice and elephants" [GM01, MHCS02, EV03], often applied to Internet traffic, precisely refers to the basic characteristic of powerlaws: a majority of values are small (mice) but the uncommon large values (elephants) are so large that they account for a large fraction of the total value. For example, web traffic usually shows around 90% of web objects below 10 KB, but larger objects often account for 90% of the total bytes. Researchers have used this general finding of powerlaw sizes to develop a generic, and mostly ad hoc, source-level model. Traffic generated according to this model consists of a collection of TCP connections that transfer a single file, such that the distribution of file sizes follows a powerlaw. Researchers often refer to this kind of synthetic traffic as "mice-and-elephants-like" or "web-like" traffic [MGT00, KHR02]. This simple approach is rather convenient for traffic generation, but it ignores the more complex patterns of connection usage (*e.g.*, bidirectionality, quiet times, *etc.*), and the differences among applications present in real Internet traffic.

It is important to note that recent work on the characterization of web traffic has improved our understanding of powerlaw/heavy-tailed modeling. Downey revisited the modeling of file sizes in [Dow01b] and of flow sizes in [Dow01a], suggesting that lognormal distributions are more appropriate than powerlaws (or heavy-tailed distributions). The historical survey by Mitzenmacher [Mit04] uncovered similar controversies in other fields, such as economics and biology. Hernández-Campos *et al.* demonstrated that lognormal distributions and powerlaws offer similar results in the regions of the distribution for which enough samples are available, specifically in the body and in the "moderate" tail. Beyond these regions, in the "far" tail, the lack of samples makes it impossible to choose between different models. This is because, for a fixed set of parameters and a fixed sample size equal to the original number of observations, some samplings of the lognormal and the powerlaw models match the original distribution, while other samplings do not. Hernández-Campos *et al.* also proposed the use of a mixture model (*i.e.*, a combination of several classical models), the double Pareto lognormal, which enables far more accurate fits than those achieved with Pareto or lognormal models. The inherently more flexible double Pareto lognormal model can capture the systematic deviations from simpler models that are commonly observed in the tails of the distributions of web object sizes. Nuzman *et al.* [NSSW02] modeled HTTP connection arrivals using the biPareto distribution, which provides a simpler but powerful alternative to mixture models. A Pareto distribution appears linear in a log-log scale, while the biPareto distribution shows two linear regions and a smooth transition between them. The biPareto distribution is therefore a generalization of the Pareto distribution.

The modeling efforts at Boston University culminated with the development of the SURGE model of web traffic [BC98]. The SURGE model described the behavior of each user as a sequence of web page downloads and think times between them. Each web page download consisted of one or more web objects downloaded from the same server. Barford and Crovella provided parametric fits for each of the components of the SURGE model, heavily relying on powerlaws and other long-tailed distributions. They also studied how SURGE traffic stressed web servers, and found SURGE's high burstiness far more demanding in terms of server CPU performance than that of less elaborate web traffic generators, such as the commercial WebStone.

A model of web traffic contemporary to SURGE was also presented by Mah [Mah97]. It described web traffic using empirical CDFs, which were derived from the analysis of packet header traces. As in the case of the SURGE model, the data came from the population of users in a computer science department. The two models were compared by Hernández-Campos *et al.* [HCJS03], showing substantial consistency.

The introduction of persistent connections in HTTP motivated further work on web traffic modeling. Barford *et al.* studied the performance implications of persistent connections [BC99], and modified the SURGE model to incorporate persistency [BBBC99]. The analysis of persistent connections was also a major topic in Smith *et al.* [SHCJO01] and Hernández-Campos *et al.* [HCJS03]. These studies were far larger in scope, focusing on the web traffic of an entire university rather than of a single department. These latter two works provided the starting point for the analysis method presented in this dissertation.

Many experimental studies made use of synthetic traffic generated according to one of the aforementioned web traffic models. For example, Christiansen *et al.* [CJOS00] made use of the Mah model, while Le *et al.* [LAJS03] used the Smith *et al.* model. The popular NS-2 [BEF$^+$00] network simulator also supports web traffic generation using models that are structurally similar to the SURGE model. This feature of NS was used in Joo *et al.* [JRF$^+$99, JRF$^+$01] to compare web traffic and infinite sources, and by Feldmann *et al.* [FGHW99] to study the impact of different parameters of the web traffic model on the burstiness of the generated traffic. Another web traffic generator available in NS-2 was developed by Cao *et al.* [CCG$^+$04]. Unlike other web traffic models, it was connection-oriented rather than user-oriented, and included non-source-level characteristics, such as packet sizes.

An important effort in web traffic analysis and generation was "Monkey See, Monkey Do" method, developed by Cheng *et al.* [CHC+04a]. The method involved recording source-level and network-level characteristics for each observed connection, and reproducing these characteristics using a synthetic workload generator. This idea is similar to the one developed in this dissertation, although we tackle the modeling and generation of entire traffic mixes and not just web traffic. In addition, their measurement methods were optimized for monitoring traffic near Google's web servers. The authors assumed independent short flows, data acquisition close to well-provisioned web servers, and no congestion in the client-to-server direction (which was plausible in the context of requests that were far smaller than responses).

2.2.2 Non-Web Traffic Source-level Modeling

Two prominent source-level modeling efforts took place before the invention of the World Wide Web. Danzig and Jamin [DJ91] developed tcplib, a collection of source-level descriptions of traffic. It included descriptions of the following applications:

- Telnet was described using three random variables: connection duration, packet inter-arrival time, and packet size. The initiator of the Telnet connection always sent one-byte packets, while the acceptor responded with packets matching the packet size distribution. The authors claimed that rlogin connections were also well-described by this model.
- File Transfer Protocol (FTP) was described using three random variables: number of items transferred, item size (*i.e.*, file size), and packet size. The model only described FTP-DATA transactions used to transfer a single file or a directory listing. It did not describe the FTP-Control connection that each client/server pair must use to manage each FTP-DATA transaction.
- Simple Mail Transfer Protocol (SMTP) was described using only one random variable: item size, which included size of mail message and address verification (*i.e.*, control) messages. Responses from the acceptor were considered negligible, and not modeled.
- Network News Transfer Protocol (NNTP) was described using two random variables: number of items transferred, and size of items (*i.e.*, NNTP articles). The bidirectional nature of the protocol and the use of control messages was not part of the model.

Tcplib also included a model of phone conversations with two random variables, talk spurt duration and quiet time (*i.e.*, pause) duration, borrowed from [Bra65]. Each random variable was specified using an empirical CDF. Traffic generation involved using the inverse transformation method [Jai91] to sample each empirical CDF independently.

In general, the application models in tcplib were rather simplistic, but they represented a giant step forward from the non-measurement-derived models of the early 90s. However, the use and capabilities of the modeled applications has dramatically changed since the development of tcplib. For example, the size of attachments in SMTP connections has dramatically increased due to the widespread implementation of Multipurpose Internet Mail Extensions (MIME). In addition, newer applications have become prominent or replaced the ones in tcplib. For example, the Telnet protocol has been mostly replaced by the Secure Shell (SSH) protocol. SSH is an encrypted protocol, so it requires more bytes per message. It also supports port forwarding, wherein other applications can communicate through SSH connections.

Paxson [Pax94] studied the same four applications as in tcplib, developing parametric models for each of them. Paxson also discussed how application characteristics change over time and across sites. This inherent variability motivated the use of parametric models, which are necessarily approximations of the empirical data. This approximation is not worse than the variability observed over time and across sites, so the author argued that parametric models were as accurate as empirical ones, but with the added benefits of being mathematically tractable and parsimonious. His analysis showed that bulk-transfer sizes were generally well-modeled by the log-normal distribution. Another of his findings was that connection inter-arrivals (except those of NNTP connections) were consistent with non-homogeneous Poisson arrivals, with fixed hourly rates.

The methodological contribution in Paxson's work is substantial. He demonstrated the difficulty of providing statistically valid parametric models of the distributions associated with Internet traffic. He consistently observed parametric fits that were clearly adequate when examined graphically, but that failed traditional goodness-of-fit tests. This was caused by the massive sample sizes, an endemic characteristics of traffic measurement datasets. As an alternative to the statistical tests, Paxson proposed the use of a goodness-of-fit metric, which provides a quantitative assessment of the distance between the empirical data and the parametric model. His proposed metric is however insensitive to deviations in the tails, casting doubt on the approach due to the ubiquitous finding of heavy-tailed phenomena in network traffic.

Web traffic quickly dominated most traffic mixes after its emergence in 1995, and remained the most prominent traffic type until file-sharing applications surpassed it in recent years. This motivated a large body of work on web traffic characterization, and little attention was paid to other traffic. The models developed by Danzig, Jamin and Paxson, were not improved or updated by other researchers.

File-sharing applications currently rival or frequently surpass web traffic in terms of traffic volume. They also represent a harder modeling problem than web traffic. The number of file-sharing applications is large and they use widely different communication strategies. Furthermore, the set of popular file-sharing applications is constantly changing. There is a growing body of traffic modeling literature focusing on file-sharing applications, but no traffic generator is yet available. Two prominent modeling studies were conducted at the University of Washington. Sariou *et al.* [SGG02] studied Napster and Gnutella traffic, and Gummadi *et al.* [GDS$^+$03] studied Kazaa traffic. Karagiannis *et al.* [KBBkc03] examined a larger set of file-sharing applications in backbone links.

Modeling of multimedia traffic has also received some attention. Variable bit-rate video was studied in Garret *et al.* and Knightly *et al.* [GW94, KZ97]. Real Audio traffic was studied by Mena and Heidemann [MH00], providing a first source-level view of streaming-media, mostly on UDP flows.

There are commercial synthetic traffic generation products such as Chariot [Inc] and IXIA 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 Internet traffic.

2.2.3 Beyond Single Application Modeling

The need for more representative traffic generation has motivated research on methods that can tackle the modeling of the entire suite of applications using an Internet link. The work in this dissertation lies in this area. Our preliminary steps were an extension of the methods used to model web traffic in Smith *et al.* [SHCJO01] to model other applications, as described in Hernández-Campos *et al.* [HCJS$^+$01]. The same kind of analysis of TCP header sequence numbers, acknowledgment numbers and connection quiet times applied to web traffic was used to populate models of SMTP and NNTP traffic. These models were derived from packet header traces collected at the University of North Carolina at Chapel Hill, and consisted of empirical distributions capturing different source-level characteristics of these protocols, such as object sizes. Lan and Heidemann [LH02] conducted a related effort, reusing the same techniques and software tools for data acquisition. Their RAMP tool populated models of web and FTP traffic directly from packet header traces, and generate traffic accordingly.

Harpoon [SB04] also tackled the same problem that is the focus of this dissertation. They considered the problem of analyzing entire traffic mixes and generating traffic accordingly. Their measurement methods were far less elaborate. Rather than the detailed models of the ADU exchange in TCP connections used in our work, Harpoon focused on modeling flows. Flows are defined as sets of packets with the same source and the same destination. As a consequence, Harpoon modeled each TCP connection as two unidirectional flows. Another difference with our approach is that Harpoon did not incorporate the notion of bidirectional data exchange, neither sequential nor concurrent, essentially treating multiple ADUs (as defined in the a-b-t model) as a single ADU. Idle times within connections were not part of the Harpoon traffic model either. In addition, any measured flow (*i.e.*, one side of a connection) with only a small amount of data or with only acknowledgment packets was not used for traffic generation. This substantially simplified the modeling, but it eliminated the rich packet-level dynamics observed in TCP connections, and demonstrated in later chapters of this dissertation. In addition to this, network-level parameters were not part of the data acquisition, so round-trip times and maximum receiver window sizes were arbitrarily chosen. Harpoon could also generate UDP traffic. The underlying model was to send packets at a constant bit rate, with either fixed or exponentially distributed interval arrivals. These models were not populated from measurement. Another novel feature of Harpoon was the ability to generate traffic that reproduced IP address structure according to a measured distribution of address frequency. Their study included a comparison between Harpoon's closed-loop traffic and traffic from a commercial (open-loop) packet-level traffic generator, demonstrating substantial differences. For example, closed-loop sources were shown to back off as congestion increases, while open-loop source did not. Like the work in this dissertation and Lan and Heidemann, Harpoon provided an automated method to acquire data and use it to generate traffic, which Sommers and Barford eloquently called "self-tuning" traffic generation. We could say that there is a growing consensus in the field of traffic generation regarding the need to develop tools that combine measurement and generation to tackle the wide variability over time and across links found in real Internet traffic.

2.3 Scaling Offered Load

One of the key requirements of traffic generation is the ability to *scale the offered load*, *i.e.*, to generate a wide range of link loads with the same model of application behavior. This makes it possible to evaluate the performance of a network mechanism under various loads, which translates into different degrees of congestion, while preserving the same application mix. For example, the evaluation of AQM mechanism in [CJOS00, LAJS03] compared the performance of FIFO to RED and other AQM mechanisms for loads between 50% to 110% of a link's capacity where the queuing mechanism was used. In these studies, the authors preceded their study by a set of *calibration* experiments. These experiments were used to derive an expression for the linear dependency between the number of (web) user equivalents and the average offered load, which enabled the researchers to systematically scale offered loads in their evaluation experiments. Calibration is generally applicable to any application-level model. When calibrating, the researchers try to relate one or more parameters of the model and the average offered load to obtain a *calibration function*. Deriving a calibration function is a time-consuming process, since an entire collection of experiments must be run to correlate offered load and model parameters with confidence.

Kamath *et al.* [KcLH$^+$02] studied load scaling methods, but they concentrated only on scaling up the offered load. Their intention was to conduct experiments with much higher offered loads than those observed during measurement. In particular, they considered the problem of generating traffic for loading a 1 Gbps link using only measurements from a 10 Mbps link, an 11-hour packet header trace. The authors considered three different techniques. The first two techniques involved a transformation of the original trace into a scaled-up version, and then a packet-level replay. The first transformation technique was *packet arrival scaling*, which scales up the load by multiplying the arrival time of each packet in the original trace by a constant factor between 0 and 1 (*i.e.*,,shrinking packet inter-arrivals). In their study, they used a scaling factor of 0.001. The second transformation technique is *trace merging*, which scales up load by merging, *i.e.*, superimposing, the packet arrivals from more than one trace. They divided the 11-hour trace into 100 subtraces and then combined them to form a shorter, higher-throughput trace. The third technique is *structural modeling* which meant to develop a web traffic model from the original trace using the methods in Smith *et al.* [SHCJO01]. The authors did not discuss how the load created by this structural model was increased. Their analysis compared a number of distributions from the generated traces to those from the original trace. Packet arrival scaling was shown to completely distort flow durations and destination address diversity. Trace merging reproduced flow and packet arrival properties accurately, but it distorted destination address characteristics (studied using the number of unique addresses observed per unit of time). Web traffic generation was accurate, but it showed far less complex distributions of connection bytes, packet sizes, and connection durations. This is because a structural model based only on web traffic lacks the diversity of application behavior, and therefore communication patterns, in the original trace, which included traffic from many different applications and not just web traffic.

2.4 Implementing Traffic Generation

Source-level traffic generators for network testbeds (rather than for software simulators) are usually implemented using user-level programs that make use of the socket interface to generate traffic. This is the case for *tcplib* [DJ91], *httperf* [MJ98], SURGE [BC98], and other web traffic generators [BD99, CJOS00]. In order to introduce network-level parameters in test-bed experiments, such as a realistic distribution of round-trip times, it is necessary to rely on a layer of simulation either in the end hosts or somewhere in the path of the traffic. For example, Rizzo's *dummynet* [Riz97] makes it possible to apply arbitrary delays, loss rates and bandwidth constraints on the end systems to specific network flows or collections of network flows (that share a network prefix). The implementation combines event-driven simulation and packet queuing, and sits between the IP and link layers. Dummynet is part of the standard distribution of the FreeBSD operating system. The experiments in this dissertation were performed using an extended version of dummynet that can be controlled from the application layer[2].

Kamath *et al.* [KcLH+02] argue that source-level traffic generation is much more demanding in terms of CPU and memory processing than packet-level replay. While it is indeed true that far more CPU time is needed to simulate endpoint behavior and use network stacks, memory requirements are actually far more stringent for packet-level replay. This is because packet header traces are much longer than their source-level representations. For example, the approach in this dissertation considers the replay of source-level traces that are roughly 100 times smaller than the packet header traces from which they were derived.

2.5 Summary

Our review of related work has focused on the existing literature in network traffic generation, including works relevant for data acquisition and traffic modeling. Characterizing network traffic at the packet level provides important insights, such as the finding of pervasive self-similarity by Willinger *et al.* [WTSW97]. However, this approach does not provide the proper foundation for generating traffic for most experimental studies. As argued by Floyd and Paxson [PF95], packet-level traffic generation breaks the end-to-end feedback loop in adaptive network protocols, such as TCP, resulting in traffic that does not react to the experimental conditions realistically. On the contrary, source-level models enable

[2] This is also possible in the original implementation, using one firewall rule for each flow, but it does not scale to the hundreds of simultaneous flows in our experiments.

closed-loop traffic generation, so they are applicable to a wider range of situations.

In the past, source-level traffic generation has been associated with models of application behavior. Our overview of the state-of-the-art discussed several highly influential works devoted to application-level modeling. Cáceres *et al.* [CDJM91] introduced empirical application models to networking research. Paxson [Pax94] proposed the use of more statistically rigorous methods for developing parametric source-level models. Crovella *et al.* [CB96] developed a rich model of web traffic, and explained self-similarity in terms of source-level characteristics.

Application-level modeling has some important shortcomings that provide the motivation for this dissertation. Internet traffic mixes are created by a large number of distinct applications, so single application models are not representative of real traffic. Furthermore, the composition of traffic mixes is constantly changing, and even individual applications often evolve, modifying the way in which they interact with the network. As a consequence, the number of high-quality application-level models is small (and insufficient), and these models are hardly ever updated. In this dissertation, we propose a more scalable approach to source-level modeling, where application behavior is described in a generic, but still detailed, manner. Furthermore, our data acquisition methods are efficient and mostly automated, dramatically reducing the time to go from measurement to traffic generation.

Our combination of data acquisition and traffic generation is most closely related to two contemporary works. Sommers and Barford [SB04] developed the Harpoon approach for generating traffic mixes whose characteristics are derived from measurements in an algorithmic manner. Their approach did not include any detailed source-level modeling of TCP connections. They described a connection simply as a unidirectional file transfer whose size is equal to the total amount of payload in its packets. In contrast, our primary emphasis is on detailed source-level modeling, where we introduce the a-b-t model and uncover the dichotomy between sequential and concurrent data exchange. Harpoon made use of simplified network-level parameters, which are set to arbitrary constants. In our approach, network-level parameters are carefully measured and incorporated into the traffic generation. The work by Sommers and Barford considered two issues that are not addressed in our own work. First, they proposed a method for generating UDP traffic. The underlying source-level model is however not derived from measurement. Second, they reproduced the IP address distribution in the replayed trace. This cannot be performed with publicly available traces, like ours, since they are anonymized.

Another work similar to ours is Cheng *et al.* [CHC^{+}04a]. The authors presented a method for characterizing packet header traces of web traffic and accurately replaying them. Generated traffic was evaluated by comparing the original trace with its synthetic version generated in a testbed. We tackle the same source-level trace replay problem but applied to every application rather than only to web traffic. Our approach is more ambitious and necessarily more abstract.

Our work also considers the common problems of resampling and scaling traffic load in networking experiments. In general, scaling offered load has been performed by conducting a preliminary experimental study to relate the parameters of the source-level model and the offered load. For example, Christiansen *et al.* [CJOS00] computed a calibration function that described offered load as a function of the number of user equivalents employed in web traffic generation. We propose an alternative approach that eliminates the need for preliminary calibration studies.

CHAPTER 3

Abstract Source-level Modeling

***model**: (11a) a description or analogy used to help visualize something (as an atom) that cannot be directly observed.*

— Merrian–Webster English Dictionary

Anything that has real and lasting value is always a gift from within.

— Franz Kafka (1883–1924)

Abstract source-level modeling provides a method to describe the workload of a TCP connection at the source level in a manner than is not tied to the specifics of individual applications. The starting point of this method is the observation that at the transport level, a TCP endpoint is doing nothing more than sending and receiving data. Each application (*i.e.*, web browsing, file sharing, *etc.*) employs its own set of data units for carrying application-level control messages, files, and other information. The actual meaning of the data is irrelevant to TCP, which is only responsible for delivering data in a reliable, ordered, and congestion-responsive manner. As a consequence, we can describe the workload of TCP in terms of the demands by upper layers of the protocol stack for sending and receiving *Application Data Units* (ADUs). This workload characterization captures only the sizes of the units of data that TCP is responsible for delivering, and abstracts away the details of each application (*e.g.*, the meaning of its ADUs, the size of the socket reads and writes, *etc.*). The approach makes it feasible to model the entire range of TCP workloads, and not just those that derive from a few well-understood applications as is the case today. This provides a way to overcome the inherent scalability problem of application-level modeling.

While the work of a TCP endpoint is to send and receive data units, its lifetime is not only dictated by the time these operations take, but also by *quiet times* in which the TCP connection remains idle, waiting for upper layers to make new demands. TCP is only affected by the duration of these periods of inactivity and not by the cause of these quiet times, which depends on the dynamics of each application (*e.g.*, waiting for user input, processing a file, *etc.*). Longer lifetimes have an important impact, since the endpoint resources needed to handle TCP state must remain reserved for a longer period of time[1]. Furthermore, the window mechanism in TCP tends to aggregate the data of those ADUs that are sent within a short period of time, reducing the number of segments that have to travel from source to destination. This is only possible when TCP receives a number of back-to-back requests to send data. If these requests are separated by significant quiet times, no aggregation occurs and the data is sent using at least as many segments as ADUs.

We have formalized these ideas into the *a-b-t model*, which describes TCP connections as sets of ADU exchanges and quiet times. The term a-b-t is descriptive of the basic building blocks of this model: *a-type* ADUs (*a*'s), which are sent from the connection initiator to the connection acceptor, *b-type* ADUs (*b*'s), which flow in the opposite direction, and quiet times (*t*'s), during which no data segments are exchanged. We will make use of these terms to describe the source-level behavior of TCP connections throughout this dissertation. The a-b-t model has two different flavors depending on whether ADU interleaving is sequential or concurrent. The *sequential a-b-t model* is used for modeling connections in which only one ADU is being sent from one endpoint to the other at any given point in time. This means that the two endpoints engage in an orderly conversation in which one endpoint will not send a new ADU until it has completely received the previous ADU from the other endpoint. On the contrary, the *concurrent a-b-t model* is used for modeling connections in which both endpoints send and receive ADUs simultaneously.

The a-b-t model not only provides a reasonable description of the workload of TCP at the source-level, but it is also simple enough to be populated from measurement. Control data contained in TCP headers provide enough information to determine the number and sizes of the ADUs in a TCP connection and the durations of the quiet times between these ADUs. This makes it possible to convert an arbitrary trace of segment headers into a set of *a-b-t connection vectors*, in which each vector describes one of the TCP connections in the trace. As long as this process is accurate, this approach provides realistic characterizations of TCP workloads, in the sense that they can be empirically derived from measurements of real Internet links.

In this chapter, we describe the a-b-t model and its two flavors in detail. For each flavor, we first discuss a number of sample connections that illustrate the power of the a-b-t model to describe TCP connections driven by different applications, and point out some limitations of this approach. We then present a set of techniques for analyzing segment headers in order to construct a-b-t connection vectors and provide a validation of these techniques using traces from synthetic applications. We finally examine

[1] Similarly, if resources are allocated along the connection's path, they must be committed for a longer period.

the characteristics of a set of real traces from the point of view of the a-b-t model, providing a source-level view of the workload of TCP.

3.1 The Sequential a-b-t Model

3.1.1 Client/Server Applications

The a-b-t connection vector of a sequential TCP connection is a sequence of one or more *epochs*. Each epoch describes the properties of a pair of ADUs exchanged between the two endpoints. The concept of an epoch arises from the client/server structure of many distributed systems, in which one endpoint acts as a client and the other one as a server. The client sends a request for some service (*e.g.*, performing a computation, retrieving some data, *etc.*) that is followed by a response from the server (*e.g.*, the results of the requested action, a status code, *etc.*). An epoch represents our abstract characterization of a request/response exchange. An epoch is characterized by the size a of the request and the size b of the response.

The HTTP that underlines the World-Wide Web provides a good example of the kinds of TCP workloads created by client/server applications. Figure 1 shows a simple *a-b-t diagram* that represents a TCP connection between a web browser and a web server, which communicate using the HTTP 1.0 application-layer protocol [BLFF96]. In this example, the web browser (client side) initiates a TCP connection to a web server (server side) and sends a request for an object (*e.g.*, HTML source code, an image, *etc.*) specified using a Universal Resource Locator (URL). This request constitutes an ADU of size 341 bytes. The server then responds by sending the requested object in an ADU of size 2,555 bytes. The representation in the figure captures:

- the sequential order of the ADUs within the TCP connection (first the HTTP request then the HTTP response – in this case, order also implies "causality"),

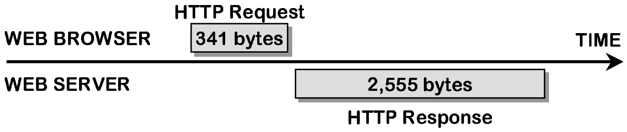


Figure 3.1: An a-b-t diagram representing a typical ADU exchange in HTTP version 1.0.

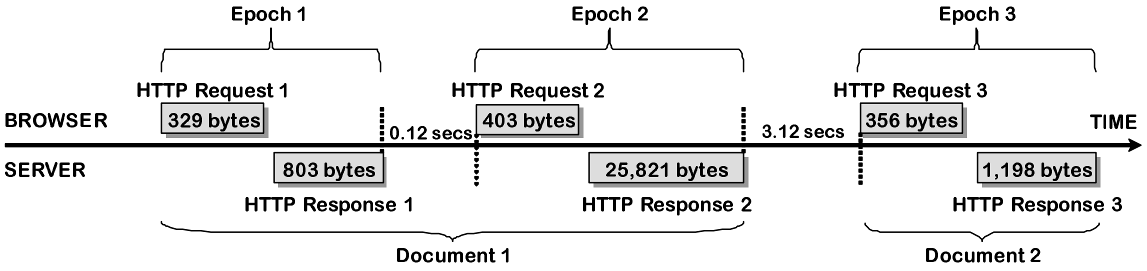


Figure 3.2: An a-b-t diagram illustrating a persistent HTTP connection.

- the direction in which the ADUs flow (above the time line for the ADU sent from the connection initiator to the connection acceptor; below the time line for the ADU sent from the connection acceptor to the connection initiator), and
- the sizes of the ADUs (using annotations and the lengths of the rectangles, which are proportional to the number of bytes).

The diagram provides a visualization in the spirit of abstract source-level modeling, since it does not incorporate any specific information about the actual contents of the ADUs. The bytes in the first ADU (HTTP request) represent an HTTP header that includes a URL, and the bytes in the second ADU (HTTP response) represent an HTTP header (with a success code of `200 OK`) followed by the requested object (*e.g.*, HTML source code). In this example, the purpose of this particular connection was well-understood, and that allowed us to assign labels to the ADUs (HTTP request and response) and to the TCP endpoints (web browser and server). In general, when we examine how the ADUs flow in an arbitrary TCP connection, we do not have this application-specific information (or we can only guess it). The same diagram (without the HTTP-specific labels) could be used to represent different connections with completely different payloads in ADUs of the same size. The diagram does not include any network-level information either, so this diagram could also represent connections with very different maximum segment sizes, round-trip times, and other network properties below the application level. Note that this example, and the following ones, came from real connections that were actually observed. In some cases, we had access to the actual segment payloads and used them to add annotations to the ADUs. In other cases, we used port numbers and our understanding of the protocols to add these annotations.

Some client/server applications use a new connection for each request/response exchange, while other applications reuse a connection for more than one exchange, creating connections with more than one epoch. As long as the application has enough data to send, multi-epoch connections can improve performance substantially, by avoiding the connection establishment delay and TCP's slow start phase. For example, HTTP was revised to support more than one request/response exchange in the same "persistent" TCP connection [FGM$^+$97]. Figure 3.2 illustrates this type of interaction. This is a connection between a web browser and a web server, in which the browser first requests the source code of an HTML page, and receives it from the web server, just like in Figure 3.1. However, the use of persistent HTTP makes it possible for the browser to send another request using the same connection. Unlike the example in Figure 3.1, this persistent connection remains open after the first object is downloaded, so the browser can send another request without first closing the connection and reopening a new one. In Figure 3.2 the web browser sends three ADUs that specify three different URLs, and the server responds with three ADUs. Each ADU contains an HTTP header that precedes the actual requested object. If the requested object is not available, the ADU may only contain the HTTP header with an error code. Note that the diagram has been annotated with extra application-level information showing that the first two epochs were the result of requesting objects from the same document (*i.e.*, same web page), and the last epoch was the result of requesting a different document.

The diagram in Figure 3.2 includes two time gaps between epochs (represented with dashed lines). In both cases, these are quiet times in the interaction between the two endpoints. We call the time between the end of one epoch and the beginning of the next, the *inter-epoch quiet time*. The first quiet time in the a-b-t diagram represents processing time in the web browser, which parsed the web page it received, retrieved some objects from the local cache, and then made another request for an object in the same document (that was not in the local cache). Because of its longer duration, the second quiet time is most likely due to the time taken by the user to read the web page, and click on one of the links, starting another page download from the same web server.

As will be discussed in Section 3.3, it is difficult to distinguish quiet times caused by application dynamics, which are relevant for a source-level model, and those due to network performance and characteristics, which should not be part of a source-level model (because they are not caused by the behavior of the application). The basic heuristic employed to distinguish between these two cases is the observation that the scale of network events is hardly ever above a few hundred milliseconds[2]. Going back to the example in Figure 3.2, the only quiet time that could be safely assumed to be due to the application (in this case, due to the user) is the one between the second and third epochs. The 120 milliseconds quiet time between the first and second epochs could easily be due to network effects

[2] Some infrequent events, such as routing changes due to link failures, can last several seconds. We generally model large numbers of TCP connections, so the few occasions in which we confuse application quiet times with long network quiet times have no measurable statistical impact when generating network traffic.

(such as having the sending of the second request delayed by Nagle's algorithm [Nag84]), and therefore should not be part of the source-level behavior. Similarly, the two a-b-t diagrams shown so far have not depicted any time between the request and the response inside the same epoch. In general, web servers process requests so quickly that there is no need to incorporate *intra-epoch quiet times* in a model of the workload of a TCP connection. While this is by far the most common case, some applications do have long intra-epoch quiet times, and the a-b-t model can include these.

Formally, a sequential a-b-t connection vector has the form $C_i = (e_1, e_2, \ldots, e_n)$ with $n \geq 1$ epoch tuples. An epoch tuple has the form $e_j = (a_j, ta_j, b_j, tb_j)$ where

- a_j is the size of the j^{th} ADU sent from the connection initiator to the connection acceptor. a_j will also be used to name the jth ADU sent from the initiator to the acceptor.
- b_j is the size of the j^{th} ADU sent in the opposite direction (and generally in response to the request made by a_j).
- ta_j is the duration of the quiet time between the arrival of the last segment of a_j and the departure of the first segment of b_j. ta_j is defined from the point of view of the acceptor (often the server), but ultimately our estimate of the duration is based on the arrival times of segments at some monitoring point.
- tb_j is either the duration of the quiet time between b_j and a_{j+1} (for connections with at least $j+1$ epochs), or the quiet time between the last data segment (*i.e.*, last segment with a payload) in the connection and the first control segment used to terminate the connection.

Note that ta_j is a quiet time as seen from the acceptor side, while tb_j is a quiet time as seen from the initiator side. The idea of these definitions is to capture the network-independent component of quiet times, without being concerned with the specific measurement method. In a persistent HTTP connection, a's would usually be associated to HTTP requests, b's to HTTP responses, ta's to processing times on the web server, and tb's to browser processing times and user think times. We can say that a quiet time ta_j is "caused" by an ADU a_j, and that a quiet time tb_j is caused by an ADU b_j. Both time components are defined as quiet times observed at one of the endpoints, and not at some point in the middle of the network where the packet header tracing takes place.

As mentioned in the introduction, the name of the model comes from the three variable names used in this model, which are used to capture the essential source-level properties: data in the "a" direction, data

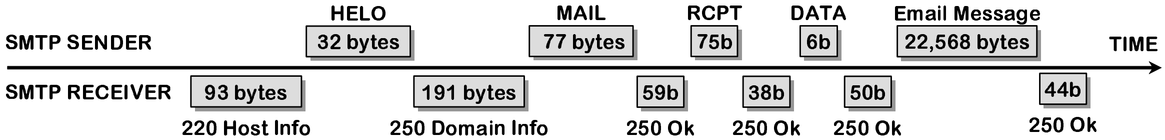


Figure 3.3: An a-b-t diagram illustrating an SMTP connection.

in the "b" direction, and time "t" (non-directional, but associated with the processing of the preceding ADU, as discussed in Section 3.1.1). Using the notation of the a-b-t model, we can succinctly describe the HTTP connection in Figure 3.1 as a single-epoch connection vector of the form

$$((341, 0, 2555, 0))$$

where the first ADU, a_1, has a size of 341 bytes, and the second ADU, b_1, has a size of 2,555 bytes. In this example the time between the transmission of the two data units and the time between the end of b_1 and connection termination are considered too small to be included in the source level representation, so they are set to 0. Similarly, we can represent the persistent HTTP connection shown in Figure 3.2 as

$$((329, 0, 403, 0.12), (403, 0, 25821, 3.12), (356, 0, 1198, 15.3))$$

where quiet times are given in seconds. Notice that tb_3 is not zero for this connection, but a large number of seconds (in fact, probably larger than the duration of the rest of the activity in the connection!). Persistent connections are often left open in case the client decides to send a new HTTP request reusing the same TCP connection[3]. As we will show in Section 3.5, this separation is frequent enough to justify incorporating it in the model. Gaps between connection establishment and the sending of a_1 are almost nonexistent.

As another example, the Simple Mail Transfer Protocol (SMTP) connection in Figure 3.3 illustrates a sample sequence of data units exchanged by two SMTP servers. The first server (labeled "sender") previously received an email from an email client, and uses the TCP connection in the diagram to contact the destination SMTP server (*i.e.*, the server for the domain of the destination email address). In this example, most data units are small and correspond to application-level (SMTP) control messages (*e.g.*, the host info message, the initial HELO message, *etc.*) rather than application objects. The actual email

[3] In general, persistent HTTP connections are closed by web servers after a maximum number of request/response exchanges (epochs) is reached or a maximum quiet time threshold is exceeded. By default, Apache, the most popular web server, limits the number of epochs to 5 and the maximum quiet time to 15 seconds.

message of 22,568 bytes was carried in ADU a_6. The a-b-t connection vector for this connection is

$$((0,0,93,0),(32,0,191,0),(77,0,59,0),(75,0,38,0),(6,0,50,0),(22568,0,44,0)).$$

Note that this TCP connection illustrates a variation of the client/server design in which the server sends a first ADU identifying itself without any prior request from the client. This pattern of exchange is specified by the SMTP protocol wherein servers identify themselves to clients right after connection establishment. Since b_1 is not preceded by any ADU sent from the connection initiator to the connection acceptor, the vector has $a_1 = 0$ (we sometimes refer to this phenomenon as a “half-epoch”).

This last example illustrates an important characteristic of TCP workloads that is often ignored in traffic generation experiments. TCP connections do not simply carry files (and requests for files), but are often driven by more complicated interactions that impact TCP performance. An epoch where $a_j > 0$ and $b_j > 0$ requires at least one segment to carry a_j from the connection initiator to the acceptor, and at least another segment to carry b_j in the opposite direction. The minimum duration of an epoch is therefore one round-trip time (which is precisely defined as the time to send a segment from the initiator to the acceptor plus the time to send a segment from the acceptor back to the initiator). This means that the number of epochs imposes a minimum duration and a minimum number of segments for a TCP connection. The connection in Figure 3.3 needs 4 round-trip times to complete the “negotiation” that occurs during epochs 2 to 5, even if the ADUs involved are rather small. The actual email message in ADU b_6 is transferred in only 2 round-trip times. This is because b_6 fits in 16 segments[4], and it is sent during TCP’s slow start. Thus the first round-trip time is used to send 6 segments, and the second round-trip time is used to send the remaining 10 segments. The duration of this connection is therefore dominated by the control messages, and not by the size of the email. In particular, this is true despite the fact that the email message is much larger than the combined size of the control messages. If the application protocol (*i.e.*, SMTP) were modified to somehow carry control messages and the email content in ADU a_2, then the entire connection would last only 4 round-trip times instead of 6, and would require fewer segments. In our experience, it is common to find connections in which the number of control messages is orders of magnitude larger than the number of ADUs from files or other dynamically-generated content. Clearly, epoch structure has an impact on the performance (more precisely, on the duration) of TCP connections and should therefore be modeled accurately.

Application protocols can be rather complicated, supporting a wide range of interactions between

[4] This assumes the standard maximum segment size, 1,460 bytes, and a maximum receiver window of at least 10 full size segments. A large fraction of TCP connections observed on real networks satisfy these assumptions.

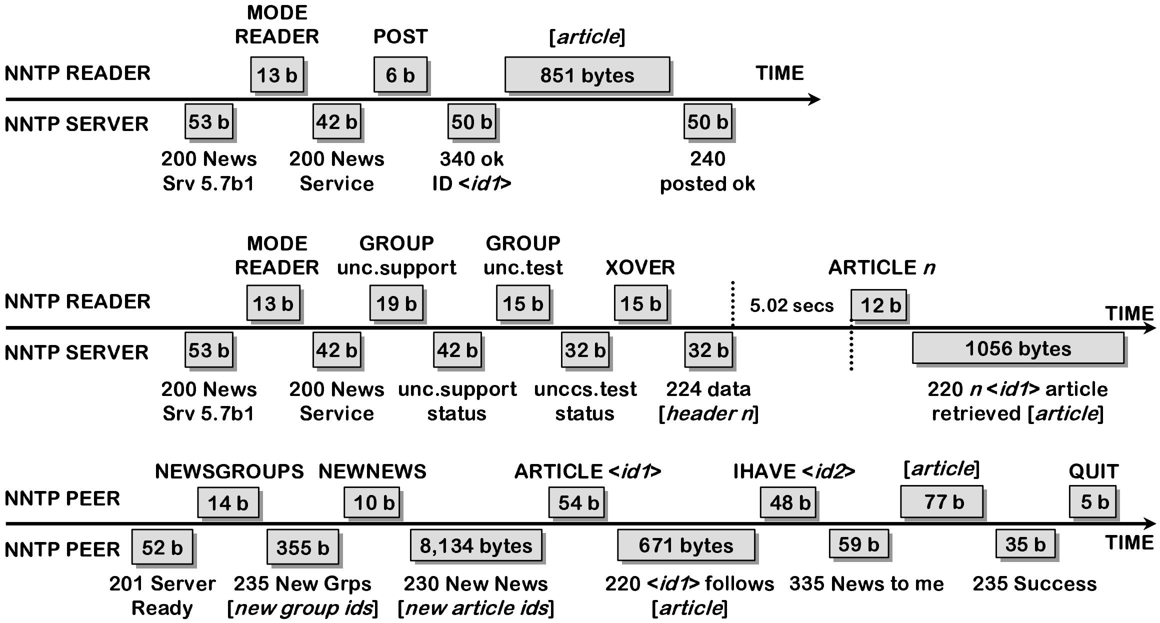


Figure 3.4: Three a-b-t diagrams representing three different types of NNTP interactions.

the two endpoints. Most of them assume a client/server model of interaction and hence can be cast into the sequential a-b-t model. For example, Figure 3.4 shows three types of interactions that are supported by the Network News Transfer Protocol (NNTP) [KL86, Bar00]. The first a-b-t diagram exhibits the straightforward behavior of an NNTP reader (*i.e.*, a client for reading newsgroup postings) posting a new article. The two endpoints exchange a few control messages in the first three epochs, and then the client uploads the content of the article in ADU a_4.

The second connection shows an NNTP reader using a TCP connection to first check whether the server knows about any new articles in two newsgroups (unc.support and unc.test). After that, the reader requests an overview of those messages (using XOVER). The server replies with the subjects of the new articles and some other information. Finally, after a 5.02 seconds of inactivity, the reader requests the content of one of the new articles. This relatively long time suggests that the user of the NNTP reader waited some time before actually requesting the reader to display the content of a new article.

The way NNTP servers interact is illustrated in the third connection. One of the peers will ask the other about new newsgroups and articles. This typically involves hundreds or even thousands of ADUs sent in each direction. The connection shown here has only a small subset of the ADUs observed in one of these connections between NNTP peers. Here the initiator peer asked for new groups first, and then for new articles. One article was sent from the initiator to the acceptor, and another one in the opposite direction.

These examples provide a good illustration of the complexity of modeling applications one by one, and they provide further evidence supporting the claim that our abstract source-level model is widely applicable. In general, the use of a multi-epoch model is essential to accurately describe how applications drive TCP connections.

Incorporating Quiet Times into Source-Level Modeling

Unlike ADUs, which flow from the initiator to the acceptor or *vice versa*, quiet times are not associated with any particular direction of a TCP connection. However, we have chosen to use two types of quiet times in our sequential a-b-t model. This choice is motivated by the intended meaning of quiet time, and by the difference between the duration of the quiet times observed at different points in the connection's path. When we were developing the model, we initially considered quiet times independent of the endpoint causing them. They were simply "connection quiet times". In practice, quiet times in sequential connections are associated with source-level behavior in only one of the endpoints. For example, a "user think time" in an HTTP connection is associated with a quiet time on the initiator side (which is waiting for the user action), while a server processing delay in a Telnet connection is associated with the acceptor side (which is waiting for a result). In every case, one endpoint is quiet for some period before sending new data, and the other endpoint remains quiet, waiting for these new data to arrive. Having two types of quiet times, ta and tb, makes it possible to annotate the side of the connection that is the source of the quiet time.

The second reason for the use of two types of quiet times is that the duration of the quiet time depends on the point at which the quiet time is measured. The endpoint that is not the source of the quiet time will observe a quiet time that depends on the network and not only on the source-level behavior of the other endpoint. This is because the new ADU which defines the end of the quiet time needs some time to reach its destination. In the example in Figure 3.2, the quiet time between a_1 and b_1 observed by the server endpoint is very small (only the time needed to retrieve the requested URL). However, this quiet time is longer when observed by the client, since it is the time between the last socket write of a_1 and the first socket read of b_1. It includes the server processing time, and at least one full round-trip time. Ideally, we would like to measure this quiet time ta_1 on the server side, in order to characterize source-level behavior in a completely network-independent manner. Similarly, we

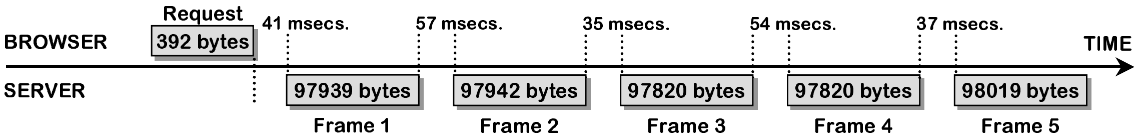


Figure 3.5: An a-b-t diagram illustrating a server push from a webcam using a persistent HTTP connection.

would like to measure tb_1 on the client side. In summary, source-level quiet times are non-directional, in the sense that they do not travel in one direction or the other, but they are associated with one of the endpoints, which is the source of the quiet time.

3.1.2 Beyond Client/Server Applications

Not all applications follow the strict pattern of requests and responses that characterizes traditional client/server applications. For example, HTTP is commonly used for server push operations[5], in which the server periodically refreshes the state of the client without any prior request. Figure 3.5 illustrates this behavior using a TCP connection where a web browser first requests a webcam URL (UNC's "Pitcam" in this example), and the web server responds with a sequence of image frames separated by small quiet times. The browser renders each frame as soon as it is received, creating a continuous movie. Each frame can be considered an individual ADU, so this connection does not follow the basic request/response sequence of previous examples. The notation provided by the sequential a-b-t model can still be used to represent this source-level behavior using the connection vector $(e_1, e_2, e_3, e_4, e_5)$ where $e_1 = (392, 0.041, 97939, 0)$, $e_2 = (0, 0.057, 97942, 0)$, $e_3 = (0, 0.035, 97820, 0)$, $e_4 = (0, 0.054, 97820, 0)$, and $e_5 = (0, 0.037, 98019, 0)$. While this connection has no natural epochs in the request/response sense, we can describe the connection by assigning each frame to a separate b_j, and each quiet time between frames to a ta_j (since the connection vector is intended to capture a quiet time on the server side).

The same type of server push behavior is found in streaming applications. A TCP connection carrying Icecast traffic (from `ibiblio.org`) is shown in Figure 3.6. Icecast is a popular audio streaming application that follows the same pattern of ADUs discussed in the previous paragraph, and can be

[5] HTTP server push is implemented using a special content type, `x-mixed-replace`, which makes the browser expect a response object that is composed of other objects (separated by a configurable boundary string). Since no limit is imposed on the number of objects in this composite, webcam movies are usually implemented as a simple sequence of JPEG images that the web browser reads and renders continuously until the user moves to another page. This type of web service should not be confused with HTML's automatic page refresh tag, which is commonly used for slow rate webcams (*e.g.*, one image every 30 seconds). In this case, the browser refreshes the current page by downloading again the current page and hence the interaction follows the regular request/response pattern.

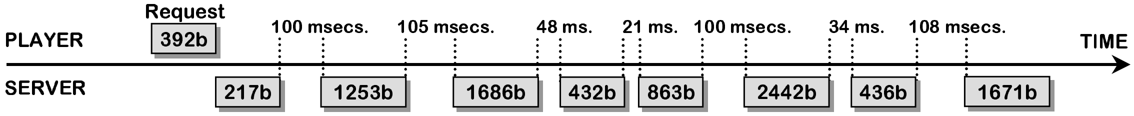


Figure 3.6: An a-b-t diagram illustrating Icecast audio streaming in a TCP connection.

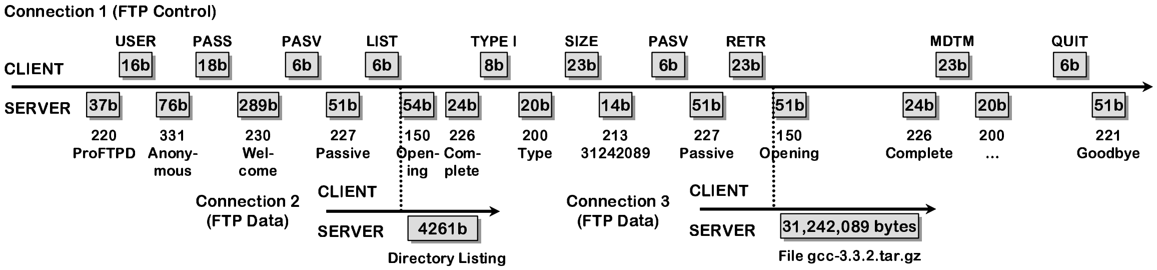


Figure 3.7: Three a-b-t diagrams of connections taking part in the interaction between an FTP client and an FTP server.

described using the same type of connection vector. Each b_j is associated to an MPEG audio frame. Note that the sizes of the ADUs and the durations of the quiet times between them are highly variable, unlike the example in Figure 3.5. Perhaps surprisingly, TCP is widely used for carrying streaming traffic today, despite its inability to perform the typical trade-off between loss recovery and delay in multimedia applications. Streaming over TCP has two significant benefits:

- Streaming traffic can use TCP port numbers associated with web traffic and therefore overcome firewalls that block other port numbers. This is important for web sites that deliver web pages and multimedia streams, since it guarantees that the user will be able to download the multimedia content.
- Most clients experience such low loss rates, that TCP's loss recovery mechanisms have an insignificant impact on the timing of the stream. The common use of stream buffering prior to the beginning of the playback further reduces the impact of loss recovery.

The interaction between the two endpoints of a client/server application does not generally require more than one TCP connection to be opened between the two endpoints. As we have seen, some applications use a new connection for each request/response exchange, while others make use of multi-epoch connections (*e.g.*, persistent connections in HTTP/1.1). Handling more than one TCP connection can have some performance benefits, but it does complicate the implementation of the applications (*e.g.*, it may require using concurrent programming techniques). However, some applications do interact using several TCP connections and this creates interdependencies between ADUs. For example, Figure 3.7 illustrates an FTP session[6] between an FTP client program and FTP server in which three connections are used. The connection in the top row is the "FTP control" connection used by the client to first identify itself (with username and password), then list the contents of a directory, and then retrieve a large file. The actual directory listing and the file are received using separate "FTP data" connections (established by the client) with a single ADU b_1. The figure illustrates how the start of the data connections depends on the use of some ADUs in the control connection (*i.e.*, the directory listing LIST does not occur until after the RETR ADUs has been received), and how the control connection does not send the `226 Complete` ADU until the data connections have completed.

While the sequential a-b-t model can accurately describe the source-level properties of these three connections, the model cannot capture the interdependency between the connections. The FTP example in Figure 3.7 shows three connections with a strong dependency. The two FTP data connections necessarily followed a `150 Opening` operation in the FTP control connection. Our current model cannot express this kind of dependencies between connections or between the ADUs of more than one connection. It would be possible to develop a more sophisticated model capable of describing these types of dependencies, but it seems very difficult to populate such a model from traces in an accurate manner without knowledge of application semantics. As an alternative, the traffic generation approach proposed in this dissertation carefully reproduces relative differences in connection start times, which tend to preserve temporal dependencies between connections. Our experimental results also suggest that the impact of interconnection dependencies is negligible, at least for our collection Internet traces.

3.2 The Concurrent a-b-t Model

In the sequential model we have considered so far, application data is either flowing from the client to the server or from the server to the client. However, some TCP connections are not driven by this traditional style of client/server interaction. Some applications send data from both endpoints of the connection at the same time. Figure 3.8 shows an NNTP connection between two NNTP peers (servers) in which NNTP's "streaming mode" is used. As shown in the diagram, ADUs b_5 and b_6 are sent from the connection acceptor to the connection initiator while ADU a_6 is being sent in the opposite direction.

[6] This is an abbreviated version of the original session, in which there was some directory navigation and more directory listings. The control connection used port 21, while the data connections used dynamically selected port numbers. Note also that significant inter-ADU times due to user think time are not shown in the diagram.

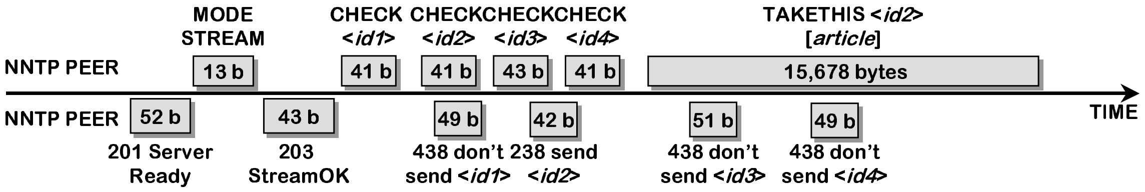


Figure 3.8: An a-b-t diagram illustrating an NNTP connection in "stream-mode", which exhibits data exchange concurrency.

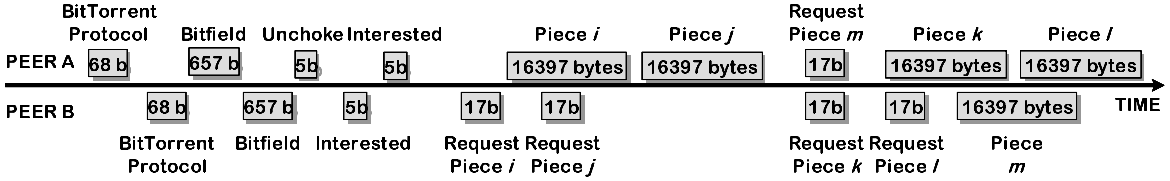


Figure 3.9: An a-b-t diagram illustrating the interaction between two BitTorrent peers.

ADUs b_5 and b_6 carry 438 messages, where the acceptor NNTP peer tells the initiator that it is not interested in articles id3 and id4. ADU a_6 carried article id2 in the opposite direction. There is no causal dependency between these ADUs, which make it possible for the two endpoints to send data independently. Therefore this connection is said to exhibit *data exchange concurrency* in the sense that one or more pairs of ADUs are exchanged simultaneously. In contrast, the connections illustrated in previous figures exchanged data units in a sequential fashion. A fundamental difference between these two types of communication patterns is that sequential request/response exchanges (*i.e.*, epochs) always take a minimum of one round-trip time. Data exchange concurrency makes it possible to send and receive more than one ADU per round-trip time, and this can increase throughput substantially. In the figure, the initiator NNTP peer is able to send check requests to the other party quicker because it can do so without waiting for the corresponding responses, each of which would take a minimum of one full round-trip time to arrive.

Another example of concurrent data exchange is shown in Figure 3.9. Here two BitTorrent peers [Coh03] exchange pieces of a large file that both peers are trying to download. The BitTorrent protocol supports the backlogging of requests (*i.e.*, pieces k and m of the file are requested before the download of the preceding piece is completed), and also the simultaneous exchange of file pieces (*i.e.*, the transmission of pieces k and l of the file coexist with the transmission of piece m). As discussed above, this type of behavior helps to avoid quiet times in BitTorrent connections, thereby increasing average throughput. Furthermore, this example illustrates a type of application in which both endpoints act as client and server (both request and receive file pieces).

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. Rather than waiting for the response of the last request to arrive, the client keeps sending new requests to the server, building up a backlog of pending requests. The server can therefore send responses back-to-back, and maximize its use of the path from the server to the client. Without concurrency, the server remains idle between the end of a response and the arrival of a new request, hence the path cannot be fully utilized.

- *Supporting "natural" concurrency*, in the sense that some applications do not need to follow the traditional request/response paradigm. In some cases, the endpoints are genuinely independent, and there is no natural concept of request/response.

Examples of protocols that attempt to keep the pipe full are the pipelining mode in HTTP, the streaming mode in NNTP, the Rsync protocol for file system synchronization, and the BitTorrent protocol for file-sharing. Examples of protocols/applications that support natural concurrency are instant messaging and Gnutella (in which the search messages are simply forwarded to other peers without any response message). Since BitTorrent supports client/server exchanges in both directions, and these exchanges are independent of each other, we can say that BitTorrent also supports a form of natural concurrency.

For data-concurrent connections, we use a different version of our a-b-t model in which the two directions of the connection are modeled independently by a pair (α, β) of connection vectors of the form

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

and

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

Depending on the nature of the concurrent connection, this model may or may not be a simplification. If the sides of the connection are truly independent, the model is accurate. Otherwise, if some dependency exists, it is not reflected in our characterization (*e.g.*, the fact that request a_i necessarily preceded response b_j is lost). Our current data acquisition techniques cannot distinguish these two cases, and we doubt that a technique to accurately distinguish them exists. In any case, the two independent vectors in our concurrent a-b-t model provide enough detail to capture the two uses of concurrent data exchange in a manner relevant for traffic generation. In the case of pipelined requests, one side of the connection mostly carries large ADUs with little or no quiet time between them (*i.e.*, backlogged responses). The exact timing at which the requests arrive in the opposite direction is irrelevant as long as there is always an ADU carrying a response to be sent. It is precisely the purpose of the concurrency to decouple the two directions to avoid the one round-trip time per request/response pair that sequential connections must incur in. There is, therefore, substantial independence in concurrent connections of this type, which supports the use of a model like the one we propose. In the case of connections that are "naturally" concurrent, the two sides are accurately described using two separate connection vectors.

3.3 Abstract Source-Level Measurement

The a-b-t model provides an intuitive way of describing source behavior in an application-neutral manner that is relevant for the performance of TCP. However, this would be of little use without a method for measuring real network traffic and casting TCP connections into the a-b-t model. We have developed an efficient algorithm that can convert an arbitrary trace of TCP/IP protocol headers into a set of connection vectors. The algorithm makes use of the wealth of information that segment headers provide to extract an accurate description of the abstract source-level behavior of the applications driving each TCP connection in the trace. It should be noted that this algorithm is a first solution to a complex inference problem in which we are trying to understand application behavior from the segment headers of a measured TCP connection without examining payloads, and hence without any knowledge of the identity of the application driving the connection. This implies "reversing" the effects of TCP and the network mechanisms that determine how ADUs are converted into the observed segments that carry the ADU. The presented algorithm is by no means the only one possible, or the most sophisticated one. However, we believe it is sufficiently accurate for our purpose, and we provide substantial experimental evidence in this and later chapters to support this claim.

3.3.1 From TCP Sequence Numbers to Application Data Units

The starting point of the algorithm is a trace of TCP segment headers, $\mathcal{T}_h$, measured on some network link. Our technique applies to TCP connections for which both directions are measured (known as a *bidirectional* trace), but we will also comment on the problem of extracting a-b-t connection vectors from a trace with only one measured direction (a *unidirectional* trace). While most public traces are bidirec-

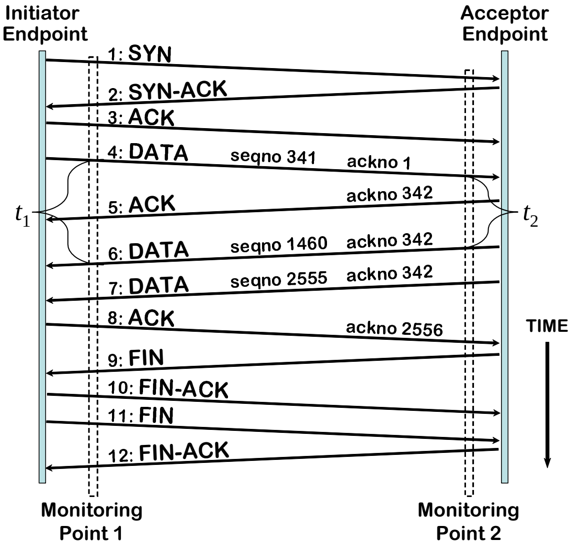


Figure 3.10: A first set of TCP segments for the connection vector in Figure 3.1: lossless example.

tional (*e.g.*, those in the NLANR repository [nlaa]), unidirectional traces are sometimes collected when resources (*e.g.*, disk space) are limited. Furthermore, routing asymmetries often result in connections that only traverse the measured link in one direction.

We will use Figure 3.10 to describe the basic technique for measuring ADU sizes and quiet time durations. The figure shows a set of TCP segments representing the exchange of data illustrated in the a-b-t diagram of Figure 3.1. After connection establishment (first three segments), a data segment is sent from the connection initiator to the connection acceptor. This data segment carries ADU a_1, and its size is given by the difference between the end sequence number and the beginning sequence number assigned to the data (bytes 1 to 341). In response to this data segment, the other endpoint first sends a pure acknowledgment segment (with cumulative acknowledgment number 342), followed by two data segments (with the same acknowledgment numbers). This change in the directionality of the data transmission makes it possible to establish a boundary between the first data unit a_1, which was transported using a single segment and had a size of 341 bytes, and the second data unit b_1, which was transported using two segments and had a size of 2,555 bytes.

The trace of TCP segments $\mathcal{T}_h$ must include a timestamp for each segment that reports the time at which the segment was observed at the monitoring device. Timestamps provide a way of estimating the duration of quiet times between ADUs. The duration of ta_1 is given by the difference between the timestamp of the 4th segment (the last and only segment of a_1), and the timestamp of the 6th segment (the first segment of b_1). The duration of tb_1 is given by the difference between the timestamp of the

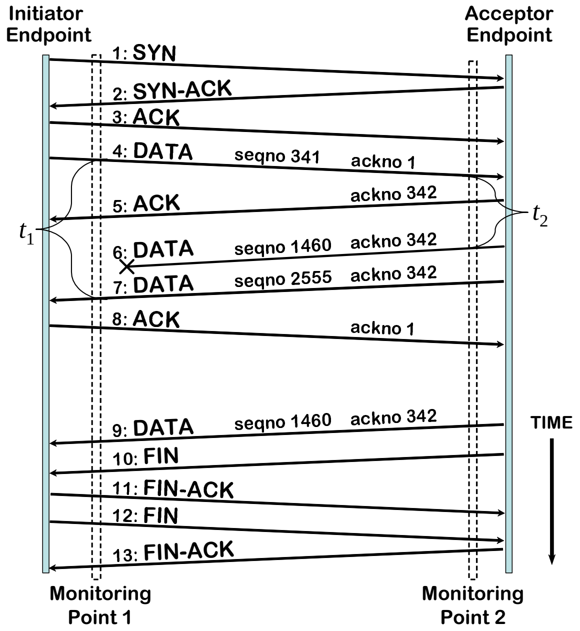


Figure 3.11: A second set of TCP segments for the connection vector in Figure 3.1: lossy example.

last data segment of b_1 (7th segment in the connection) and the timestamp of the first FIN segment (8th segment in the connection).

Note that the location of the monitoring point between the two endpoints affects the measured duration of ta_1 and tb_1 (but not the measured sizes of a_1 and b_1). Measuring the duration of ta_1 from the monitoring point 1 shown in Figure 3.10 results in an estimated time t_1 that is larger than the estimated time t_2 measured at monitoring point 2. Inferring application-layer quiet time durations is always complicated by this kind of measurement variability (among other causes), so short quiet times (with durations up to a few hundred milliseconds) should not be taken into account. Fortunately, the larger the quiet time duration, the less significant the measurement variability becomes, and the more important the effect of the quiet time is on the lifetime of the TCP connection. We can therefore choose to assign a value of zero to any measured quiet time whose duration is below some threshold, *e.g.*, 1 second, or simply use the measurement disregarding the minor impact of its inaccuracy.

If all connections were as "well-behaved" as the one illustrated in Figure 3.10, it would be trivial to create an algorithm to extract connection vectors from segment header traces. This could be done by simply examining the segments of each connection and counting the bytes sent between data directionality changes. In practice, segment reordering, loss, retransmission, duplication, and concurrency make the analysis much more complicated. Figure 3.11 shows a second set of segment exchanges that carry the same a-b-t connection vector of Figure 3.1. The first data segment of the ADU sent from the connection acceptor, the 6th segment, is lost somewhere in the network, forcing this endpoint to retransmit this segment some time later as the 9th segment. Depending on the location of the monitor (before or after the point of loss), the collected segment header trace may or may not include the 6th segment. If this segment is present in the trace (like in the trace collected at monitoring point 2), the analysis program must detect that the 9th segment is a retransmission and ignore it. This ensures we compute the correct size of b_1, *i.e.*, 2,555 bytes rather than 4,015 bytes. If the lost segment is not present in the trace (like in the trace collected at monitoring point 1), the analysis must detect the reordering of segments using their sequence numbers and still output a size for b_1 of 2,555 bytes. Measuring the duration of ta_1 is more difficult in this case, since the monitor never saw the 6th segment. The best estimation is the time t_1 between the segment with sequence number 341 and the segment with sequence number 2555. Note that if the 6th segment is seen (as for a trace collected at monitoring point 2), the best estimate is the time t_2 between 5th and 6th segments. A data acquisition algorithm capable of handling these two cases cannot simply rely on counts and data directionality changes, but must keep track of the start of the current ADU, the highest sequence number seen so far, and the timestamp of the last data segment. In our analysis, rather than trying to handle every possible case of loss and retransmission, we rely on a basic property of TCP to conveniently reorder segments and still obtain the same ADU sizes and inter-ADU quiet time durations. This makes our analysis simpler and more robust.

3.3.2 Logical Order of Data Segments

A fundamental invariant that underlies our previous ADU analyses is that every byte of application data in a TCP connection receives a sequence number, which is unique for its direction[7]. This property also means that data segments transmitted in the same direction can always be *logically ordered* by sequence number, and this order is independent of both the time at which segments are observed and any reordering present in the trace. The logical order of data segments is a very intuitive notion. If segments 6 and 7 in Figure 3.10 carried an HTML page, segment 6 carried the first 1,460 characters of this page, while segment 7 carried the remaining 1,095. Segment 6 logically preceded segment 7. When the same page is transmitted in Figure 3.11, the first half of the HTML is in segment 6 (which was lost) and again in segment 9. Both segments 6 and 9 (which were identical) logically precede segment 7, which carried the second half of the HTML page.

[7] This is true as long as the connection carries 4 GB or less. Otherwise, sequence numbers are repeated due to the wraparound of their 32-bit representation. We discuss how to address this difficulty at the end of Section 3.3.3.

The notion of logical order of data segments can also be applied to segments flowing in opposite directions of a sequential TCP connection. Each new data segment in a sequential connection must acknowledge the final sequence number of the last in-order ADU received in the opposite direction. If this is not the case, then the new data is not sent in response to the previous ADU, and the connection is not sequential (*i.e.*, two ADUs are being sent simultaneously in opposite directions). In the previous examples in Figures 3.10 and 3.11, we can see that both data segments comprising b_1 acknowledge the final sequence number of a_1. Intuitively, no data belonging to b_1 can be sent by the server until a_1 is completely received and processed. The data in a_1 logically precede the data in b_1, and therefore the segment carrying a_1 logically precedes the segments carrying b_1. Given the sequence and acknowledgment numbers of two data segments, flowing in the same or in opposite directions, we can always say whether the two segments carried the same data, or one of them logically preceded the other.

Connections that fit into the sequential a-b-t model are said to preserve a *total order of data segments* with respect to the logical flow of data:

> For any pair of data segments p and q, such that p is not a retransmission of q or *vice versa*, either the data in p logically precedes the data in q, or the data in q logically precedes the data in p.

In the example in Figure 3.11, the data in segment 9 logically precedes the data in segment 7 (*e.g.*, segment 9 carries the first 1460 bytes of a web page, and segment 7 carries the rest of the bytes). We know this because the sequence numbers of the bytes in segment 9 are below the sequence numbers of the bytes in segment 7. The first monitoring point observes segment 7 before segment 9, so temporal order of these two segments did not match their logical data order. A total order also exists between segments that flow in opposite directions. In the example in Figure 3.11, the data in segment 4 logically precede the data carried in the rest of the data segments in the connection. Timestamps and segment reordering play no role in the total order that exists in any sequential connection.

Logical data order is not present in data-concurrent connections, such as the one shown in Figure 3.8. For example, the segment that carried the last b-type ADU (the `438 don't send` ADU) may have been sent roughly at the same time as another segment carrying some of the new data of the data unit sent in the opposite direction (such as a `CHECK` ADU). Each segment would use new sequence numbers for its new data, and it would acknowledge the data received so far by the endpoint. Since the endpoints have not yet seen the segment sent from the opposite endpoint, the two segments cannot acknowledge each other. Therefore, there is no causality between the segments, and no segment can be said to precede the other. This observation provides a way of detecting data concurrency purely from the analysis of TCP segment headers. The idea is that a TCP connection that violates the total order of data segments described above can be said to be concurrent with certainty. This happens whenever a pair of data segments, sent in opposite directions, do not acknowledge each other, and therefore cannot be ordered according the logical data order.

Formally, a connection is considered to be concurrent when there exists at least one pair of data segments p and q that either flow in opposite directions and satisfy

$$p.seqno > q.ackno \tag{3.1}$$

and

$$q.seqno > p.ackno, \tag{3.2}$$

or that flow in the same direction and satisfy

$$p.seqno > q.seqno \tag{3.3}$$

and

$$q.ackno > p.ackno. \tag{3.4}$$

, Where $p.seqno$ and $q.seqno$ are the sequence numbers of p and q respectively, and $p.ackno$ and $q.ackno$ are the acknowledgment numbers of p and q respectively. Note that, for simplicity, our $.ackno$ refers to the cumulative sequence number accepted by the endpoint (which is one unit below the actual acknowledgment number stored in the TCP header [Pos81]). The first pair of inequalities defines the *bidirectional test* of data concurrency, while the second pair defines the *unidirectional test* of data concurrency. We next discuss why a connection satisfying one of these tests must necessarily be associated with concurrent data exchanging.

We consider first the case where p and q flow in opposite directions, assuming without loss of generality that p is sent from initiator to acceptor and q from acceptor to initiator. If they are part of a sequential connection, either p is sent after q reaches the initiator, in which case p acknowledges q so $q.seqno = p.ackno$, or q is sent after p reaches the acceptor in which case $p.seqno = q.ackno$. Otherwise, a pair of data segments that do not acknowledge each other exists, and the connection exhibits data concurrency.

In the case of segments p and q flowing in the same direction, we assume without loss of generality that

$p.seqno < q.seqno$. The only way in which $q.ackno$ can be less than $p.ackno$ is when p is a retransmission sent after q, and at least one data segment k with new data sent from the opposite direction arrives between the sending of p and the sending of q. The arrival of k increases the cumulative acknowledgment number in p with respect to q, which means that $q.ackno < p.ackno$. In addition, k cannot acknowledge p, or p would not be retransmitted. This implies that the connection is not sequential, since the opposite side sent new data in k *without* waiting for the new data in p.

Thus, only data-concurrent connections have a pair of segments that can simultaneously satisfy inequalities (3.1) and (3.2) or inequalities (3.3) and (3.4). These inequalities provide a formal test of data concurrency, which we will use to distinguish sequential and concurrent connections in our data acquisition algorithm. Data-concurrent connections exhibit a *partial order of data segments*, since segments flowing in the same direction can always be ordered according to sequence numbers, but not all pairs of segments flowing in opposite directions can be ordered in this manner.

Situations in which all of the segments in a concurrent data exchange are actually sent sequentially are not detected by the previous test. This can happen purely by chance, when applications send very little data or send it so slowly that concurrent data sent in the reverse direction is always acknowledged by each new data segment. Note also that the test detects *concurrent exchanges of data* and not concurrent exchanges of segments in which a data segment and an acknowledgment segment are sent concurrently. In the latter case, the logical order of data inside the connection is never broken because there is no data concurrency. Similarly, the simultaneous connection termination mechanism in TCP in which two FIN segments are sent concurrently is usually not associated with data concurrency. In the most common case, none of the FIN segments or only one of them carries data, so the data concurrency definition is not applicable. It is however possible to observe a simultaneous connection termination where both FIN segments carry data, which is considered concurrency if these segments satisfy inequalities (3.1) and (3.2).

3.3.3 Data Analysis Algorithm

We have developed an efficient data analysis algorithm that can determine whether a connection is sequential or concurrent, and can measure ADU sizes and quiet time durations in the presence of arbitrary reordering, duplication, and loss. Rather than trying to analyze every possible case of reordering, duplication/retransmission, and loss, we rely on the logical data order property, which does not depend on the original order and timestamps.

Given the set of segment headers of a TCP connection sorted by timestamp, the algorithm performs two passes:

1. Insert each data segment as a node into the data structure `ordered_segments`. This is a list of nodes that orders data segments according to the logical data order (bidirectional order for sequential connections, unidirectional order for concurrent connections). The insertion process serves also to detect data exchange concurrency. This is because connections are initially considered sequential, so their segments are ordered bidirectionally, until a segment that cannot be inserted according to this order is found. No backtracking is needed after this finding, since bidirectional order implies unidirectional order for both directions.
2. Traverse `ordered_segments` and output the a-b-t connection vector (sequential or concurrent) for the connection. This is straight-forward process, since segments in the data structure are already ordered appropriately.

The first step of the algorithm creates a doubly-linked list, `ordered_segments` in which each list node represents a data segment using the following four fields:

- $seqno_A$: the sequence number of the segment in the initiator to acceptor direction (that we will call the A direction). This sequence number is determined from the final sequence number of the segment (if the segment was measured in the "A" direction), or from the cumulative acknowledgment number (if measured in the "B" direction).
- $seqno_B$: the sequence number of the segment in the acceptor to initiator direction.
- *dir*: the direction in which the segment was sent (A or B).
- *ts*: the monitoring timestamp of the segment.

The list always preserves the following invariant that we call *unidirectional logical data order*: for any pair of segments p and q sent in the same direction D, the `ordered_segments` node of p precedes the `ordered_segments` node of q if and only if $p.seqno_D < q.seqno_D$. At the same time, if the connection is sequential, the data structure will preserve a second invariant that we call *bidirectional logical data order*, which is the opposite of the data concurrency conditions defined above: for any pair of segments p and q, the `ordered_segments` node of p precedes the `ordered_segments` node of q if and only if

$$(p.seqno_A < q.seqno_A) \wedge (p.seqno_B = q.seqno_B)$$

or

$$(p.seqno_A = q.seqno_A) \wedge (p.seqno_B < q.seqno_B).$$

Insertion of a node into the list starts backward from the tail of the `ordered_segments` looking for an insertion point that would satisfy the first invariant. If the connection is still being considered sequential, the insertion point must also satisfy the second invariant. This implies comparing the sequence numbers of the new segment with those of the last segment in the `ordered_segments`. The comparison can result in the following cases:

- The last segment of `ordered_segments` precedes the new one according to the bidirectional order above. If so, the new segment is inserted as the new last element of `ordered_segments`.
- The last segment of `ordered_segments` and the new segment have the same sequence numbers. In this case, the new segment is a retransmission and it is discarded.
- The new segment precedes the last segment of `ordered_segments` according to the bidirectional order. This implies that network reordering of TCP segments occurred, and that the new segment should be inserted before the last segment of `ordered_segments` to preserve the bidirectional order of the data structure. The new segment is then compared with the predecessors of the last segment in `ordered_segments` until its proper location is found, or inserted as the first segment if no predecessors are found.
- The last segment of `ordered_segments` and the new segment have different sequence numbers and do not show bidirectional order. This means that the connection is concurrent. The segment is then inserted according to its unidirectional order.

Since TCP segments can be received out of order by at most W bytes (the size of the maximum receiver window), the search pass (third bullet) never goes backward more than W segments. Therefore, the insertion step takes $O(s\ W)$ time, where s is the number of TCP data segments in the connection.

The second step is to walk through the linked list and produce an a-b-t connection vector. This can be accomplished in $O(s)$ time using `ordered_segments`. For concurrent connections, the analysis goes through the list keeping separate data for each direction of the connection. When a long enough quiet time is found (or the connection is closed), the algorithm outputs the size of the ADU. For sequential connections, the analysis looks for changes in directionality and outputs the amount of data in between the change as the size of the ADU. Sufficiently long quiet times also mark ADU boundaries, indicating an epoch without one of the ADUs.

Reordering makes the computation of quiet times more complex than it seems. As shown in Figure 3.11, if the monitor does not see the first instance of the retransmitted segment, the quiet times should be computed based on the segments with sequence numbers 341 and 2555. This implies adding two more fields to the list nodes:

- *min_ts*: the minimum timestamp of any segment whose position in the order is not lower than the one represented by this node. Due to reordering, one segment can precede another in the bidirectional order and at the same time have a greater timestamp. In this case, we can use the minimum timestamp as a better estimate of the send time of the lower segment.
- *max_ts*: the maximum timestamp of any segment whose place in the order is not greater than the one represented by this node. This is the opposite of the previous *min_ts* field, providing a better estimate of the receive time of the greater segment.

These fields can be computed during the insertion step without any extra comparison of segments. The best possible estimate of the quiet time between two ADU becomes

$$q.min\_ts - p.max\_ts$$

for p being the last segment (in the logical data order) of the first ADU, and q being the first segment (in the logical data order) of the second ADU. For the example in Figure 3.11, at monitoring point 1, the value of *min_ts* for the node for the 9th segment (that marks a data directionality boundary when segment nodes are sorted according to the logical data order) is the timestamp of the 7th segment. Therefore, the quiet time ta_1 is estimated as t_1. Note that the use of more than one timestamp makes it possible to handle IP fragmentation elegantly. Fragments have different timestamps, so a single timestamp would have to be arbitrarily set to the timestamp of one of the fragments. With our algorithm, the first fragment provides sequence numbers and usually *min_ts*, while the last fragment usually provides *max_ts*.

Other Issues in Trace Processing

Our trace processing algorithm makes two assumptions. First, it assumes we can isolate the segments of individual connections. Second, it assumes that no wraparound of sequence numbers occurs (otherwise, logical data order would not be preserved). These two assumptions can be satisfied by preprocessing the trace of segment headers. Isolating the segments of individual TCP connections was accomplished by sorting packet header traces on five keys: source IP address, source port number, destination IP address, destination port number, and timestamp. The first four keys can separate segments from different TCP connections as long as no source port number is reused. When a client establishes more than one connection to the same server (and service), these connections share IP addresses and destination port numbers, but not source port numbers. This is true unless the client is using so many connections that it reuses a previous source port number at some point. Finding such source port number reuses is relatively common in our long traces, which are at least one hour long. Since segment traces are sorted by timestamp, it is possible to look for pure SYN segments and use them to separate TCP connections that reuse source port numbers. However, SYN segments can suffer from retransmissions, just like any other segment, so the processing must keep track of the sequence number of the last SYN segment observed. Depending on the operating system of the connection initiator, this sequence number is either incremented or randomly set for each new connection. In either case, the probability of two connections sharing SYN sequence numbers is practically zero.

Segment sorting according to the previous 5 keys requires $O(s\ log\ s)$ time (we use the Unix `sort` utility for our work). It is also possible to process the data without an initial sorting step by keeping state in memory for each active connection. On the one hand, this can potentially eliminate the costly $O(s\ log\ s)$ step, making the entire processing run in linear time. On the other hand, it complicates the implementation, and increases the memory requirements substantially[8]. Detecting the existence of distinct connections with identical source and destination IP addresses and port numbers requires $O(s)$ time, simply by keeping track of SYN sequence numbers as discussed above. In our implementation, this detection is done at the same time as segments are inserted into `ordered_segments` data structure, saving one pass.

TCP sequence numbers are 32-bit integers, and the initial sequence number of a TCP connection can take any value between 0 and $2^{32}-1$. This means that wraparounds are possible, and relatively frequent. One way to handle sequence number wraparound is by keeping track of the initial sequence number and performing a modular subtraction. However, if the SYN segment of a connection is not observed (and therefore the initial sequence number is unknown), using modular arithmetic will fail whenever the

[8] The well-known `tcptrace` tool [Ost], provides a good example of the difficulty of efficiently implementing this technique. `tcptrace` can analyze multiple connections at the same time, by keeping separate state for each connection, and making use of hashing to quickly locate the state corresponding to the connection to which a new segment belongs. When this tool is used with our traces, we quickly run out of memory on our processing machines (which have 1.5 GB of RAM). This occurs even when we use `tcptrace`'s real-time processing mode, which is supposed to be highly optimized. We believe it is possible to perform our analysis without the sorting step, but it is certainly much more difficult to develop a memory-efficient implementation.

connection suffers from reordering of the first observed segments. In this case the subtraction would start in the wrong place, *i.e.*, from the sequence number of the first segment seen, which is not the lowest sequence number due to the reordering. One solution is to use backtracking, which complicates the processing of traces.

A related problem is that representing sequence numbers as 32-bit integers is not sufficient for connections that carry more than 2^{32} bytes of data (4 GB). The simplest way to address this measurement problem is to encode sequence numbers using more than 32 bits in the `ordered_segments` data structure. In our implementation we use 64 bits to represent sequence numbers, and rely on the following algorithm[9] to accurately convert 32 bit sequence numbers to 64-bit integers even in the presence of wraparounds. The algorithm makes use of a wraparound counter and a pair of flags for each direction of the connection. The obvious idea is to increment the counter each time a transition from a high sequence number to a low sequence number is seen. However, due to reordering, the counter could be incorrectly incremented more than once. For example, we could observe four segments with sequence numbers $2^{32} - 1000, 1000, 2^{32} - 500$, and 2000. Wraparound processing should convert them into $2^{32} - 1000, 2^{32} + 1000, 2^{32} - 500$, and $2^{32} + 2000$. However, if the wraparound counter is incremented every time a transition from a high sequence number to a low sequence number is seen, the counter would be incremented once for the segment with the sequence number 1000 and again for the segment with sequence number 2000. In this case, the wraparound processing would result in four segments with sequence numbers $2^{32} - 1000, 2^{32} + 1000, 2^{32} - 500$, and $2^{32} + 2^{32} + 2000$. The second increment of the counter would be incorrect.

The solution is to use a flag that is set after a “low” sequence number is seen, so the counter is incremented only once after each “crossing” of 2^{32}. This opens up the question of when to unset this flag so that the next true crossing increments the counter. This can be solved by keeping track of the crossing of the middle sequence number. In our implementation, we use two flags, `low_seqno` and `high_seqno`, which are set independently. If the next segment has a sequence number in the first quarter of 2^{32} (*i.e.*, in the range between 0 and $2^{30} - 1$), the flag `low_seqno` is set to true. If the next segment has a sequence number in the fourth quarter of 2^{32} (*i.e.*, in the range between 2^{31} and $2^{32} - 1$), the other flag`high_seqno` is set to true. These flags are unset, and the counter incremented, when both flags are true and the next segment is not in the first or the fourth quarter of 2^{32}. Sequence numbers in the first quarter are incremented to 2^{32} times the counter plus 1. The rest are incremented by 2^{32} plus the

[9] We have not addressed the extra complexity that TCP window scaling for Long-Fat-Networks (RFC 1323 [JBB92]) introduces. It is often the case that TCP options are not available in the traces, so the use of window scaling and TCP timestamps has to be inferred from the standard TCP header. This is a daunting task. If the options are available, it is straightforward to combine regular sequence numbers and timestamps to handle this case.

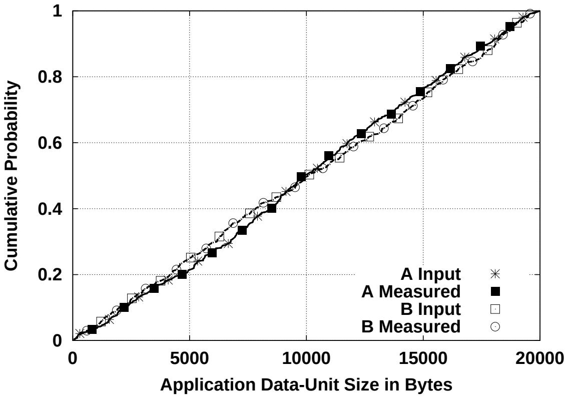
counter. This handles the pathological reordering case in which the sequence number of the first segment in a connection is very close to zero, and the next segment is very close to 2^{32}. In this case the low sequence number would be incremented by 2^{32}. This algorithm requires no backtracking, and runs in $O(s)$ time. In our implementation, the sequence number conversion algorithm has been integrated into the same pass as the insertion step of the ADU analysis.

Our data acquisition techniques have been implemented in the analysis program `tcp2cvec`. The program also handles a number of other difficulties that arise when processing real traces, such as TCP implementations that behave in non-standard ways. In addition, it also implements the analysis of network-level parameters described in the next chapter.

3.4 Validation using Synthetic Applications

The data analysis techniques described in the previous section are based on a number of properties of TCP that are expected to hold for the vast majority of connections recorded. For example, the logical data order property should always hold, since TCP would fail to deliver data to applications otherwise. There are, however, a number of possible sources of uncertainty in the accuracy of the data acquisition method, and this section studies them using testbed experiments.

The concept of an ADU provides a useful abstraction for describing the demands of applications for sending and receiving data using a TCP connection. However, the ADU concept is not really part of the interface between applications and TCP. In practice, each TCP connection results from the use of a programming abstraction, called a socket, that receives requests from the applications to send and receive data. These requests are made using a pair of socket system calls, `send()` (application's write) and `recv()` (application's read). These calls pass a pointer to a memory buffer where the operating system can read the data to be sent or write the data received. The size of the buffer is not fixed, so applications are free to decide how much data to send or receive with each call and can even use different sizes for different calls. As a result, applications may use more than one send system call per ADU, and there may be significant delays between successive calls belonging to the same ADU. These operations can further interact with mechanisms in the lower layers (*e.g.*, delayed acknowledgment, TCP windowing, IP buffering, etc.) creating even longer delays between segments carrying ADUs. Such delays distort the relationship between application-layer quiet times and segment dynamics, complicating the detection of ADU boundaries due to quiet times.



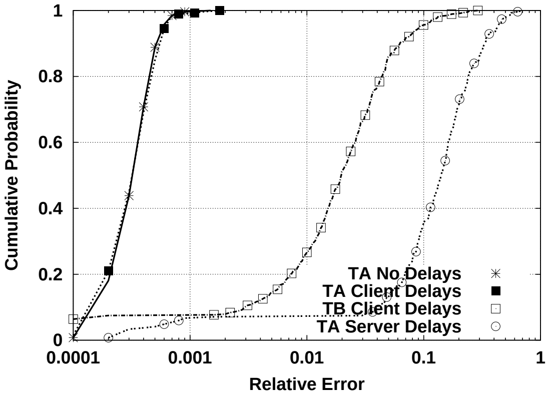


Figure 3.12: Distributions of ADU sizes for the testbed experiments with synthetic applications.

Figure 3.13: Distributions of quiet time durations for the testbed experiments with synthetic applications.

To test the accuracy of our data acquisition techniques, we constructed a suite of test applications that exercise TCP in a systematic manner. The basic logic of each test application is to establish a TCP connection and send a sequence of ADUs with a random size, and with random delays between each pair of ADUs. In the a-b-t model notation, this means creating connections with random a_i, b_i, ta_i and tb_i. As the test application runs, it logs ADU sizes and various time intervals as measured by the application. In addition, the test application can set the socket send and receive calls to random I/O sizes, and can introduce random delays between successive send or receive calls within a single ADU. In our experiments, the test application was run between two real hosts, and traces of the segment headers were collected and analyzed using our measurement tool. Our validation compared the result of this analysis and the correct values logged by the applications.

We conducted an extensive suite of tests, but limit our report to only some of the results. Specifically we only show the results with the most significant deviations from the correct values for ADU sizes or quiet time durations. Figure 3.12 shows the relative error, defined as

$$\frac{\text{value} - \text{approximation}}{\text{value}}$$

, in measuring the randomly generated ADU sizes when random send/receive sizes and random delays between socket operations were used in the test applications. The distribution of sizes of a-type ADUs as logged by the application is labeled “A Input”, while the distribution of sizes of a-type ADU measured from segment headers is labeled “A Measured”. There is virtually no difference between the correct and inferred values. Figure 3.12 also shows the same data for the b-type distributions which appear equally accurate. This means that our analysis will correctly infer ADU sizes even though send/receive sizes and socket operation delays are variable.

In general, we found only two cases that expose limitations in the data acquisition method when analyzing sequential connections. While random application-level send and receive sizes, and random delays between successive send operations within a data unit do not have a significant effect, random delays between successive receive operations produce errors in estimating some quiet time durations. In this case, the application inflates the duration of a quiet time by not reading data that may already be buffered at the receiving endpoint. The consequence is a difference between the quiet time as observed at the application level and the quiet time observed at the segment level. The quiet time observed by the application is the time between the last read used to receive the ADU a_i (or b_i) and the first write used to send the next ADU b_i (a_{i+1}). The quiet time observed at the segment level is the time between the arrival of the last segment of a_i (b_i) and the departure of the first segment of b_i (a_{i+1}). If the application reads the first ADU slowly, using read calls with significant delays between them, it will finish reading a_i (b_i) well after the last segment has reached the endpoint. In this case, the quiet time appears significantly shorter at the application level than at the segment level.

For example, a data unit of 1,000 bytes may reach the receiving endpoint in a single segment and be stored in the corresponding TCP window buffer. The receiving application at this endpoint could read the ADU using 10 `recv()` system calls with a size of only 100 bytes, and with delays between them of 100 milliseconds. The reading of this ADU would therefore take 900 milliseconds, and hence the application would start measuring the subsequent quiet time 900 milliseconds after the arrival of the data segment. Our measurement of quiet time from segment arrivals can never see this delay in application reads, and would therefore add 900 milliseconds to the quiet time. For most applications we claim there is no good reason to delay read operation more than a few milliseconds. Therefore, the inaccuracy demonstrated here should be very infrequent. Nonetheless we have no direct means of assessing this type of error in our traces.

Figure 3.13 shows the relative error in the measurement of quiet time duration when there are random delays between successive read operations. The worst error is found when measuring quiet times between a_i and b_i (*i.e.*, within an epoch) when random read delays occur on the connection acceptor (receiver of a_i and b_i). Even in this case, 70% of values have less than 20% error in an experiment with what we considered severe conditions of delays between read operations for a single ADU (random delays between 10 and 100 milliseconds).

We also studied the impact of segment losses on the accuracy of the measurements. In general,

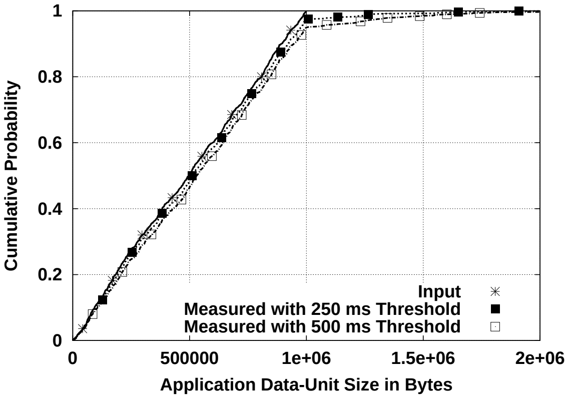


Figure 3.14: Distributions of ADU sizes for the testbed experiments with synthetic applications.

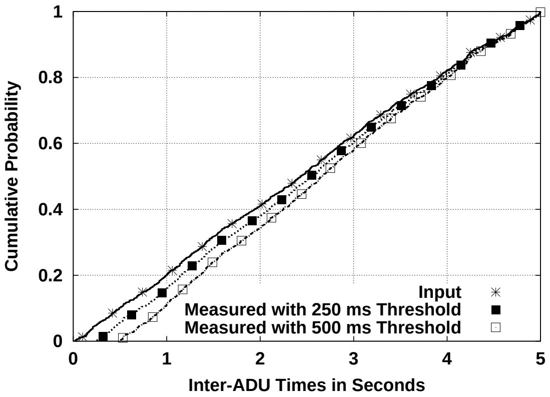


Figure 3.15: Distributions of quiet time durations for the testbed experiments with synthetic applications.

the algorithm performs well, but the analysis helped us to identify one troublesome case. If the last segment of an ADU is lost, the receiver side does not acknowledge the last sequence number of the ADU. After a few hundred milliseconds the sender side times out and resends the last segment. If the loss of the segment occurs before the monitoring point, no retransmission is observed for this last segment. If the time between this last segment and its predecessor is long enough (due to the TCP timeout), the ADU is incorrectly divided into two ADUs. Other types of segment loss do not have an effect on the measurement, since the algorithm can use the observation of retransmission and/or reordering to identify quiet times not caused by source-level behavior. The troublesome case is so infrequent that we did not try to address it. However, we note that it seems possible to develop a heuristic to detect this type of problem. The idea would be to estimate the duration of the TCP retransmission timeout, and ignore gaps between segments that are close to this estimate. The implementation of this heuristic would be complicated by the need to take into account differences in the resolution of the TCP retransmission timers, round-trip time variability and the possibility of consecutive losses.

Measuring the size of ADUs in concurrent connections is generally more difficult. This is because a change in the directionality of sequence number increases does not constitute an ADU boundary and thus we have to rely instead on quiet times to split data into ADUs. Figure 3.14 compares the input distribution of ADU sizes (from both a-type and b-type ADUs) and the measured sizes when the sizes of socket reads/writes and the delays between them are random. The measurement is generally very accurate, although some ADUs that were sent with small quiet times between them are mistakenly joined into the same measured ADU. This creates a longer tail in the measured distributions. Reducing the quiet time threshold from 500 to 250 milliseconds does little to reduce the measurement inaccuracy.

The measured quiet times are also quite close to those at the application level, as shown in Figure 3.15. The small inaccuracy comes again from ADUs that are joined together when their inter-ADU times are short. This inaccuracy biases the measured distribution of quiet times against small values (notice that the measured distributions start at a higher value). Reducing the minimum quiet time threshold to 250 milliseconds makes the measured distribution closer to the actual distribution.

3.5 Analysis Results

The a-b-t model provides a novel way of describing the workload that applications create on TCP connections. Thanks to the efficiency of the analysis method presented in Section 3.3, we are able to process large packet header traces from several Internet links. This section presents our results. The analysis of the a-b-t connection vectors extracted from disparate traces reveals that certain distributional properties remain surprisingly homogeneous across links and times-of-day, while others change substantially. To the best of our knowledge, this is the first characterization of the behavior of sources driving TCP connections that considers the entire mix of application traffic rather than just one or a few applications.

Our results come from the five traces shown in Table 3.1. This table reports statistics that compare the number of connections that are determined to be sequential and those that are determined to be concurrent according to the analysis algorithm described in section 3.3. The main lesson from Table 3.1 is the very different view of aggregate source-level behavior that counting connections or counting bytes provide. In terms of the number of connections, concurrent connections appear insignificant, accounting for a mere 3.6% of the connections in the Leipzig-II trace. The picture is completely different, however, when we consider the total number of bytes carried in those concurrent connections. In this case, concurrent connections account for 21.7% of the Leipzig-II workload, clearly suggesting that concurrency is frequently associated with TCP connections that carry large amounts of data. Abilene-I provides an

	Sequential Connections				**Concurrent Connections**			
Trace	**Count**	%	**GB**	%	**Count**	%	**GB**	%
Abilene-I	2,335,428	98.4	400.36	68.1	39,260	1.7	187.95	31.9
Leipzig-II	1,836,553	96.4	46.08	78.3	68,857	3.6	12.77	21.7
UNC 1 AM	529,381	98.5	90.35	82.4	8,345	1.6	19.34	17.6
UNC 1 PM	2,124,431	99.1	189.75	87.9	18,855	0.9	26.11	12.1
UNC 7:30 PM	808,857	98.7	102.04	76.8	10,542	1.3	30.83	23.2

Table 3.1: Breakdown of the TCP connections found in five traces.

even more striking illustration, where 31.9% of the bytes were carried by concurrent connections, which only accounted for 1.7% of the total number of connections in the trace. This is not surprising given that one of the motivations for the use of data exchange concurrency is to increase throughput. Applications with a substantial amount of data to send can greatly benefit from higher throughput, and this justifies the increase in complexity that implementing concurrency requires. On the contrary, applications which generally transfer small amounts of data have less incentive to complicate their application protocols in order to support concurrency. In this fashion, interactive traffic (*e.g.*, telnet, SSH, IRC), which tends to be associated with large numbers of small ADUs, does not usually profit from concurrency.

It is important to note that two types of TCP connections are not included in the statistics in Table 3.1: unidirectional connections and connections that carried no application data (*i.e.*, no segment carried a payload). Unidirectional connections are those for which the trace contains only segments flowing in one direction (either data or ACK segments). There are two major causes for these types of connections[10]. First, attempts to contact a nonexistent or unavailable host may not receive any response segments. In this case, the trace would show only one or a few SYN segments flowing in one direction, and no communication of application data between the two hosts. Attempts to connect to firewalled hosts also result in similar unidirectional connections. Second, routing asymmetries, that are known to be frequent in the Internet backbone, may result in connections that traverse the measured link only in one direction. Among our traces, routing asymmetries are only possible for the Abilene-I trace. The UNC and Leipzig-II traces were collected from border links that carry all of the network traffic to and from these two institutions. Two other possible causes of unidirectionality, that we believe have a much smaller impact on the count of unidirectional connections, are the effects of trace boundaries, which can limit the tracing to only a few segments flowing in one direction; and misconfigurations, where incorrect or spoofed source addresses are used.

In the UNC and Leipzig-II traces, the number of unidirectional connections was relatively high. We found between 249,923 (Leipzig-II) and 1,963,511 (UNC 1 AM) unidirectional connections. Since these are traces without any routing asymmetry, it is clear that a substantial number of attempts to establish a TCP connection failed. For example, the UNC 1 AM trace has approximately one million more unidirectional connections than the other two UNC traces. These connections are likely related to

[10] It is very unlikely that any of these connections was measured as unidirectional due to measurement losses. The traces studied in this section were collected using a high-performance monitoring device, a DAG card [Pro], that did not report any losses during data acquisition.

some traffic anomaly, such as malicious network scanning[11] and port scanning[12]. We have not studied this phenomenon further, but it is clearly important to filter out unidirectional connections to produce the results in Table 3.1. Otherwise, the percentages would be misleading, since this table is about connections that exchanged one or more ADUs during TCP application communication, and unidirectional connections did not engage in any kind of useful communication. Furthermore, unidirectional connections accounted for less than 0.15% of the bytes in the Leipzig-II and UNC traces.

The number of unidirectional connections in the Abilene-I trace was even larger: 2.6 millions in the Indianapolis to Cleveland direction and 22.3 millions in the opposite direction. Unlike the UNC and Leipzig-II traces, these connections accounted for a significant fraction of the bytes in each direction (1.63% and 14.42%). This fact, and a closer examination of the connections[13], confirmed that routing asymmetry is present in the Abilene-I trace. Asymmetric connections can carry application data, and therefore should be considered in source-level studies. However, our concurrency test requires bidirectional measurements, so the type of breakdown shown in Table 3.1 cannot be performed with the unidirectional connections in the Abilene-I trace.

Our traces also include a significant number of connections that did not carry any application data (*i.e.*, TCP connections that were established and terminated without transmitting a single data segment[14]). The number of connections without any data units varied between 75,522 in the UNC 1 AM trace and 400,853 in the Abilene-I trace. These "dataless" connections can again be due to network and port scanning, and also to failed attempts to establish TCP connections. These failures can come from attempts to contact endpoint port numbers on which no application is listening[15]. They can also come from aborted connections which are due to high loss rates, excessive round-trip times, or implementation problems. While the number of connections without application data is relatively high when compared with the number of connections in Table 3.1, these connections accounted for less than 0.11% of the bytes.

[11] Network scanning is a technique for discovering the hosts attached to a network by probing each possible IP address in a network domain. The basic technique is to send a packet which generally requires a response from the host that received it (*e.g.*, an ICMP echo request, a TCP SYN segment). Malicious users often scan remote networks to find hosts before trying to break into them. Network scanning with TCP segments is available in many popular tools, *e.g.*, `nmap`.

[12] Port scanning is similar to network scanning, but it involves probing a range of port numbers (for a single IP address) rather than probing a range of IP addresses. The goal of port scanning is to discover active services, which could potentially have vulnerabilities. Port scanning is performed using any TCP segment (or UDP datagram) that elicits a response from the victim (*e.g.*, a SYN segment requires a SYN-ACK in response, a malformed segment requires a RST segment in response).

[13] We found numerous connections that had data segments with increasing sequence numbers.

[14] In some cases, these connections showed some data segments with a sequence number above that of the FIN segments. These cases seemed to be caused by TCP implementation errors.

[15] In this case the destination endpoint responds with a TCP reset segment, and no application-level communication takes place.

The rest of this section examines the distributional properties of the connection vectors derived from the traces. Connection vectors constitute a rich data set that can be explored along different axes. We have chosen to first compare traces collected at different sites. This helps us study variability in source-level behavior originating from differences in the populations of users and services. The second part of the section studies the three traces from UNC, analyzing the changes in source-level behavior due to the strong time-of-day effects that most Internet links exhibit. At the same time, this section illustrates the significant difference between TCP connections initiated from one side of the link (by clients inside UNC) and those initiated from the other side (by clients outside UNC that contacted servers inside UNC).

Note that the analysis below reports only on those connection vectors derived from TCP connections that were *fully captured*, *i.e.*, those for which we believe that every segment was observed. In practice, we consider that a connection was fully captured when we observe both the start of the connection, marked by SYN and SYN-ACK segments, and the end of the connection, marked by FIN or RST segments. This does not necessarily mean that we observed every single segment of the connection[16], but it does imply that the full source-level behavior of the connection is observed. Another reason to work only with fully captured connections is that the absence of connection establishment segments prevents us from identifying the connection initiator. It is often the case that the acceptor is listening on a reserved port number (< 1024), which provides a way to address this difficulty. However, there is still a large fraction of the connections that use dynamic port numbers, and for which the initiator cannot be identified with certainty.

3.5.1 Variability Across Sites

Sequential Connections

We start our statistical analysis with the characterization of sequential connections from different sites. Figure 3.16 examines the distributions of the sizes of the ADUs for three traces: Abilene-I, Leipzig-II and UNC 1 PM. We use the letter "*A*" to refer to a distribution of a-type ADU sizes, and the letter "*B*" to refer to a distribution of b-type ADU sizes. The distributions in this figure only include samples from sequential connection vectors. We can distinguish two regions in this plot. For sizes of ADUs above 250 bytes, the shape of the *A* distributions is remarkably similar for all three traces, and

[16] In some (rare) cases, we may miss some segments before connection establishment (*e.g.*, we miss the first SYN segment but observe its retransmission), or we may miss some segments after connection establishment (*e.g.*, we miss the retransmission of the final FIN segment and its acknowledgment).

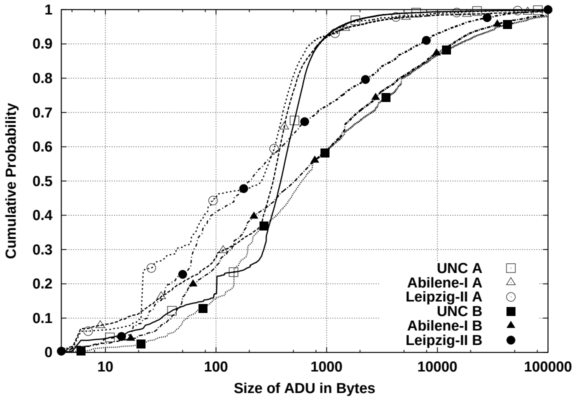


Figure 3.16: Bodies of the A and B distributions for Abilene-I, Leipzig-II and UNC 1 PM.

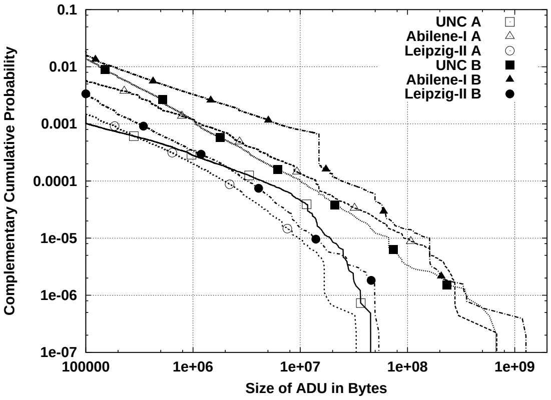


Figure 3.17: Tails of the A and B distributions for Abilene-I, Leipzig-II and UNC 1 PM.

quite different from the shapes of the B distributions. The vast majority of the ADUs sent from the connection initiator (92%) had a size below 1,000 bytes. This is consistent with the idea that a-type ADUs mostly carry small requests and control messages. Most a-type ADUs can therefore be carried in a single standard-size segment of 1960 bytes. The shape of the B distributions is also consistent with our intuition, although the Leipzig-II distribution is significantly lighter than the others. The B distributions are heavier than the A distributions. Between 38% and 27% of the b-type ADUs are larger than 1460 bytes, so they require two or more segments to be transported from the connection acceptor to the connection initiator. Only 8% to 12% of the b-type ADUs carried 10,000 bytes or more. We also note that for ADU sizes below 250 bytes, the plot shows less similarity among distributions of the same type. However, the logarithmic scale on the x-axis can be misleading. The large separation between the curves corresponds to only a few tens of bytes, and this has little impact on TCP performance. ADUs as small as 250 bytes can always be transported in a single (small) segment.

Figure 3.17 shows the tails of the A and B distributions using complementary cumulative distribution functions. It shows that even a-type ADUs can be quite large, and that the distributions are consistent with heavy-tailness (*i.e.*, exhibits linear decay in the log-log CCDF). For this reason, Pareto or Lognormal models could provide a good foundation for analytical modeling of the distributions[17]. Interestingly, when we compare A and B distributions for the same trace, we find that B distributions are only slightly heavier than A distributions, especially for Abilene-I and Leipzig-II. This implies that there are protocols in which the initiator sends large ADUs to the acceptor. For example, web browsers are often used to upload files and email attachments for web-based email accounts. It is also interesting to

[17] The tail of a Pareto distribution is always linear in a CCDF, and the tail of a Lognormal distribution can be linear for an arbitrary number of orders of magnitude.

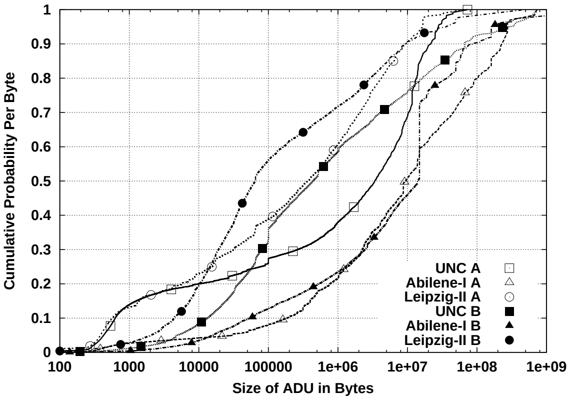


Figure 3.18: Bodies of the A and B distributions with per-byte probabilities for Abilene-I, Leipzig-II and UNC 1 PM

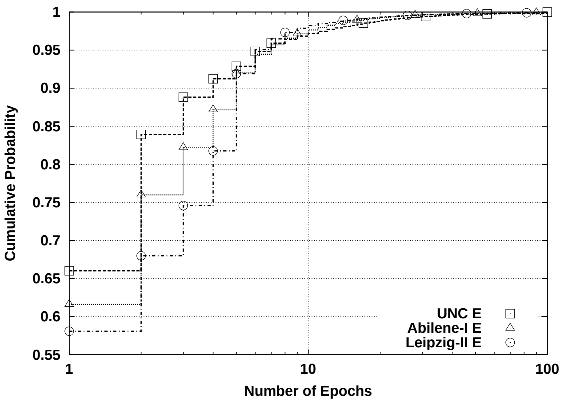


Figure 3.19: Bodies of the E distributions for Abilene-I, Leipzig-II and UNC 1 PM.

note that Abilene-I's A distribution is heavier than UNC's and Leipzig-II's B distributions, and that UNC's B distribution is significantly heavier than Leipzig-II's B distribution. We believe this reflects the type of network measured and/or the population of users. Transferring large ADUs is more feasible in higher capacity networks, and this fosters the use of more data-intensive applications and more data-intensive uses of applications. Abilene is a well-provisioned backbone network that carries traffic between well-connected American universities, so it seems more likely to exhibit connections with larger ADUs.

The small probabilities of finding large ADUs shown in Figures 3.16 and 3.17 can give the false impression that only small ADUs are important. Figure 3.18 corrects this view by plotting the probability that a byte is carried in an ADU of a given size. The figure shows that the majority of the bytes in the network were carried in large ADUs. For example, the probability that a byte was carried in an ADU of 100,000 bytes or more was as high as 0.9 for Abilene-I. This is in stark contrast to the corresponding Abilene-I distribution in Figure 3.16, where the probability of an ADU of 100,000 bytes or more is as low as 0.01 for the three traces.

The three networks show remarkably different distributions in Figure 3.18. This is in part due to the impact of sampling on this type of analysis, which is rather sensitive to the number of samples in the tail of the distribution. Adding a single very large sample can shift the entire distribution downward, since the probability of finding a byte in the rest of the ADU sizes decreases significantly. However, we can still make interesting observations about the bodies of these distributions based on their shapes (which are not affected by sampling artifacts). The distributions for UNC and Leipzig-II show two striking crossover points, the first one around 10 KB and the second one around 10 MB. The curves before the first crossover point show that the ADUs carrying 20% of the a-type bytes tended to be much smaller

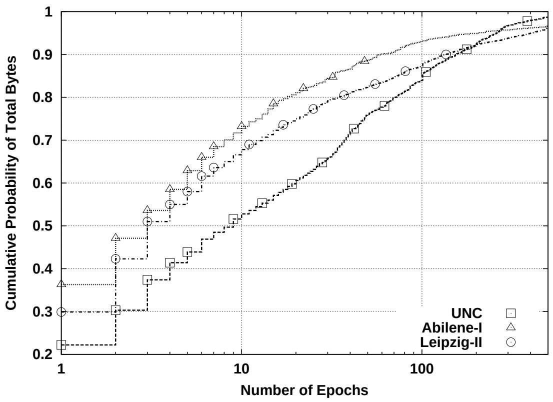


Figure 3.20: Bodies of the E distributions with per-byte probabilities for Abilene-I, Leipzig-II and UNC 1 PM.

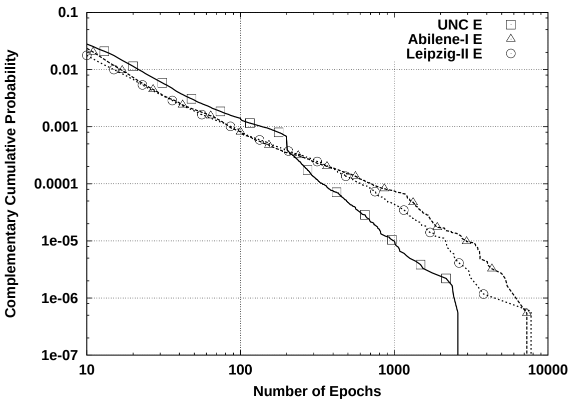


Figure 3.21: Tails of the E distributions for Abilene-I, Leipzig-II and UNC 1 PM.

than those carrying 20% of the b-type bytes. The curves between the two crossover points show the opposite for larger ADUs. Here 50% of the a-type bytes are carried in ADUs that tended to be much larger than those ADUs carrying b-type bytes. The situation reverses again after the second crossover point. This shows that the A distributions are strongly bimodal: objects are either much smaller or much larger than the average b-type ADU. The same phenomenon is found in the Abilene-I distributions between 10 KB and 1 MB, but the difference in probability is much smaller here (and could be explained by tail sampling artifacts). In addition, there is a third crossover point in the Abilene-I distributions, which defines a new region between 15 and 250 MB.

The distribution of the number of epochs E in each set of connection vectors is shown in Figure 3.19. Between 58% and 66% of the connection vectors have a single epoch. This includes a significant number of connections with a single half-epoch that come from FTP-DATA connections. Only 5% of the connections have more than 10 epochs. This does not mean that connections with a large number of epochs are unimportant. As Figure 3.20 shows, connections with a large number of epochs are responsible for a large fraction of the bytes. For example, connections with 10 epochs or more, which represent 3% of the connections, carried between 30% and 50% of the total bytes, depending on the trace.

Figure 3.20 shows that UNC's E distribution is substantially heavier than the ones for the other two traces when probability is computed over the total number of bytes. This suggests that the type of traffic in the UNC trace includes applications that make more use of multi-epoch connections. This also provides evidence that connections with moderate numbers of epochs can fit within the shorter duration (1 hour) of this trace. Otherwise, the Abilene-I trace (2 hours long) and the Leipzig-II traces(2 hours and 45 minutes long) would show heavier bodies. On the contrary, the tails of the E distributions shown

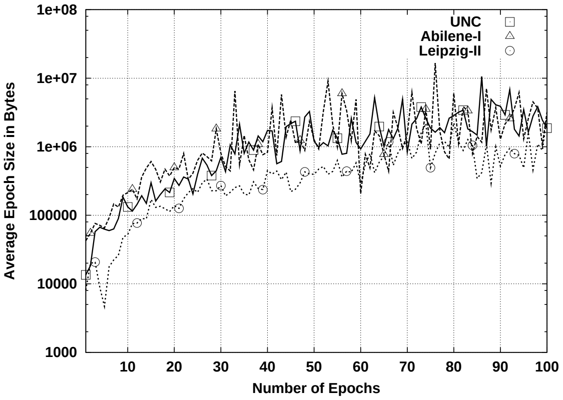


Figure 3.22: Average size $a_j + b_j$ of the epochs in each connection vector as a function of the number of epochs, for UNC 1 PM, Abilene-I and Leipzig-II

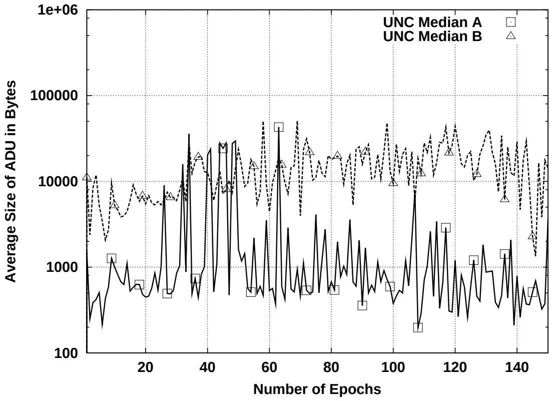


Figure 3.23: Average of the median size of the ADUs in each connection vector as a function of the number of epochs, for UNC 1 PM.

in Figure 3.21 are significantly heavier for Abilene-I and Leipzig-II than for UNC. This perhaps suggests that 1-hour traces are too short to observe connections with thousands of epochs. The sharp change in the slope of the tail of UNC's E distribution could be explained by a common application that has a fixed limit on the number of epochs (perhaps 110). However, we know of no such application.

One interesting modeling question is whether there is any dependency between the size of the ADU in one epoch and the number of epochs in the connection. If these are independent, it would be straightforward to generate synthetic connection vectors simply by first sampling a number of epochs E and then assigning ADU sizes by sampling from A and B. Figure 3.22 shows that this independence does not exist. The average size of an epoch (*i.e.*, $a_j + b_j$) increases very quickly for connections up to 30 epochs (notice the logarithmic y-axis). Connections with more epochs show high variability in the average size of their epochs. UNC and Abilene-I have quite similar averages that are much larger than those found in Leipzig-II (but note the sharp increase in average sizes for connections with 60 to 80 epochs).

Figures 3.24-3.26 provide further evidence against the independence of ADU sizes and number of epochs, and illustrate some remarkable complexity and site dependence. The plots illustrate how the number of epochs changes the size of the typical ADU, where ”typical” is defined as the median of the sizes of the ADUs in each connection vector. Since a large number of connection vectors have the same number of epochs, we summarized these data by plotting the average of the median sizes *vs.* the number of epochs. Unlike the data in Figure 3.22, we analyzed median ADU sizes for a-type and b-type ADUs separately.

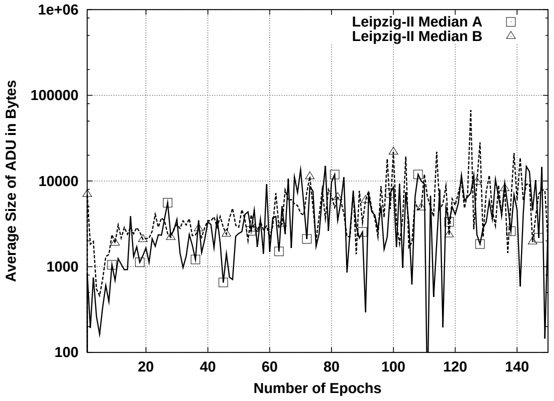


Figure 3.24: Average of the median size of the ADUs in each connection vector as a function of the number of epochs, for Leipzig-II.

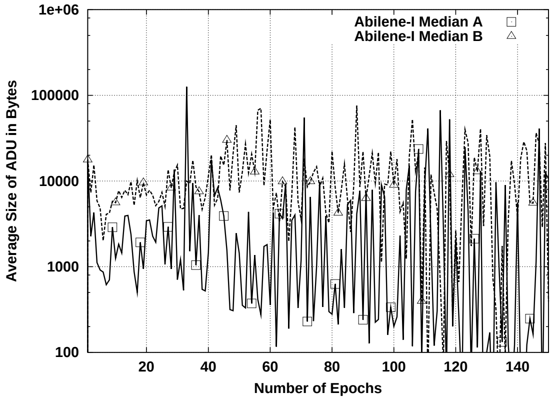


Figure 3.25: Average of the median size of the ADUs in each connection vector as a function of the number of epochs for Abilene-I.

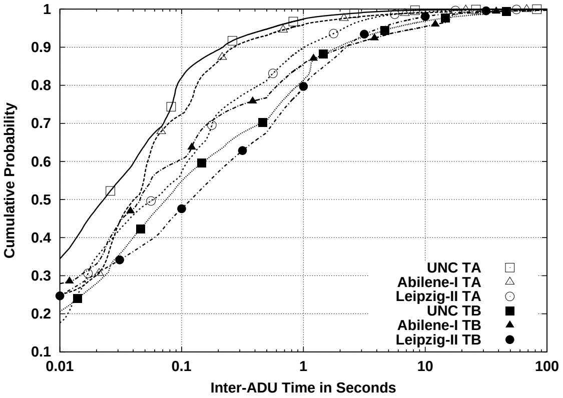


Figure 3.26: Bodies of the TA and TB distributions for Abilene-I, Leipzig-II and UNC 1 PM.

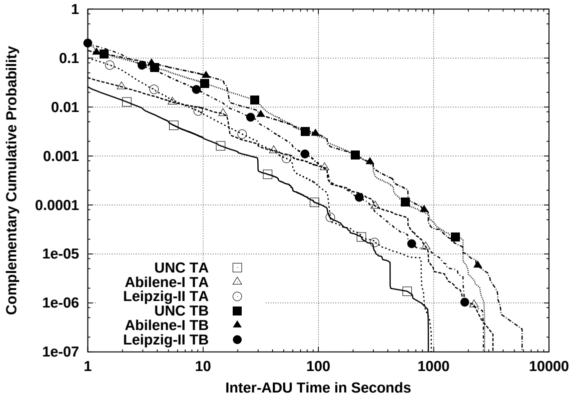


Figure 3.27: Tails of the TA and TB distributions for Abilene-I, Leipzig-II and UNC 1 PM.

The two distributions for UNC trace in Figure 3.23 are completely different (the median sizes for b-type ADU are much larger). There are, however, some epochs sizes between 25 and 50 for which a-type data units can be as large as b-type data units. Leipzig-II shows a completely different structure in Figure 3.24, where a-type ADUs are shown to be as large as b-type ADUs, and both are larger than UNC's a-type ADUs, and smaller than UNC's b-type ADUs. Abilene-I's distribution of b-type ADUs is similar to that of UNC. On the contrary, Abilene-I's distribution of a-type ADUs shows extreme variability for 60 epochs or more, and this phenomenon is completely absent in UNC's distribution. The conclusion of these four plots is clear: it is quite unrealistic to generate synthetic connection vectors using a simple model that assumes independence between ADU sizes and number of epochs.

Figure 3.26 examines the distributions of quiet times between ADUs. Shown are the distributions TA for ta_j and TB for tb_j. Note that the quiet times between the last ADU and connection termination, *i.e.*, tb_j for the last epoch, are not included in TB. The plot shows that, as the durations of the quiet times increase, the bodies of the TA distributions become increasingly lighter than those of the TB distributions. This is consistent with our understanding of client/server applications. Inter-epoch quiet times (TB) are usually user-driven, while intra-epoch quiet times (TA) are usually due to server processing delays. Server processing delays should generally be far shorter than user think times. For UNC and Abilene-I, most of the probability mass of TA is below 100 milliseconds, while that of TB is spread more widely. This is a strong indication that quiet times on the order of a few hundred milliseconds mostly reflect source-level quiet times. Observing TA being significantly lighter than TB is explained by the presence of user think times. Neither network delays nor the location of the monitor can provide an alternative explanation of the difference, since both factors have exactly the same impact on both distributions. The bodies Leipzig-II's TA and TB distributions are substantially heavier than the corresponding bodies of the other two traces. This could be due in part to network-level components of these distributions. Since Leipzig is in Europe, clients in the Leipzig-I trace suffer far longer round-trip times to US servers than clients found in the UNC and Abilene-I traces.

Unlike the bodies, the tails of the distributions shown in Figure 3.27 do not show the same difference between Leipzig-II and the other traces. This is consistent with the expectation that these longer quiet times are completely dominated by source-level behavior, and not by the impact of network location (*i.e.*, Europe *vs.* U.S.A.). We observe that Abilene-I's and UNC's TB are both substantially heavier than Leipzig-II's TB. Also, Leipzig-II's TA becomes lighter than Abilene-I's TA for quiet times above 11 seconds. Interestingly, we also find a similar shape for the two heaviest tails, Abilene-I's and UNC's TB, which came from traces of very different durations (2 hours *vs.* 1 hour). This provides strong evidence that trace boundaries are not introducing artifacts in our characterization of inter-ADU quiet times, despite the hard upper limit that trace duration imposes on quiet time duration.

Figure 3.28 shows the distribution of extra quiet times between the last ADU in a connection and TCP's connection termination. In the UNC and Abilene-I traces, 84% of the connections had extra quiet times below 1 second. The extra quiet time is actually zero for 83% of the cases, where the last segment of the last ADU had the FIN flag enabled. Leipzig-II showed an even higher percentage, 65%, of long quiet times after the last ADU. In all cases, we find large jumps in the probability for some values (*e.g.*, 7, 11 and 15 seconds). Moreover, the tails are surprisingly long. Since most connections transfer small amounts of data, this high frequency of extra quiet times has an important impact on the lifetimes of TCP connections observed from real links, and play an important role in realistic traffic generation.

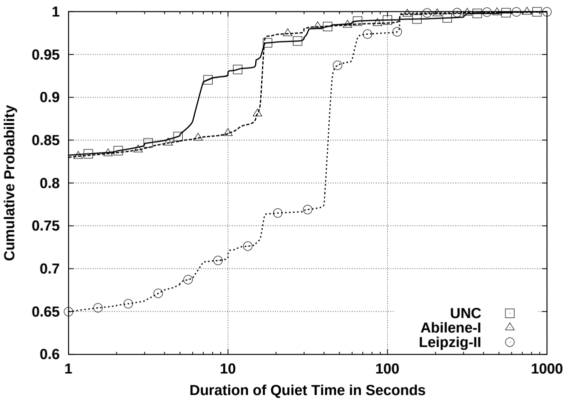


Figure 3.28: Distribution of the durations of the quiet times between the final ADU and connection termination.

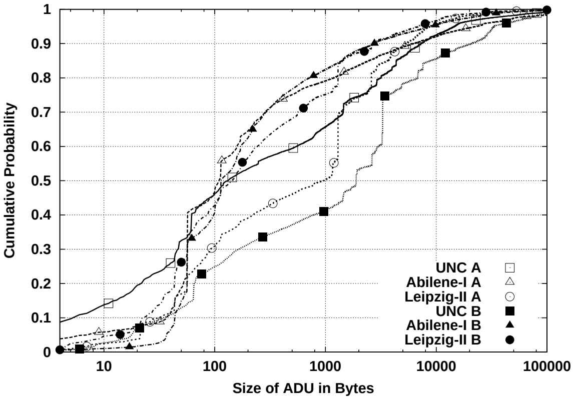


Figure 3.29: Bodies of the A and B distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

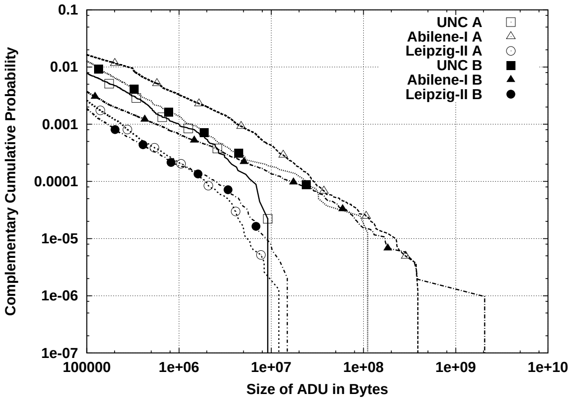


Figure 3.30: Tails of the A and B distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

Concurrent Connections

Concurrent connections exhibit substantially different distributions. Figure 3.16 showed distributions of a-type ADU sizes with bodies that were clearly lighter than those of b-type ADU sizes. In contrast, Figure 3.29 shows that concurrent connections made use of larger a-type ADUs, and that the shapes of A and B are not consistent across sites. Abilene-I does not show any significant difference between A and B, while Leipzig-II and UNC distributions do show a heavier B. The tails of these distributions shown in Figure 3.30 are as heavy as those for sequential connections, with the same three distributions (Abilene-I's A and B and UNC's B) having much longer tails that the other three. This phenomenon is far more striking for concurrent connections.

The distributions of quiet time durations shown in Figure 3.31 reveal that concurrent connections do

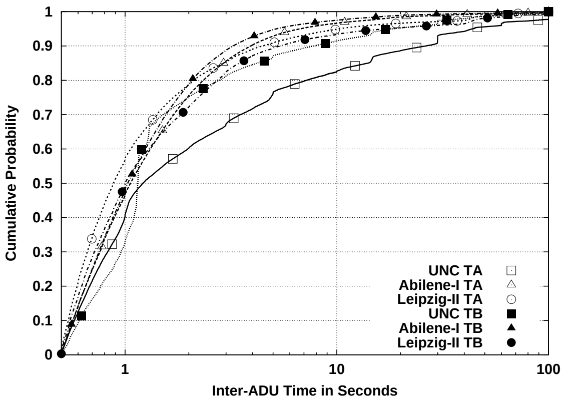


Figure 3.31: Bodies of the TA and TB distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

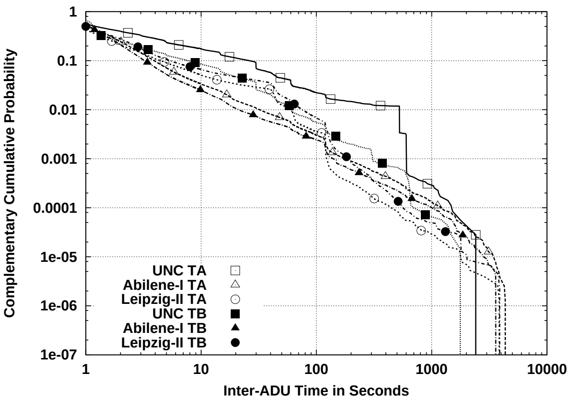


Figure 3.32: Tails of the *TA* and *TB* distributions for the concurrent connections in Abilene-I, Leipzig-II and UNC 1 PM.

not exhibit the clear separation between TA and TB that was observed for the sequential connections in Figure 3.26. This is consistent with the motivations for using concurrent data exchanges given in section 3.2. Connections that use concurrency to improve throughput by keeping the pipeline full do so to reduce the impact of user delays and client processing, thereby making TB lighter. Connections used by applications that are naturally concurrent should not exhibit any systematic difference between TA and TB distributions. Note that the minimum quiet time was 500 milliseconds, which was the duration of our threshold separating ADUs in concurrent connections.

The TA distribution for concurrent connections is significantly heavier for UNC. This suggests the presence of a concurrent application at UNC that is rather asymmetric and that is not so common in Abilene-I and Leipzig-II. The tails of the TA and TB distributions for concurrent connections shown in Figure 3.32 exhibit similar shapes and lengths to those found for sequential connections.

3.5.2 Time-of-Day Variability and Workload Directionality

The previous analysis illustrated the variability of the a-b-t distributions when several sites are compared. It also pointed out a number of features that are consistent with the communication patterns that motivate our models. TCP workloads at the same site can also exhibit significant differences, as the set of dominant applications changes throughout the day. For example, we expect to find substantial traffic from applications that are used for study and work activities (*e.g.*, e-business, research digital libraries) from 8 AM to 5 PM in the academic environment. In contrast, our guess is that traffic from gaming and other leisure time applications should be more common after 5 PM, mostly coming from

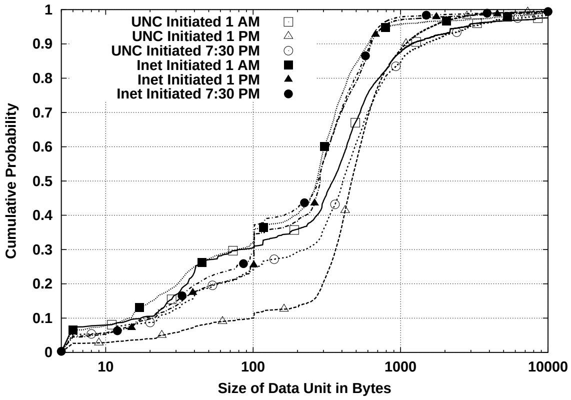


Figure 3.33: Bodies of the *A* distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

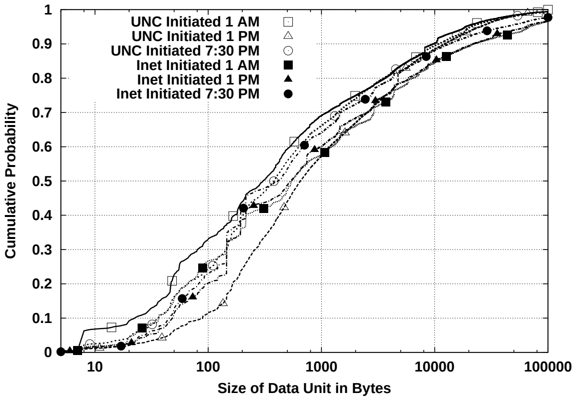


Figure 3.34: Bodies of the *B* distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

the dorms where students live. This change in the mix of applications should have an impact on the source-level properties of the traffic.

Another important dimension of traffic variability that was not considered in the previous section was the fact that traffic may be asymmetric. For example, traffic created by UNC clients is representative of the network activity of a large population of users (30,000) that can access any kind of service on the Internet. On the contrary, traffic created by clients from outside UNC is representative of the type of services that an academic institution offers to the rest of the Internet. This dichotomy should have an impact on the source-level properties of the traffic, as traffic from UNC's connection initiators is expected to be driven by a rather different mix of applications than that of UNC's connection acceptors.

Figure 3.33 provides a first illustration of the impact of these two kinds of variability on source-level properties. The plot shows *A* distributions for sequential connections observed at UNC during three different intervals (1 to 2 AM, 1 to 2 PM, and 7:30 to 8:30 PM). The plots separate data from connections initiated by UNC clients (labeled "UNC Initiated") and data from connections initiated by clients outside UNC (labeled "Inet Initiated"). The significant difference between *A* distributions for UNC initiators is in sharp contrast with the quite similar *A* distributions for UNC acceptors. This shows that time-of-day variation is substantial for connections initiated at UNC, but not for connections initiated outside UNC. This is consistent with the observation that UNC services, such as the large software repository `ibiblio.org`, are available 24 hours a day, and they serve clients from different parts of the world throughout the entire day. On the contrary, the activities of UNC clients are a function of campus activity and its evolution along a diurnal cycle. The distributions of b-type ADU sizes in Figure 3.34 also reflect this dichotomy. The *B* distributions on UNC initiated connections for

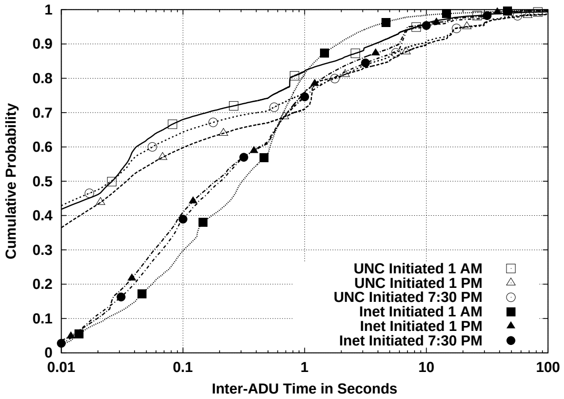


Figure 3.35: Bodies of the TB distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

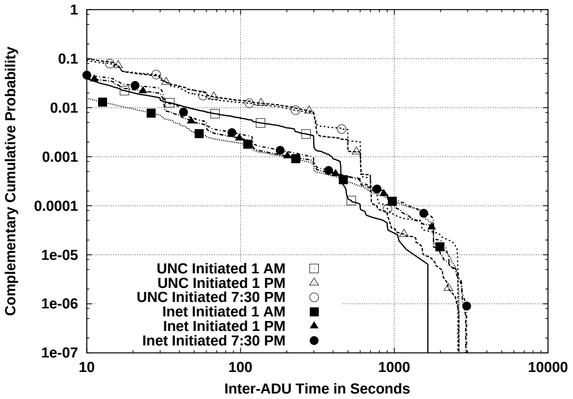


Figure 3.36: Tails of the TB distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

the 1 AM and 1 PM traces form an envelope around the other distributions, while the three distributions for non-UNC initiators are remarkably similar.

Figure 3.35 serves to illustrate the impact of monitor location on the measurement of quiet times. UNC traces were collected on the border link between UNC and the rest of the Internet. This means that the monitoring occurred very close, in terms of delay, to UNC clients and UNC servers. Going back to the diagram in Figure 3.10, this means that connections initiated from UNC are seen from the first monitoring point (very close to the client), while those initiated from outside UNC are seen from the second monitoring point (very far from the client). As a consequence, TB distributions from UNC clients, which measure the time between the end of a response b_j and the beginning of a new request a_{j+1}, are observed much closer to the clients, and are characterized very accurately. TB distributions from non-UNC clients are measured much further from the client, so they tend to overestimate true quiet times. As discussed before, this type of inaccuracy is a function of round trip time. This is clearly shown in Figure 3.35, where TB distributions from UNC initiators are much lighter than those for non-UNC initiators for quiet times below 1 second. As quiet times get larger and larger, the inaccuracy due to the placement of the monitoring point becomes less and less significant. The crossing points of the distributions between 500 milliseconds and 1 second suggest that the characteristics of applications and user behavior start to dominate measured quiet times above a few hundred milliseconds.

The same observations regarding the impact of the monitoring point also holds for the TA distributions in Figures 3.37 and 3.38. Here the effect of the monitoring point is reversed: ta_j is observed far from the client for UNC initiated connections, and close to the client for non-UNC initiated connections).

Time-of-day effects are less clear in Figure 3.35. If we look at quiet times above 1 second (the relevant

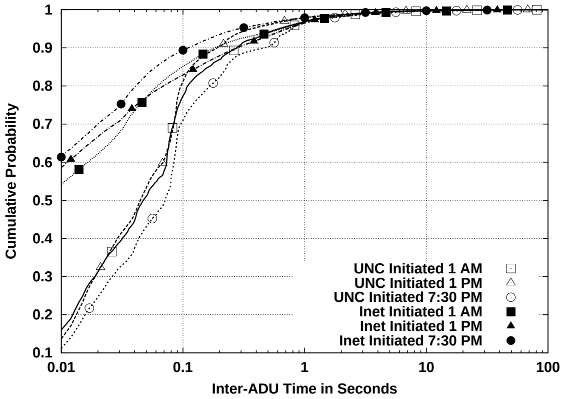


Figure 3.37: Bodies of the TA distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

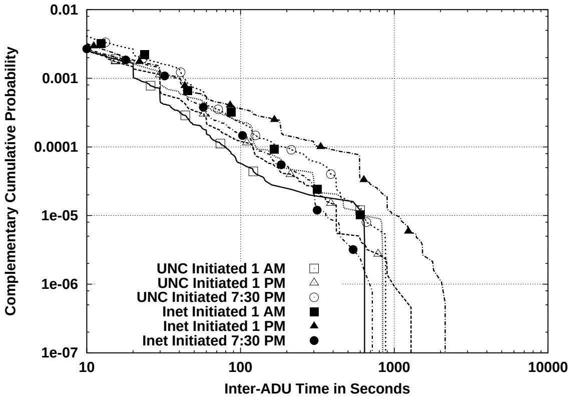


Figure 3.38: Tails of the TA distributions for UNC 1 AM, UNC 1 PM and UNC 7:30 PM.

ones), we can see that the distributions for 1 PM and 7:30 PM are quite similar for both directions, while those for 1 AM are lighter and not consistent with each other (especially for UNC acceptors). This is also true for the tails of these distributions shown in Figure 3.36 for quiet times below 500 seconds. The tails of the TA distributions in Figure 3.38 do not show any consistent pattern (*i.e.*, no grouping based on time-of-day or directionality). They are also somewhat lighter than the TB distributions.

3.6 Summary

This chapter presented our method for describing source-level behavior in an abstract manner using the *a-b-t model*. The basic observation behind this model is that the job of a TCP connection is to transfer one or more application data units (ADUs) between two network endpoints. TCP is sensitive to the sizes of these ADUs, which determine the number of segments required to transfer them, but it is insensitive to the actual semantics of each ADU. Consequently, we proposed to describe the source-level workload of TCP connections in terms of ADUs, characterizing their number, order, and sizes. Additionally, we also observed that applications may remain inactive during long periods of time (*e.g.*, during user think times), which often results in TCP connections that last far longer than required to transfer their ADUs. This motivated us to also incorporate quiet times into our generic descriptions of source-level behavior. We formulated these ideas into the a-b-t model, which describes source-level behavior in abstract terms common to all applications. The model distinguishes a-type ADUs, sent from the connection initiator to the connection acceptor, and b-type ADUs, sent in the opposite direction the connection. It also distinguishes between quiet times due to inactivity on the initiator endpoint and due to inactivity on the acceptor endpoint.

Our analysis of TCP connections observed on real Internet links revealed two types of source-level behavior, which motivated us to develop two different versions of our a-b-t model. Most TCP connections exchange ADUs in a sequential, alternating manner, where *a-type* ADUs usually play the role of request from client and *b-type* ADUs usually play the role of responses from servers. We describe this first type of source-level behavior using the *sequential version* of our a-b-t model, which consists of a sequence of epochs, where each epoch captures one exchange of ADUs (*i.e.*, one a-type ADU and one b-type ADU). The rest of the TCP connections exhibit *data exchange concurrency*, where their endpoints send at least one pair of ADUs simultaneously. We describe this second type of source-level behavior using the *concurrent version* of our a-b-t model, where the ADUs and the quiet times from each endpoint are described independently. The examples from real applications examined in this chapter demonstrated the ability of the a-b-t model to provide a detailed description of source-level behavior for both sequential and concurrent data-exchanges. This means that our approach is able to characterize the source-level behavior of *entire traffic mixes* without any need to understand the specific semantics of each individual application present in the mix.

A fundamental strength of abstract source-level modeling is the possibility of acquiring data from packet header traces in an efficient manner. This is critical to make the approach widely applicable. Packet header traces do not contain any application-level payload, so they are easy to anonymize simply by replacing IP addresses. As a consequence, many organizations have made packet header traces of their Internet links public [nlab]. We proposed a data analysis algorithm that can transform the set of segment headers observed for each connection in a trace into an a-b-t connection vector. The cost of this algorithm is $O(sW)$, where s is the number of segments and W the maximum window size. The algorithm relies on the concept of *logical data order* (*i.e.*, the order of data as understood by the application layer) to robustly handle segment reordering and retransmission. This approach enables us to measure the real size of ADUs at the application level, to distinguish between source-level quiet times and quiet times due to losses, and to identify data exchange concurrency without false positives. We validated this algorithm using synthetic applications, studying the impact of the sizes of socket reads and writes, delays between socket operations and packet loss. The results demonstrated that our data acquisition algorithm is very accurate. Our validation also studied the accuracy of our data acquisition when our basic algorithm is extended with a quiet time threshold to separate consecutive ADUs flowing in the same direction. Even in this case, we only uncovered minor inaccuracies in the measured inter-ADU quiet times when arbitrary delays between socket reads are used and when connections suffered from packet loss.

We concluded the chapter with a statistical analysis of the a-b-t connection vectors in five packet header traces. Three of these traces came from our own data collection effort at the University of North Carolina at Chapel Hill, and the other two traces, Leipzig-II and Abilene-I, came from NLANR's public repository of packet header trace. Before we presented the analysis, we pointed out the need to filter out the following two types TCP connections:

- Connections for which no observed segment carried application data, and therefore had no ADUs. They corresponded to failed attempts to establish a TCP connection (*e.g.*, due to closed ports), denial-of-service attacks (*e.g.*, SYN attacks), and port scanning activity. These connections were very numerous, but they carried an insignificant fraction of the total traffic in each trace. Properly characterizing these "ADU-less" connections is outside the scope of this dissertation.
- Connections for which segments are observed in only one direction. We found a significant number of unidirectional connections only in the case of Abilene-I, since this trace was collected traffic in a backbone network where asymmetric routing was common. Distinguishing between sequential and concurrent connections require to observe both directions of a connection, so we ignored unidirectional connections in our later analysis and traffic generation.

In addition, our statistical analysis of the traces considered only fully-captured TCP connections, those for which we observed both the segment performing connection establishment and connection termination. We therefore ignored partially-captured connections, which contained only partial information about source-level behavior. Our results considered sequential and concurrent connections separately. We can highlight the following observations from these results:

- Every trace showed a small fraction of concurrent connections, at most 3.6%, but they account for a far more substantial fraction of the total bytes, between 18% and 32%. This is consistent with our observation that concurrency can increase throughput, so it is often implemented in bulk applications that transfer large amounts of data.
- Regarding the bodies of distributions of ADU sizes, sequential connections showed a substantial difference between a-type and b-type ADUs. The sizes of 90% of the a-type ADUs were at most 1,000 bytes, while the sizes of 90% of the b-type ADUs were at most 10,000 bytes. The observed differences across sites paled in comparison to this phenomenon. On the contrary, the tails of the distributions appeared similar for a-type and b-type ADUs, being consistent with heavy-tailness in both cases. Concurrent connections did not show a systematic difference between a-type and

b-type ADUs, but their size distributions varied widely for the three sites and also exhibited heavy-tailness. Another interesting observation is that between 80% and 90% of the bytes were carried in ADUs whose size was above 10,000 bytes.

- Regarding the distribution of the number of epochs, we found a large fraction of connections, between 57% and 65%, with only one epoch. However, these connections accounted for a far smaller fraction of the total bytes, between 22% and 38%. Most of the remaining connections had a moderate number of epochs, between 2 and 10. Connections with tens or hundreds of epochs represented only 5% of the connections, but they carried 30% to 50% of the bytes.

- Our joint analysis of ADU size and number of epoch revealed a complex inter-dependency. The average amount of data in an epoch and the median size of ADUs showed substantial variability for different values of the number of epochs in a connection, without any apparent pattern. In addition, the results of the joint analysis are very different across sites. It does not seem possible to develop a simple parametric model for these data.

- Regarding the bodies of the distributions of quiet times, sequential connections showed a larger fraction of durations above 1 second for quiet times on the client side, between a b-type ADU and the a-type ADU that follows it. Quiet times on the server side, between an a-type ADU and the following ADU, were less substantial but also significant. This motivated us to incorporate server-side quiet times on our model. Both distributions showed substantial tails. The difference between the two distributions of quiet time durations appear less significant for concurrent connections.

- A significant percentage of connections, between 65% and 83%, showed a quiet time between the last ADU and TCP's connection termination with a duration above 1 second. This quiet time often increased the duration of the connection dramatically, since connections with little data completed their data transfer very quickly, but remained idle waiting to be closed. This finding justified the addition of a final quiet time duration to our a-b-t model.

- Our comparison of the distributions from the three UNC traces, which were collected at three different times of the day, revealed clear differences in the data. These differences are however less dramatic than those observed when traces from three different sites are compared.

CHAPTER 4

Network-Level Parameters and Metrics

If you are distressed by anything external, the pain is not due to the thing itself, but to your estimate of it; and this you have the power to revoke at any moment.

— Marcus Aurelius (121–180)

Reality continues to ruin my life.

— Bill Watterson (1958–), Calvin and Hobbes

The workload of TCP connections represents the demands of applications for sending and receiving data in a reliable, ordered, and congestion-responsive manner. How well TCP can satisfy these demands depends on the conditions of the network path between the two endpoints of each TCP connection, and the way TCP reacts to these conditions. An obvious example of a network condition that affects TCP is congestion that leads to segment loss. When a data segment is lost, TCP must retransmit it, and this implies some reduction in performance (*e.g.*, throughput) as the same data segment (rather than a new one) has to be sent again. In addition, TCP considers loss as an indication of network congestion, and reacts by reducing its sending rate. Different versions of TCP implement different ways of adjusting this sending rate. This means that the characteristics of the set of segments in a TCP connection are not just a function of the source-level behavior of the endpoints. This fact will have profound implications for the validation of our approach to synthetic traffic generation.

Intuitively, demonstrating that synthetic traffic is "realistic" must be based on a comparison of the statistical properties of real and synthetic traffic. If these properties are reasonably approximated, we can argue with confidence that the traffic generation method and its underlying statistical model provide an adequate foundation for experimental networking research. The comparison can be performed at two levels. First, we can compare source-level properties using the a-b-t modeling approach (see for example section 3.5). Second, we can compare network-level properties, *i.e.*, properties of the actual segments that make up individual connections in real and generated traffic. The material in this chapter is concerned with developing methods for making this latter comparison meaningful.

Since network conditions have an important impact on TCP connections, comparing real and synthetic traffic at the network-level is difficult if network conditions are not incorporated to some extent into the traffic generation system. For example, if we generate traffic that is intended to resemble that of some real link, and connections on this link experience substantial loss rates, the characteristics of the synthetic traffic would be rather different if the synthetic traffic did not experience comparable loss rates. Otherwise, the synthetic traffic would experience higher transfer rates, shorter durations, *etc.* The first part of this chapter considers methods for characterizing three important, and perhaps the dominant, network-level properties of TCP connections: round-trip times, receiver window sizes, and loss rates. These three properties will be incorporated in our traffic generation method as input parameters, and will make synthetic traffic more comparable to real traffic. Additionally, we also examine the properties of a number of real traces to illustrate the wide range of network conditions in which TCP operates, and how this range changes from one network link to another.

The second part of the chapter considers the actual problem of comparing traffic at the network-level. The research literature has identified a number of statistical properties of traffic that can serve as metrics for assessing the realism of synthetic traffic. We describe these properties and consider their application in the context of comparing traffic traces. We also examine a number of real traces in light of these metrics. Our analysis reveals important differences between the traces, and uncovers some dependencies between network-level metrics and types of source-level behavior.

4.1 Network-level Parameters

4.1.1 Round-Trip Time

The Round-Trip Time (RTT) between two network hosts is defined as the time required to send a packet from one host to another plus the time required to send a packet in the reverse direction. These two times are often very similar, but may sometimes vary considerably (*e.g.*, in the presence of asymmetric routing). In general, round-trip times are not constant, since queuing delays, switching fabric contention, route lookup times, *etc.*, vary over the lifetime of a connection.

Impact of Round-Trip Time

Round-trip times play a very important role in TCP connections. As indicated in Chapter 3, the exchange of a request ADU and its response ADU (*i.e.*, an epoch) in a TCP connection requires at least one round-trip time. This is independent of the amount of data exchanged. In addition, the speed at which data can be delivered (known as throughput[1]), is also a function of the round-trip time of the TCP connection.

The minimum time between the sending of a data segment and the arrival of its corresponding acknowledgment is exactly one round-trip time. Without TCP's window mechanism, TCP would only be able to send one segment per round-trip time, since it would have to wait for the acknowledgment before sending the next data segment. Therefore, peak throughput would be given by the maximum segment size S divided by the round-trip time R. This would imply that the longer the round-trip time, the lower the throughput S/R of the connection would be. In order to increase performance, a TCP endpoint can send a limited number of segments, a *window*, to the other endpoint before receiving an acknowledgment for the first segment. The number of segments W in the window gives the peak throughput of a TCP connection, $\frac{W*S}{R}$. This peak throughput can be lower if the path between the two endpoints has a capacity C that is lower than $\frac{W*S}{R}$, so the peak throughput of a TCP connection is given by $min(\frac{W*S}{R}, C)$. This implies that if W is not large enough to fill the available capacity C, R is the limiting factor in the peak throughput of a TCP connection.

A new TCP connection is not allowed to reach its peak throughput until it completes a "ramp-up" period known as *slow start* [Pos81]. The throughput of TCP during slow-start is also highly dependent on round-trip time. At the start of each connection, TCP does not make use of the entire window to send data, but rather probes the capacity of the network path between the two endpoints by sending an exponentially increasing number of segments during each round-trip time. This normally means that TCP sends only 1 segment in the first round-trip time, 2 in the second one, 4 in the third one, and so on, doubling the number of segments after each round-trip time until this number reaches a maximum of W segments. The throughput of the slow-start phase is therefore a function of round-trip time and maximum segment size, but it depends little on receiver window size and capacity. For example, an ADU that fits in 4 segments, requires 3 round-trip times to be transferred in the slow-start phase (one segment is sent in the first round-trip time, two in the second one, and one more in the final one), so

[1] More precisely, throughput is the rate of transfer taking into account not only application data but also control headers added by TCP and lower network layers. A related concept, goodput, is the rate of transfer of application data, *i.e.*, TCP payload. This distinction is important, but in the discussion above, throughput and goodput are affected similarly by round-trip times, so we simply talk about throughput.

the throughput of the connection is $\frac{4S}{3R}$. For common values of R and S, $S = 1460$ bytes and $R = 100$ milliseconds, the throughput would be 156 Kbps. This same ADU sent later in the connection using a single window would achieve a much higher throughput (*e.g.*, the four segments could be sent back to back, so they would reach the destination after only one half the round-trip time, $\frac{R}{2}$, achieving a throughput of $\frac{8S}{R} = 934$ Kbps).

Passive Estimation of Round-Trip Times

The dependency between TCP throughput and round-trip time implies that the distribution of round-trip times of the TCP connections found on a link has a substantial impact on the characteristics of a trace. If we intend to compare the throughputs of connections in traces from real links with those in synthetic traces, traffic generation must employ similar round-trip times. This requires us to be able to extract RTTs from a trace by analyzing packet dynamics. Extracting round-trip times from packet traces has received only limited attention in the literature [JD02, AKSJ03]. Nonetheless we can refine some of the existing ideas to obtain the distribution of round-trip times of connections in a trace in a manner that is useful for traffic generation.

Before we describe several methods for characterizing round-trip times, it is important to point out that the round-trip time of a TCP connection is not a fixed quantity. The time required for a segment to travel from one endpoint to another has several components. Transmission and propagation delays are more or less constant for a given segment size, but queuing delays, medium access contention, and router and endpoint processing, introduce variable amounts of extra delay. The TCP segments observed in our traces are exposed to these delays, whose variability is not always negligible, as our later measurement results illustrate. In summary, the segments of a TCP connection are exposed to a distribution of round-trip times, rather than to a fixed round-trip time.

We can think about the segments of a TCP connection as probes that sample the dynamic network conditions along their path, experiencing variable delays. As shown in the previous chapter, most TCP connections carry a small amount of data, providing only a few samples of these underlying conditions. This makes it very difficult to fully characterize the distribution of round-trip times experienced by an individual connection using only passive measurement methods (*i.e.*, only by looking at packet headers). In addition to the low number of samples per connection, TCP's delayed acknowledgment mechanism adds extra delays to some samples. This introduces even more variability, this time unrelated to the path of the connection. As we discuss below, the presence of delayed acknowledgments makes statistics

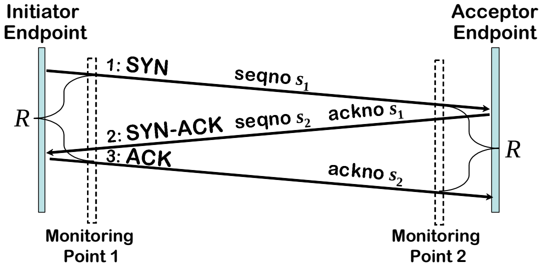


Figure 4.1: A set of TCP segments illustrating RTT estimation from connection establishment.

(such as the mean and standard deviation) computed from RTT samples, grossly overestimate the true mean and standard deviation of the underlying distribution of round-trip times. In our work, we favor more robust statistics, such as the median, or the minimum, which provide a good way of characterizing the non-variable component of a connection's round-trip time. For simplicity, our traffic generation will simulate the minimum round-trip time observed for each connection.

The SYN Estimator

The simplest way of estimating the round-trip time of a connection from its segment header is to examine the segments sent by the initiator endpoint during connection establishment. The use of this *SYN estimator* is illustrated in Figure 4.1. The initial SYN segment sent from the initiator to the acceptor is supposed to be immediately acknowledged by a SYN-ACK segment sent in the opposite direction. The initiator endpoint would then respond to the SYN-ACK segment[2] with an ACK segment. The initiator may or may not piggy-back data on this segment, but this does not affect RTT estimation significantly. The time between the arrival of the SYN segment and the arrival of the ACK segment is the round-trip time R of the connection (more precisely, a sample of the round-trip time). Measuring R using the departure times of the SYN and the ACK segments from the initiator endpoint gives approximately the same result as measuring R using the arrivals of these segments at either the monitoring point or the connection acceptor. In general, the SYN estimator is a good indicator of the minimum round-trip time, *i.e.*, total transmission and propagation delay. This is because TCP endpoints respond immediately[3] to connection establishment segments, and also because the small packets used by the SYN estimator are less likely to encounter queuing delays that the larger ones found later in the connection. The SYN

[2] For simplicity, our illustrations use acknowledgment numbers that refer to the cumulative sequence number accepted by the endpoint, which is one unit below the actual acknowledgment number stored in the TCP header [Pos81].

[3] Endpoints are not required to behave in this manner by any RFC, but it makes little sense to delay the acknowledging of SYN segments. On the contrary, delaying the acknowledging of data segments gives the endpoints a chance to receive a second data segment and acknowledge both data segments using a single acknowledgment.

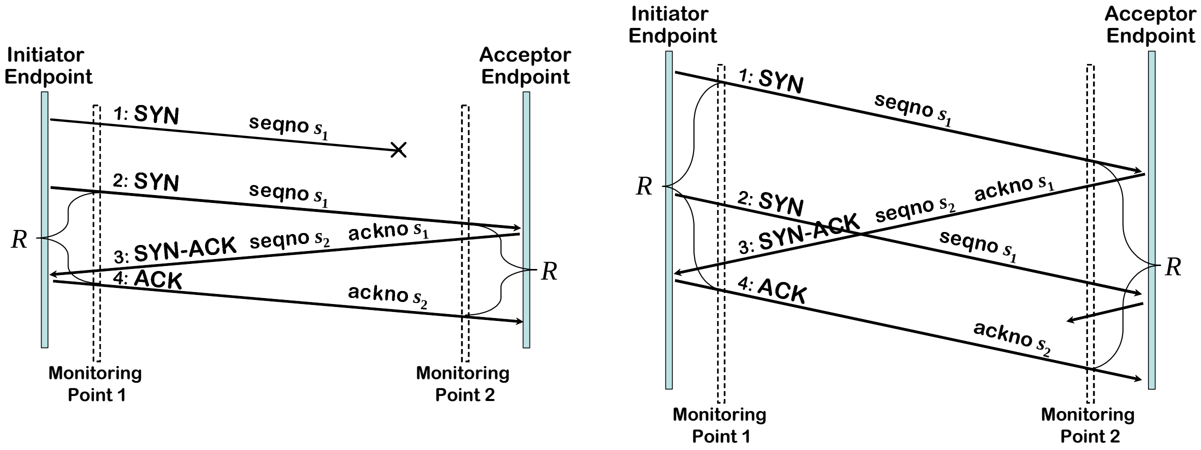


Figure 4.2: Two sets of TCP segments illustrating RTT estimation ambiguities in the presence of loss (left) and early retransmission (right) in connection establishment.

estimator has been a popular means of estimated round-trip times [AKSJ03, CCG$^+$04].

The SYN estimator has a number of shortcomings. First, the estimator provides a single sample of the round-trip time, which may be a poor representative of the average round-trip time of the connection. Second, partially-captured connections in a trace may not include connection establishment segments, so the SYN estimator cannot be used to determine their round-trip times. Third, the round-trip time of a connection with a retransmission of the SYN segment (or of the SYN-ACK segment), cannot be estimated with confidence, since the coupling of the SYN and the ACK segments becomes ambiguous. The problem is that the monitor may see two instances of the SYN segment, and either one could be coupled with the ACK for the purpose of computing the RTT. This difficulty is illustrated in Figure 4.2. The left side of the figure shows an example of connection establishment in which the first SYN segment is lost. In this case R is the time between the arrival of the second SYN segment and the arrival of the ACK segment, and not the time between the first SYN segment and the arrival of the ACK segment. However, it is not always correct to couple the last retransmission of the SYN with the ACK, and this is illustrated in the right side of the figure. The diagram shows a connection with such a large R that the initiator endpoint times out before the receipt of the SYN-ACK and sends an *early* (*i.e.*, unnecessary) retransmission of the SYN before the SYN-ACK reaches the initiator. In this case, R should be computed as the difference between the arrival times of the first SYN and the ACK, and not between the arrival times of the second SYN and the ACK. Note that standard TCP implementations time out and retransmit SYN segments after 3 seconds without receiving an acknowledgment [APS99]. The two cases result in exactly the same sequence of segments observed at the monitoring point, so it is not generally possible to accurately choose the right SYN to couple with the corresponding ACK

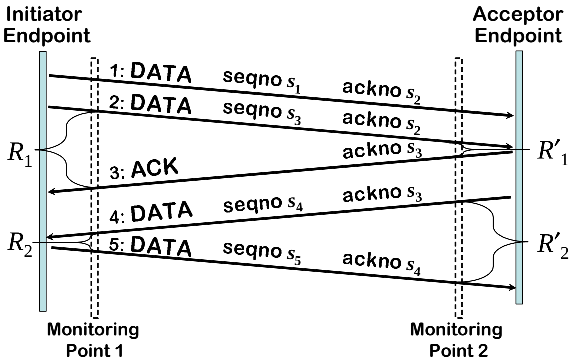

Initiator Endpoint
Acceptor Endpoint
1: DATA seqno s_1 ackno s_2
2: DATA seqno s_3 ackno s_2
3: ACK ackno s_3
4: DATA seqno s_4 ackno s_3
5: DATA seqno s_5 ackno s_4
R_1
R_2
R'_1
R'_2
Monitoring Point 1
Monitoring Point 2

Figure 4.3: A set of TCP segments illustrating RTT estimation using the sum of two OSTTs.

segment by looking only at the sequence of segments [KP88a, Ste94].

The early retransmission of SYN segments when the RTT is greater than 3 seconds implies that the simple SYN estimator, at least in this basic form, cannot be used to study the tail of the round-trip time distribution (this issue has been overlooked in the literature). In theory, one could disambiguate the case of a timed-out SYN-ACK using the observation that SYN segments are retransmitted only after 3 seconds without receiving the SYN-ACK [Bra89]. However, our empirical observations show that this heuristic is unreliable as the timing of arrivals is imprecise, and not all TCP implementations seem to use the 3-second timeout properly. Detection of an unexpected retransmission of the SYN-ACK (or the ACK) can also be used to develop a heuristic, but cases with multiple losses can be very complicated to disambiguate.

The OSTT Estimator

A second technique for estimating round-trip times is illustrated in Figure 4.3. The location of the monitor divides the path of a connection into two sides, and we can estimate the *One-Side Transit Time* (OSTT) independently for each side. The sum of the two OSTTs gives an estimate of the round-trip time of the connection. The idea is that the arrival times of a data segment and its acknowledgment segment at the monitor provides an estimation of the OSTT from the measurement point to one of the endpoints. Round-trip time estimation using the OSTT method requires the collection of one or more samples of the OSTT between the initiator and the monitoring point, and one or more samples of the OSTT between the acceptor and the monitoring point. In Figure 4.3, a sample R_1 of the OSTT for the right side of the path (*i.e.*, OSTT between the acceptor and the monitoring point) is given by the difference in the arrival times of segments 2 and 3. A sample R_2 of the OSTT for the left side of the path (*i.e.*, between the initiator and the monitoring point) is given by the difference in the arrival times of segments 4 and 5. Thus, a sample of the full round-trip time R is given by $R_1 + R_2$. One way of seeing this graphically is to do the mental exercise of shifting the monitoring point toward the initiator. As we do this, the R_1 increases, while R_2 decreases. When the monitoring point reaches the initiator endpoint, R_1 is exactly the round-trip time of the connection, and R_2 is zero.

The OSTT-based estimation of the RTT is independent of the location of the monitoring point. For example, the arrival of segments at the second monitoring point in Figure 4.3 provides a sample $R'_1 + R'_2$ which is equal to $R_1 + R_2$. This is a substantial improvement over existing methods, since it implies that we can perform RTT estimation for connections observed at any point on their path. Previous work, such as Aikat *et al.* [AKSJ03], constrained itself to traces collected very close the edge of the path, so they could assume that the delay between the monitoring point and local networks was minimal. This results in an estimate in which only R_1 is computed under the assumption that R_2 is very small. The use of the sum of the OSTTs is more flexible, since it makes it possible to extract RTTs from any trace, and not just edge traces. This allowed us to analyze a backbone trace like Abilene-I, making our traffic analysis and generation technique more widely applicable.

There are, however, a number of difficulties with OSTT-based round-trip time estimation. A first problem is that each pair of segments provides a different estimation of the OSTT (due to differences in queuing delay and other sources of delay variability), so we have to decide how to combine the OSTT samples from one side of the connection with those from the other side. In other words, each connection provides a set of OSTT samples for one side, $\{R_{11}, R_{12}, \dots, R_{1n}\}$, and another set of OSTT samples for the other side, $\{R_{21}, R_{22}, \dots, R_{2m}\}$, where $n \geq 0$ and $m \geq 0$ are not necessarily equal. The question is then how to combine these samples into a single estimate of R, which we will call $\hat{R}$. If we assume low variability, we could simply sum the means of the two sets of estimates,

$$\hat{R} = \frac{\sum_{i=1}^{n} R_{1i}}{n} + \frac{\sum_{i=1}^{m} R_{2i}}{m}. \tag{4.1}$$

However, as we discuss in the next section, the sum of means can introduce substantial inaccuracy due to TCP's delayed acknowledgment mechanism.

A second problem is that the sum of OSTT samples requires at least one sequence number/acknowledgment number pair for each side of the connection. Otherwise, one of the sets of OSTT samples is empty, and we have no information about the delay on one side of the connection. This prevents us from using the sum of OSTTs estimator for connections that send data only in one direction.

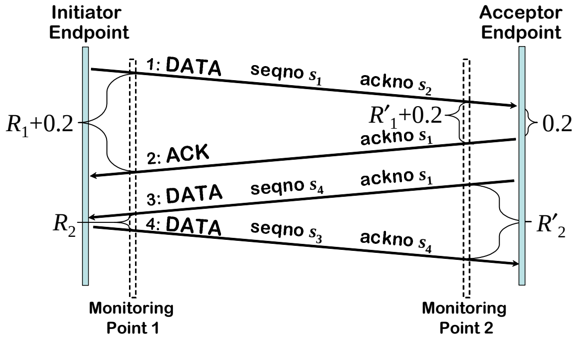


Figure 4.4: A set of TCP segments illustrating the impact of delayed acknowledgments on OSTTs.

Finally, we must note that the time between the arrival of a data segment and its first acknowledgment is not always a good estimator of the OSTT. This is mostly due to two causes: retransmission ambiguity and delayed acknowledgments. Retransmissions may create ambiguous cases in which we cannot match the pair of data and ACK segments. This is the well-known *retransmission ambiguity* problem, which was first discussed by Karn and Partridge [KP88b] in the context of estimation of TCP's retransmission timeout. Whenever a data segment is retransmitted, it is not possible to decide whether to compute the OSTT using the first or the second instance of the data segment. These data segments cannot therefore be used to obtain a new OSTT sample. This retransmission ambiguity is similar to the SYN retransmission problem shown in Figure 4.2.

Delayed acknowledgments can add up to 500 milliseconds[4] [Bra89] of extra delay in the OSTT estimates, whenever a segment is not acknowledged immediately. Figure 4.4 illustrates this problem. The right side OSTT is 200 milliseconds larger than it should be due to the delayed sending of the acknowledgment in segment 2. The distortion of OSTT samples caused by delayed acknowledgments is pervasive, since the number of segments in a window is often an odd number, and TCP implementations are allowed to keep (at most) one unacknowledged segment. An odd number of segments in a window means that the last segment does not trigger an immediate acknowledgment, which adds an extra delay to its corresponding sample. Furthermore, performance enhancement heuristics implemented in modern TCP stacks often add PUSH flags to TCP segments carrying data in the middle of an ADU, and this flag forces the other endpoint to immediately send an acknowledgment [Pos81]. This creates even more cases in which the last segment of the window has to be acknowledged separately using a delayed acknowledgment. The empirical results presented below illustrate the impact of this problem.

[4] Typical values are between 100-200 milliseconds.

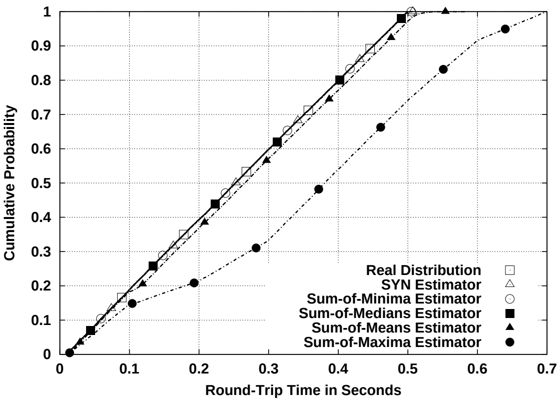


Figure 4.5: Comparison of RTT estimators for a synthetic trace: no loss and enabled delayed acknowledgments.

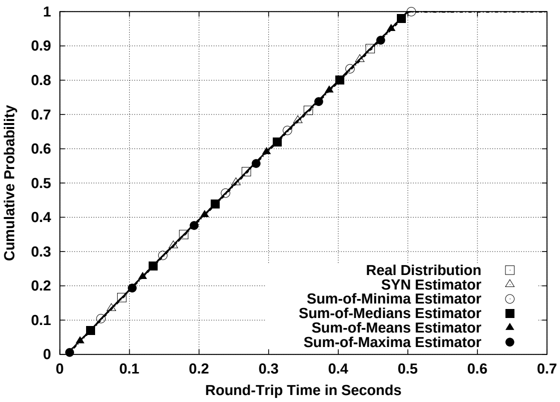


Figure 4.6: Comparison of RTT estimators for a synthetic trace: no loss and disabled delayed acknowledgments.

Validation of Round-Trip Time Estimators

We evaluated the round-trip time estimation techniques proposed above using synthetic traffic in a testbed where RTTs could be controlled precisely. Figure 4.5 shows the results of a first experiment, in which a uniform distribution of round-trip times between 10 and 500 milliseconds was simulated using a modified version of *dummynet* [Riz97]. Each connection had exactly the same round-trip time throughout its lifetime, so every one of its segments was artificially delayed by the same amount. During the experiment, a large number of single epoch connections was created. The sizes of a_1 and b_1 for each connection were randomly sampled from a uniform distribution with values between 10,000 and 50,000 bytes. We collected a segment header trace of the traffic and applied the round-trip time estimation techniques described above. Figures 4.5 and 4.6 compare the results. As shown in Figure 4.5 the SYN estimator can measure the distribution of round-trip times flawlessly in this experiment. The input distribution of RTTs (marked with white squares) exactly matches the distribution computed using the SYN estimator (marked with white triangles).

Figure 4.5 also studies the accuracy of several OSTT-based estimators. As discussed in the previous section, the analysis of the OSTTs in a TCP connection results in two sets of estimations, $\{R_{11}, R_{12}, \ldots, R_{1n}\}$ and $\{R_{21}, R_{22}, \ldots, R_{2m}\}$, for the initiator-to-monitor side and for the acceptor-to-monitor side respectively. For each connection, the estimated round-trip time $\hat{R}$ has to be derived from these collections of numbers. The figure shows the result of computing the distribution of round-trip times using four different methods of deriving $\hat{R}$. The first method is the sum-of-minima, where $\hat{R}$ is the sum of the minimum value in $\{R_{11}, R_{12}, \ldots, R_{1n}\}$ and the minimum value in $\{R_{21}, R_{22}, \ldots, R_{2m}\}$.

In the figure, the sum-of-minima estimation of the distribution of round-trip times (marked with white circles) is exactly on top of the input distribution, so this estimator is exact. The same is also true when the sum of medians is used. This shows that there is no significant variability between the minimum and the median of each set of OSTTs, which is expected in our uncongested experimental environment.

Figure 4.5 shows another two distributions derived from OSTT samples that are less accurate characterizations of the real RTT distribution in the testbed experiment. The distribution (marked with black triangles) of round-trip times obtained using the sum of the mean of the OSTTs, *i.e.*, Equation 4.1, is slightly heavier that the real distribution of round-trip times. This is due to the presence of a few OSTT samples that are above the real OSTT of the connection, which skew the mean but not the median or the minimum. The magnitude of these larger samples is strikingly illustrated by the curve corresponding to the sum of the maximum OSTTs (marked with back circles). This curve is far heavier than the previous one, and certainly a poor representative of the original distribution of round-trip times. The use of the maximum makes this last estimator focus on the largest OSTTs, which are shown to be quite far from the true values of the OSTT. The exact cause of this inaccuracy is the use of delayed acknowledgments in TCP, which was illustrated in Figure 4.4. Delayed acknowledgments make some OSTT samples include extra delays due to the behavior of the TCP stack and not the path between the endpoints. In particular, the distribution computed using the sum-of-maxima is 200 milliseconds heavier than the input distribution for most of its values. This is consistent with the default value of FreeBSD's delayed acknowledgment mechanism, which is 100 milliseconds. Connections where both the initiator-to-monitor and theR_1 acceptor-to-monitor sets of OSTTs have values from delayed acknowledgments result in values of $\hat{R}$ equal to $R + 100 + 100$ milliseconds.

To confirm this hypothesis, we conducted a second experiment, with exactly the same setup, although this time TCP's delayed acknowledgment mechanism was completely disabled. The results of estimating the distribution of round-trip times in this second experiment are shown in Figure 4.6. Every estimation method is accurate in this case, which proves our hypothesis about the impact of delayed acknowledgments. The conclusion is that the first three estimators are preferable, since they are robust to the inaccuracy that delayed acknowledgments introduce when measuring round-trip times from segment header traces. Interestingly, the impact of delayed acknowledgment on passive RTT estimation has been overlooked in the literature [Ost, AKSJ03, JD02].

The discussion of the RTT estimation methods in the previous section pointed out the need to filter out samples from retransmissions. The previous two experiments were run in an uncongested testbed,

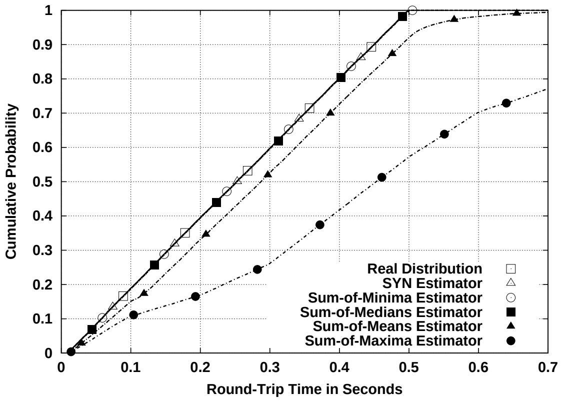


Figure 4.7: Comparison of RTT estimators for a synthetic trace: fixed loss rate of 1% for all connections.

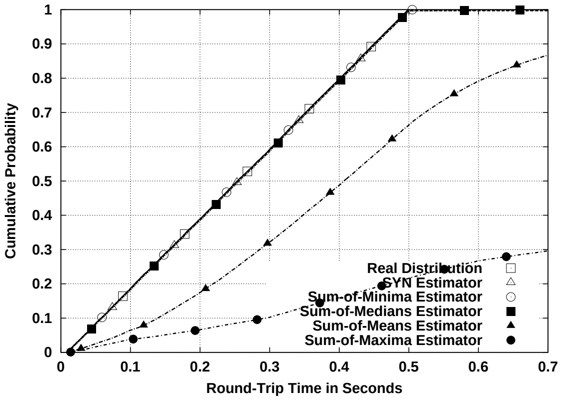


Figure 4.8: Comparison of RTT estimators for a synthetic trace: loss rates uniformly distributed between 0% and 10%.

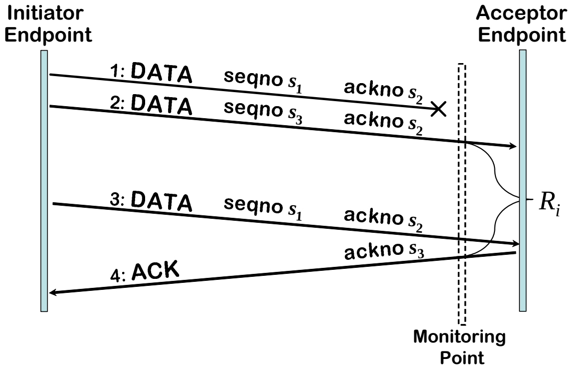

Initiator Endpoint
Acceptor Endpoint
1: DATA seqno s_1 ackno s_2
2: DATA seqno s_3 ackno s_2
3: DATA seqno s_1 ackno s_2
ackno s_3
4: ACK
R_i
Monitoring Point

Figure 4.9: A set of TCP segments illustrating an invalid OSTT sample due to the interaction between loss and cumulative acknowledgments.

where no losses were expected. Since loss is common in the real traces that we study in this dissertation, we further validated these methods using experiments where *dummynet* was used to introduce artificial loss rates under our control. Figure 4.7 compares the six distributions obtained using the six RTT estimators in an experiment with a fixed loss rate of 1%. Once again the first three estimators measure the distribution of physical round-trip times accurately, while the sum-of-means and the sum-of-maxima overestimate the true distribution. The overestimation is even more pronounced in another experiment in which loss rates were uniformly distributed between 0% and 10%. The estimated RTT distributions are shown in Figures 4.8 and 4.10. The first figure uses the same range in the x-axis as Figure 4.7, while the second figure uses a broader range in the x-axis, between 0 and 5 seconds. The first three estimators are not affected by losses, but the RTT distribution computed by the sum-of-means estimator is substantially heavier than the original. Similarly, the distribution computed by the sum-of-maxima is several times larger than the real distribution.

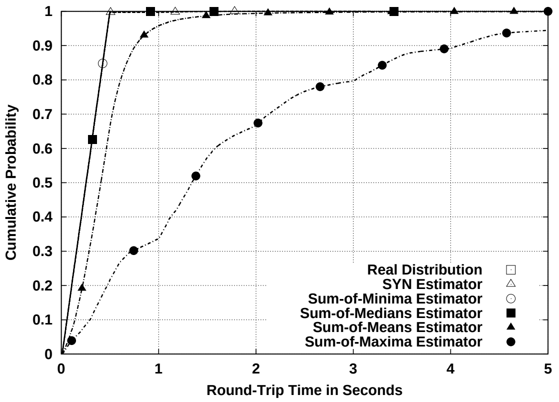


Figure 4.10: Comparison of RTT estimators for a synthetic trace: loss rates uniformly distributed between 0% and 10%.

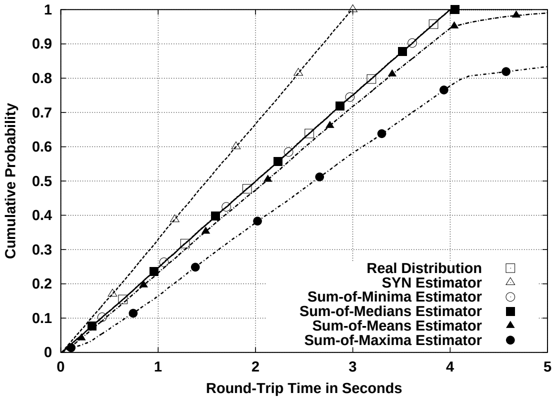


Figure 4.11: Comparison of RTT estimators for synthetic traces: fixed loss rate of 1%; real RTTs up to 4 seconds.

The cause of the additional inaccuracy in the sum-of-means estimator is the interaction between losses and TCP's cumulative acknowledgment mechanism, which prevent us from disambiguating samples from retransmissions. This problem is illustrated in Figure 4.9. Segments 1 and 2 with sequence numbers s_1 and s_2 respectively are sent from the initiator to the acceptor, but segment 1 is lost before the monitor. Since TCP's acknowledgments are cumulative, this means that the acceptor endpoint cannot acknowledge segment 2 alone[5]. Some time later, after the initiator times out, another segment with sequence number s_1 is sent from initiator to acceptor. Upon its arrival, the acceptor can send a cumulative acknowledgment with sequence number s_2. Using the timestamps of segments 2 and 4, we could compute an OSTT R_i. However, R_i is clearly not a good representative of the OSTT between the monitor and the acceptor, and therefore this sample is incorrect. The true value of the OSTT would be the difference between the timestamps of segments 3 and 4, which is much smaller than R_i. In this example, filtering samples from retransmitted sequence numbers does not help, since no retransmission was observed for s_2. In general, it is important to either filter out any sample associated with reordering (*e.g.*, segment 3 which has a lower sequence number than segment 2), or use an estimator, such as the sum-of-medians, that is robust to the distortion created by samples like R_i. Otherwise, OSTTs can be substantially overestimated, as illustrated in Figure 4.8.

Figure 4.11 reports on another experiment in which round-trip times were distributed between 10 and 4,000 milliseconds, and the underlying loss rate was 1%. Unlike the previous experiments, the SYN estimator results in a lighter distribution of round-trip times than the original one. This is due to the SYN retransmission timeout, which is set to 3 seconds [Bra89]. Connections with a round-trip

[5] Some implementations send an ACK whenever an out-of-order data segment is received, like Segment 2 in this case, but this behavior is not mandated by Internet standards. RFC 2581 [APS99] only recommends it.

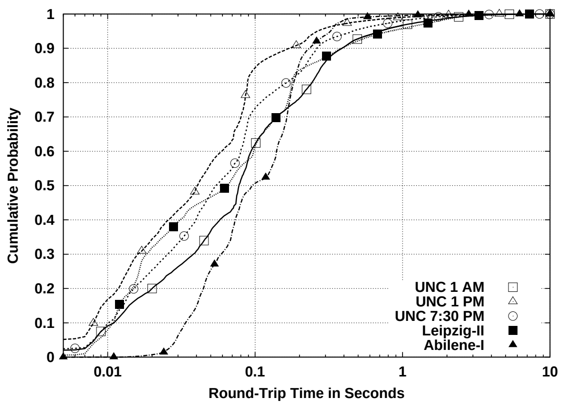


Figure 4.12: Bodies of the RTT distributions for the five traces.

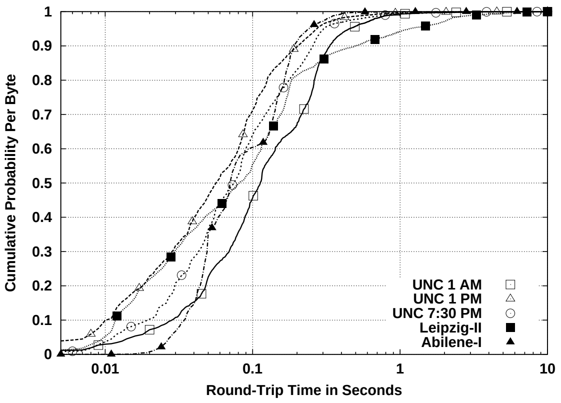


Figure 4.13: Bodies of the RTT distributions with per-byte probabilities for the five traces.

time above 3 seconds always retransmit their SYN segment, and therefore make their SYN estimator invalid. Therefore, these connections provide no samples when the SYN estimator is used, resulting in a distribution of RTTs limited to a maximum of 3 seconds. However, in these cases, the sum-of-minima and the sum-of-medians estimator were again able to estimate the distribution of round-trip times accurately.

Measurement Results

Figure 4.12 shows the distributions of round-trip times computed using the sum-of-minima estimator for the five traces listed in Table 3.1. The first observation is that the distribution of round-trip times is significantly variable across sites and for different times of the day at the same site. While the majority of round-trip times are between 7 milliseconds and 1 second for UNC and Leipzig, they are distributed in a far narrower range, between 20 milliseconds and 400 milliseconds, for Abilene-I. This is probably due to the fact that the Abilene-I trace was collected in the middle of a backbone network that mostly carries traffic between US universities and research centers so intercontinental round-trip times are very uncommon in this trace. This is also a lightly loaded network, so extra delays due to queuing are very uncommon. Note also that the distributions for UNC become lighter as we consider busier times of the day. The cause for this is an open question. The distribution for Leipzig-II does not exactly match any of the ones for UNC, but its body fluctuates within the envelope formed by the UNC distributions.

Figure 4.13 shows the same distributions but the probability of each round-trip time is computed for each byte rather than for each connection. A probability of 0.5 in this plot means that 50% of the bytes were carried in connections with a round-trip time of a given value or less. For example, for the

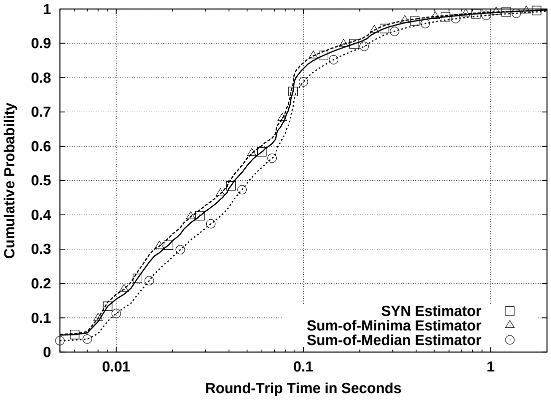


Figure 4.14: Comparison of the sum-of-minima and sum-of-medians RTT estimators for UNC 1 PM.

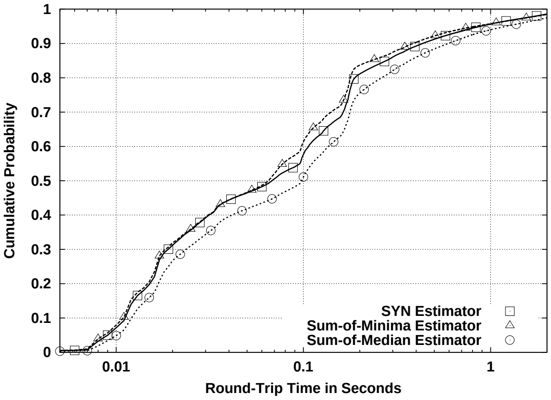


Figure 4.15: Comparison of the sum-of-minima and sum-of-medians RTT estimators for Leipzig-II.

UNC 1 AM trace, 50% of the bytes were carried in connections that experienced round-trip times of 110 milliseconds or less. Previously (*e.g.*, Figure 4.12) a probability of 0.5 meant that 50% of the connections experienced round-trip times of a given value or less. In general, we observe that the smallest round-trip times are somewhat less significant in terms of bytes than they are in terms of connections. Interestingly, Abilene-I does not differ much from the other distributions in this case. Another interesting observation is that a substantial number of bytes in the Leipzig-II traces were carried in connections with round-trip times between 300 milliseconds and 3 seconds, and this phenomenon is not observed for the other distributions. This could be explained by the location of this link in Europe, and the fact that it may carry a significant amount of traffic to distant US servers.

Figures 4.14 and 4.15 compare the variability in the results when different estimators are used for the same trace. In the UNC 1 PM trace and the Leipzig-II trace, the sum-of-medians estimator results in a somewhat heavier distribution of round-trip times but maintains more or less the same shape of the distribution. Given that these estimators were shown to be robust to losses and TCP artifacts in the previous section, the difference between the sum-of-minima and the sum-of-medians seems due to true round-trip time variability. While we are not implementing RTT variability within individual TCP connections in our experiments, it seems possible to reproduce this variability during traffic generation. This could be achieved by combining the distributions from the sum-of-minima and the sum-of-medians to give connections more variable round-trip times. For example, given a connection with a sum-of-minima estimate of $\hat{R}_{min}$ and sum-of-medians estimate of $\hat{R}_{median}$, let $\delta = \hat{R}_{median} - \hat{R}_{min}$. During traffic generation, the segments of this connection could be delayed by a random quantity between $\hat{R}_{median} - \delta$ and $\hat{R}_{median} + \delta$, or some variation of this scheme. Note that this basic method needs to be refined to eliminate segment reordering, which would occur frequently with the described approach.

4.1.2 Receiver Window Size

When a segment is received by a TCP endpoint, its payload is stored in an operating system buffer until the application uses a system call to receive the data. In order to avoid overflowing this buffer, TCP endpoints use a field in the TCP header to tell each other about the amount of free space in this buffer, and they never send more data than can possibly fit in this buffer. This mechanism, known as *flow control*, imposes a limit on the maximum throughput of a TCP connection. A sender can never send more data than the amount of free buffer space at the receiver. We refer to this free space as the receiver window size. The TCP header of each segment includes the size of the receiver window on the sender endpoint at the time the segment was sent. This value is often called the "advertised" window size, and defined as a "receiver-side limit on the amount of outstanding (*i.e.*, unacknowledged) data" by RFC 2581 [APS99]. The size of the advertised window shrinks as new data reach the endpoint (since data are placed in the TCP buffer), and grows when the application using the TCP connection consumes these data (which are removed from the TCP buffer).

A TCP connection with a maximum receiver window of W segments[6], a maximum segment size of S bytes, and a round-trip time of R seconds, can at most send data at $\frac{W*S}{R}$ bytes per second. This peak throughput can be further constrained by the capacity of the path C, so peak throughput is $min(\frac{W*S}{R}, C)$. As we will show, connections often use small receiver window sizes that significantly constrain performance, *i.e.*, $\frac{W*S}{R} << C$, and this should be taken into account during traffic generation.

We can measure the distribution of receiver window sizes by examining segment headers. As pointed out in [CHC+04b], some TCP implementations (*e.g.*, Microsoft Windows) do not report their maximum receiver window size in their first segment (*i.e.*, the SYN or SYN-ACK) as one would expect, but do it in their first data segment. This is because some implementations allocate a small amount of buffering (*e.g.*, 4 KB) to new TCP connections, but increase this amount after connection establishment is successfully completed (*e.g.*, increasing it to 32 KB). In our work, we compute the maximum receiver window sizes as the maximum value of the advertised window size observed in the segments of each TCP connection. This gives us two maximum receiver window sizes per connection, one for each endpoint. There is no reason why the two endpoints must use receiver windows of equal size.

[6] The advertised receiver window size is given in bytes in the TCP header. We describe it here and in section 4.1.1 in terms of segments for convenience when considering the impact of round-trip times.

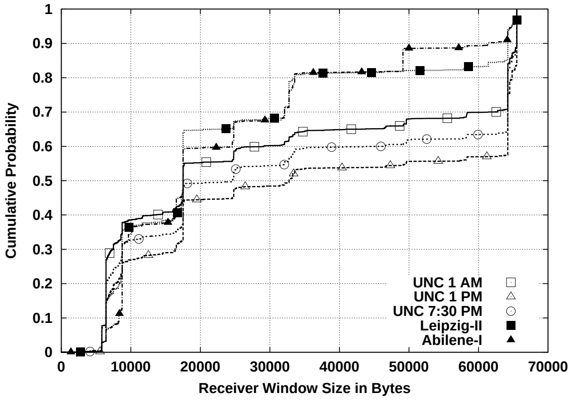


Figure 4.16: Bodies of the distributions of maximum receiver window sizes for the five traces.

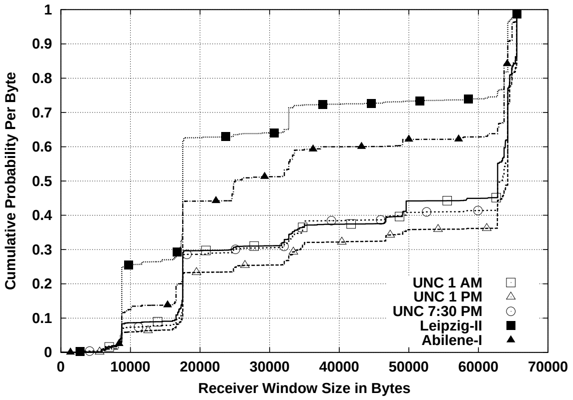


Figure 4.17: Bodies of the distributions of maximum receiver window sizes with per-byte probabilities for the five traces.

Figure 4.16 shows the distribution of maximum receiver window sizes in five traces. In general, window sizes are a multiple of the maximum segment size (usually 1,460 bytes), so we observe numerous jumps in the cumulative distribution function. Notice for example the jumps at 12 segments, $1,460*12 = 17,520$ bytes (approximately 16 KB), and 44 segments, $1,460*44 = 64,240$ bytes (approximately 64 KB). The field in the TCP header that specifies the receiver window size is 16 bits long, so the maximum receiver window size is 65,535 bytes[7].

We can make two interesting observations from Figure 4.16. First, a significant fraction of the connections used small receiver window sizes in all traces. For example, between 45% and 65% of the connections had window sizes below 20,000 bytes. Second, we observe a surprising difference between the UNC distributions and those from Leipzig-II and Abilene-I. We see a much larger fraction of the largest windows at UNC, suggesting a different distribution of endpoint TCP implementations, or widespread tuning of the servers located on the UNC campus. This is in sharp contrast to the results for round-trip times, where Leipzig-II and UNC were alike and quite different from Abilene-I.

Figure 4.17 shows an alternative view of the distributions of maximum receiver window sizes by computing the probability that each byte in the traces was carried in a connection with certain maximum receiver window size. The plot shows that connections with the largest window sizes carry many more bytes than those with small sizes. This is likely to be explained by tuning of the TCP endpoint parameters

[7] Some implementations support the window scaling option described in [JBB92], which enables larger windows. These larger windows are specified as the product of the receiver window size encoded in a 16-bit field in the TCP header, and a multiplier encoded in a TCP option (almost always 64 KB). We have not studied this feature in our work. The use of the window scaling is negotiated by the endpoints using a TCP header option, and TCP options are often not included in segment header traces, making the analysis difficult. It would however be possible to study the maximum amount of unacknowledged data in each connection, which would allow us to identify violations of the advertised window. For these cases, we could estimate the scaled window size by multiplying the advertised window by 64 KB.

by administrators and server vendors in environments with large data transfers.

4.1.3 Loss Rate

TCP reacts to loss by retransmitting segments, which makes TCP a *reliable* transport protocol, and reducing its sending rate, a mechanism known as *congestion control*. The reduction in sending rate is implemented using a TCP variable known as the congestion window size G, which further limits the maximum number of packets that can be sent by one endpoint. Throughout the lifetime of a TCP connection, TCP endpoints are only allowed to have a maximum of $min(G, W)$ outstanding (unacknowledged) segments in the network. This limits peak throughput to $min(\frac{min(G,W)*S}{R}, C)$.

The size of the congestion window is reduced every time TCP detects loss, so lossy connections have lower throughput than lossless ones. Numerous papers have developed analytical expressions that consider the impact of loss on average throughput. These papers make use of different analysis techniques and consider different models of TCP behavior and loss patterns. However, the simple relationship between loss and rate given in [MSM97] is enough to illustrate the basic impact of loss. In general, the average throughput of a TCP connection is $\frac{S*K}{R\sqrt{p}}$, where S is the maximum segment size, K is a constant equal to $\sqrt{\frac{3}{2}}$, R is the round-trip time and p is the loss rate. Therefore, average throughput is inversely proportional to the square root of the loss rate p, and it decreases very quickly as p increases. Note that the maximum window size is not part of this equation, but peak throughput is still limited by W (and by round-trip time), as mentioned above.

We define the loss rate of a TCP connection as the number of lost segments divided by the total number of segments sent, l/s. Assuming segments have an equal probability of loss, the loss rate is equal to the probability of losing an individual segment. Measuring the exact loss rate experienced by a TCP connection depends on our ability to count all segments, including those that may be lost before the monitoring point, and detecting all losses, which may occur before or after the monitoring point. The exact calculation of the loss rate of a connection is a very difficult task. In our work, we make use of two heuristics that should provide a good approximation of a connection's loss rate. We make no attempt to address the most difficult and ambiguous cases of loss detection, which our experience leads us to believe are uncommon.

Our measurement of loss rate from traces of segment headers relies on detecting retransmissions and making use of the same indications of loss that TCP employs. For each connection, we compute the total number of segments transmitted s as the total number of *data* segments in the connection. In addition, we compute the total number of lost segments l using the number of retransmitted data segments r, and the number of triple duplicate acknowledgment events d. We need both numbers r and d, since they provide complementary information. Triple duplicates can tell us about losses that occur before the monitoring point, which do not create observable retransmissions. Retransmissions can tell us about losses recovered using the retransmission timer, which do not create triple duplicates.

Estimating the loss rate p of a TCP connection simply as $(r + d)/s$ tends to overestimate loss rate when the monitoring point is located after the point of loss. In the most common situation, when the loss of a segment in one direction happens *before* the monitoring point, the trace collected at the monitoring point includes no retransmission and sends three duplicate acknowledgments in the opposite direction. These acknowledgments share the same sequence number, which corresponds to the sequence number of the segment that preceded (according to TCP's logical data order) the lost segment. However, when the loss happens *after* the monitoring point, the trace includes both a retransmission, in the direction in which the loss occurred, and a triple duplicate acknowledgment event, in the opposite direction. We can therefore compute a better estimate of loss rate by ignoring the triple duplicate events whenever a corresponding retransmission is observed. Doing so means that triple duplicates are used to estimate loss before the monitoring point, while retransmissions are used to estimate loss after the monitoring point. Applying this idea, the estimate of loss rate that we use in our work is therefore $(r + d')/s$, where d' includes only triple duplicate events not associated with observed retransmissions.

Note also that our computation of the loss rate considers only losses of data segments, and not losses of pure acknowledgments. Losses of acknowledgments can also reduce the size of the congestion window G, but measuring acknowledgment loss rate is even harder given the cumulative nature of the acknowledgment numbers and the fact that endpoints may acknowledge every data segment received, or every other data segment. We are not overly concerned by this simplification. This is because under the assumption that losses are caused by congestion, pure acknowledgments are far less likely to be dropped given their much smaller size.

In order to study the accuracy of our estimation of loss rates, we conduct a number of controlled experiments similar to those used to evaluate the different round-trip time estimation techniques. Figure 4.18 shows the results of two laboratory experiments in which an artificial loss rate of 1% was imposed on connections carrying a single epoch with $a_1 = b_1 = 10,000,000$ bytes. Transferring ADUs of this size requires a minimum of 6,850 data segments. Losses were created using *dummynet* , so each connection

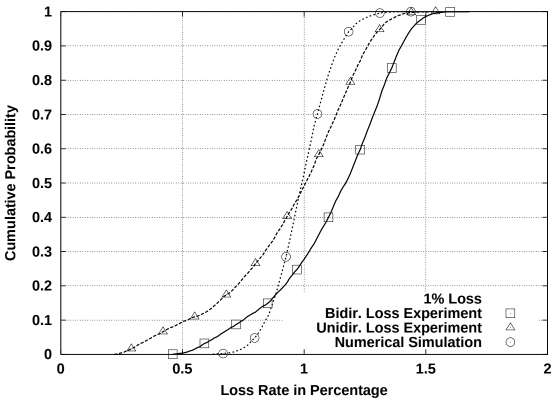


Figure 4.18: Measured loss rates from experiments with 1% loss rates applied only on one direction or on both directions of the TCP connections.

in the experiment had a drop probability equal to 0.01. The figure illustrates several points about our loss rate computation. We first compare the measured loss rates for two scenarios: one where a segment loss probability of 0.01 was applied by *dummynet* only to one direction of the connections, and another one where it was applied to both directions. In Figure 4.18, the first scenario is labeled "unidirectional loss experiment" and the second is labeled "bidirectional loss experiment". The mean value of the data segment loss rate in the unidirectional loss experiment (marked with white triangles) was 1%, exactly the intended value. We also observe that 90% of the connections experienced loss rates between 0.5% and 1.5%. The bidirectional loss experiment (results marked with white squares) illustrates the dependency between the two directions of a TCP connection. The mean of the CDF is substantially higher for this experiment, and the distribution shows a fixed positive offset of 20%. This is because losses of acknowledgments in one direction also triggered retransmissions in the other, increasing the measured (data segment) loss rate. In other words, loss of acknowledgments inflated the estimated loss rates, since data was not really lost.

Our second observation about Figure 4.18 is that the range of the two distributions is quite wide, showing substantial variability around the target loss rate of 1%. This is partly explained by the random sampling in *dummynet* 's implementation of per-flow loss rates. *Dummynet* drops segments in an independent manner, by generating a random number between 0 and 1 for each segment, and only dropping a segment if its corresponding random number is between 0 and 0.01. This means that even with large ADUs, the drop probability rate experienced by the connection in the testbed experiments was not exactly 0.01.

In order to study the impact of this random sampling, we conduct a numerical simulation, and the result is illustrated using the third CDF in Figure 4.18. This distribution comes from simulating each connection by sampling a uniform distribution (with a range between 0 and 1) 6,850 times (the number of data segments in 10 MB). Each sample is meant to simulate one segment that may or may not be lost. If the value of the sample is equal to or greater than 0.01, the segment is not counted as a loss. If the sampled value is less than 0.01, the segment is counted as a loss. In this case, we continue to sample the uniform distribution until the value obtained is equal to or greater than 0.01. These extra samples are used to simulate the possibility of losing retransmissions, which can also be dropped by *dummynet* with the same probability. The result of this sampling process is two counts:

- the total number of segments s^* in the simulated connection, which is the number of times that the uniform distribution was sampled, and
- the total number of loss events l^*, which is the number of times that the samples from the uniform distribution were less than 0.01.

The ratio l^*/s^* is the simulated loss rate p^* for one connection. We repeated this process 4,200 times, which was the number of connections in the testbed experiments, and constructed a CDF of the resulting loss rates which is shown in Figure 4.18. The CDF exhibits substantial variability around 1%. Therefore, sampling variability partially explains the variability observed in the loss rates that we measured from the testbed experiments. Note that the variability in our lab experiments and in the numerical simulation is tied to the size of the ADU (10 MB in a minimum of 6,850 segments). Increasing this size would reduce the variability of the measured loss rates in proportion to the square root of the number of segments (a basic probability result for sample means). However, our illustration of the sampling variability using 10 MB ADUs is already conservative, since most TCP connections carry far less data and therefore need fewer segments.

The CDF from the numerical simulation provides us with a gold standard for our measurements, since our loss rate estimates should reflect the actual drop rates that *dummynet* imposed to the connections in the testbed. Still, further work is need to explain the remaining difference, and possibly refine our measurement technique. In any case, the experiments serve to confirm that our loss rate estimate is reasonably close to the true loss.

The distributions of loss rates in our collection of real traces is shown in Figure 4.19. Between 92.5% and 96.2% of the connections did not experience any losses, while the remaining connections did experience quite significant loss rates. This is consistent across all measured sites. The result is quite

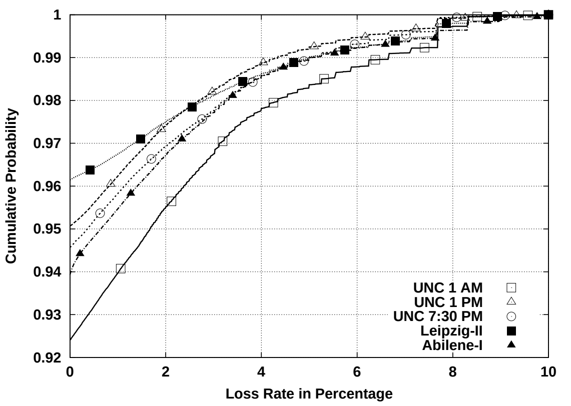


Figure 4.19: Bodies of the distributions of loss rates for the five traces.

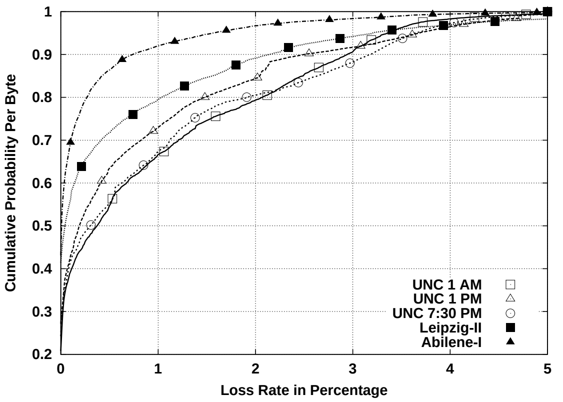


Figure 4.20: Bodies of the distributions of loss rates with per-byte probabilities for the five traces.

different when the probability is computed in terms of bytes rather than in terms of connections, as shown in Figure 4.20. Most bytes were carried in connections that experience at least some loss. We can also observe that Abilene-I's connections were substantially less affected by loss than the connections in the other traces, since Abilene-I's CDF shows higher cumulative probabilities. For example, the distribution for the Abilene-I trace shows that only 8% of the bytes were carried in connections with 1% loss or more. The distributions for the rest of the traces show that between 20% and 34% of the bytes were carried in connections with 1% loss or more. Loss is specially significant at UNC, where 34% of the bytes were carried by connections with loss rates above 1% (a high loss rate). Interestingly, the UNC trace with the highest load (UNC 1 PM) had a lighter distribution of per-byte loss rates.

4.2 Network-level Metrics

The previous section considered methods for characterizing network-level properties of traffic that can be incorporated into traffic generators as input parameters. Here we consider other network-level properties that can be used to compare traces, providing a way to assess the realism of synthetic traffic. In order to make the distinction between these two types of network-level properties clearer, we apply the term *network-level parameter* to those properties that are part of the input of the traffic generation method, and the term *network-level metric* to those properties that are not part of the input but are still useful for characterizing the output (*i.e.*, the synthetic traffic). The key idea, demonstrated in Chapter 6, is that synthetic traffic can closely approximate real traffic in terms of these network-level metrics, as long as source-level and network-level parameters are incorporated into the traffic generation method.

The success of this approach confirms that the parameters we have incorporated in our approach are significant, and that the data acquisition methods we propose are sufficiently accurate to achieve high realism in traffic generation.

4.2.1 Aggregate Throughput Time Series

A basic property of the performance of a network link is the number of bytes and packets[8] that traverse the link per unit time. We will call this property aggregate throughput, since it is the result of multiplexing the throughputs of the individual connections that form the traffic carried by a network link. Accurately reproducing aggregate throughput will be an important part of our evaluation.

Aggregate throughput is generally very variable, so researchers (and practitioners) usually study the time series of aggregate throughputs in order to understand the dynamics of network traffic. Formally, an aggregate throughput time series at scale t is defined as a vector $X^t = (X_1^t, X_2^t, \ldots, X_n^t)$ where X_i^t is the number of bytes (or packets) observed at a measurement point between time $t(i-1)$ and time ti for some constant interval t. This constant integral t is called the *scale* of the time series. X_i is often referred to as the i-th bin of the time series, which is sometimes called a time series of bin counts.

We consider three ways of studying aggregate throughput time series in this dissertation. First, we make use of plots of aggregate throughput against time, "throughput plots", which provide a simple yet informative visualization of the dynamics of the traffic throughout the entire trace. Second, we examine the marginal distribution of the time series using a CDF, which enables us to study the fine scale characteristics of the throughput process. These two methods are described in more detail below. While they are useful, they are sensitive to the scale t at which the time series is analyzed (*e.g.*, throughput per minute is much smoother than throughput per millisecond). For this reason, we complement our analysis with a third method, wavelet analysis. Wavelet analysis is a multi-resolution method particularly suitable to study how the statistical nature of X^t changes as a function of t. This type of study, often known as "traffic scaling", is specially important for Internet traffic, which exhibits strong long-range dependence. We employ both plots of the wavelet spectrum of a throughput time series, and wavelet-based estimates of Hurst parameters with confidence intervals.

When it comes time to validate synthetic traffic generation methods, an important aspect of the

[8] In this section, we will often use the term packet rather than segment. In the context of TCP traffic, a time series of packets per unit time and a time series of segments per unit time are the same thing. However, the traffic measurement literature generally talks about packet throughput (not segment throughput), often using the unit Kilo packet per second (Kpps).

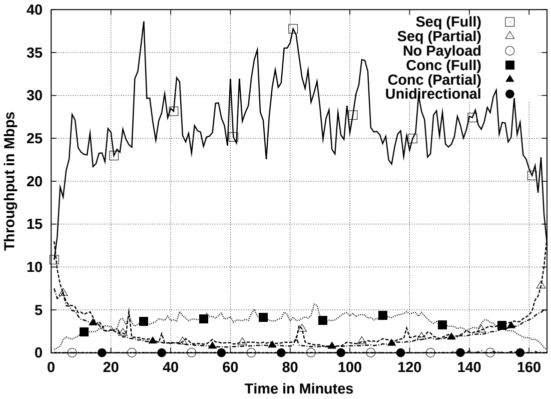


Figure 4.21: Breakdown of the byte throughput time series for Leipzig-II inbound.

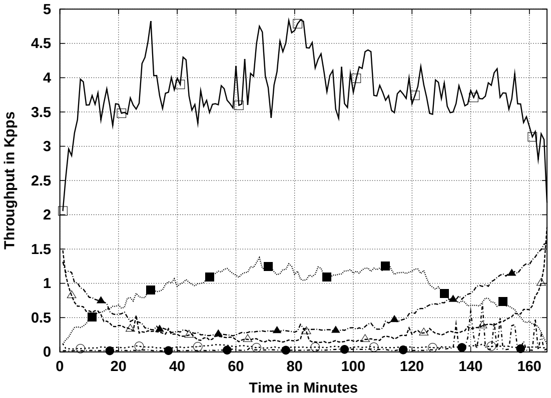


Figure 4.22: Breakdown of the packet throughput time series for Leipzig-II inbound.

validation will be a qualitative comparison of plots of throughput time series and plots of their marginal distributions and wavelet spectra. Here we describe the use of these visualizations to understand the nature of throughput on the links we have measured.

Throughput Plots

Figure 4.21 shows a breakdown of the aggregate byte throughput of the Leipzig-II trace in the inbound direction (*i.e.*, TCP traffic coming into the University of Leipzig). The scale of the time series (the bin size) is one minute. The time series of all byte arrivals has been partitioned into six time series according to the type of abstract source-level behavior (sequential, concurrent or no payload[9]), depending on whether the start and the end of the connection were observed (fully or partially captured connections), and whether the connection was observed only in one direction of the link (unidirectional connections) or in both. The analysis of abstract source-level behavior described in Section 3.5 was used to classify connections into these categories, and then the original segment header traces were partitioned according to this classification. This type of analysis complements the one performed at the source-level in Section 3.5, giving us a sense of the relative importance of sequential and concurrent connections. Also, our traffic generation will only make use of connections that were fully captured, *i.e.*, fully characterized, so it is important to understand the importance of the traffic in the rest of the connections (so we know what we are missing).

Figure 4.21 shows that sequential connections that were fully captured account for the vast majority of the bytes to Leipzig-II inbound. Since connections observed near the boundaries of the trace are more

[9] A connection without any useful TCP payloads has an empty connection vector since no ADU is sent.

likely to be observed only partially, the time series shows a much smaller number of bytes in the first and in the last ten minutes of the trace. On the contrary, the time series of partially-captured sequential connections has a much larger number of bytes in the first and the last ten minutes. This is because the probability of observing only part of a connection increases as we get closer to the trace boundaries. For this reason, in the first ten minutes we see many more connections that started before the start of the trace, and in the last ten minutes we see many more connections that ended after the end of the trace. We will refer to this increased likelihood of finding partially-captured connections near trace boundaries as the *connection sampling bias*.

The solid line with white squares in Figure 4.21 shows the time series of fully-captured sequential connection. When we examine the stable region of this time series (*i.e.*, ignoring the first and last 10 minutes), we can see substantial variability between the minimum of 22 Mbps and the maximum of 38 Mbps. The rest of the time series in this plot are far less "bursty". The average throughput of the time series for concurrent connections is much smaller, and partially-captured connections only account for a tiny fraction of the bytes. The number of bytes in connections without any useful data payload is insignificant, as one would expect in a properly working network in which little malicious activity is taking place[10].

Figure 4.22 shows the time series of packet arrivals for the inbound direction of the Leipzig-II trace. As in the previous figure, fully-captured sequential connections account for the majority of the packets, and the time series exhibits substantial variability. Notice however that the number of packets in fully-captured concurrent connections is more significant in terms of packets than in terms of bytes (the percentage of packets was higher than the percentage of bytes). The time series of the number of packets in "no payload" connections and in unidirectional connections is also more significant. Notice the large spikes at the end of the time series of unidirectional connections. These spikes could be related to some malicious activity, like network or port scanning[11], or connection attempts to a popular server that is temporarily offline[12]. This feature was not present in the corresponding time series of byte arrivals.

The same time series for the reverse direction of the Leipzig link are shown in Figures 4.23 and 4.24. The magnitude of the throughput time series is significantly smaller in this case, suggesting that the

[10] The "no payload" time series would have been much more significant if, for example, a denial-of-service attack using SYN segments had taken place. These segments, and the likely SYN-ACK segments sent in response by the victim, would have not carried any (useful) payloads (no application-level communication would have taken place), and would have been classified as "no payload" traffic.

[11] This type of activity creates unidirectional traffic whenever the target host is firewalled, or otherwise unreachable, or the target IP does not exist. The location of these spikes at the end of the trace is purely accidental.

[12] In this case, clients would try to open a connection by sending a SYN segment (and several retransmissions), which will receive no response since the destination server is not online. These types of connection attempts to offline hosts show up as unidirectional connections in segment header traces.

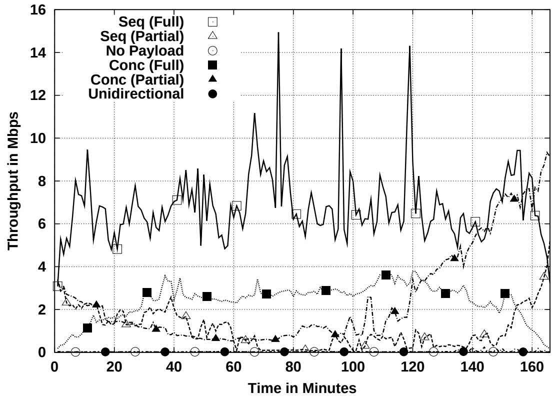


Figure 4.23: Breakdown of the byte throughput time series for Leipzig-II outbound.

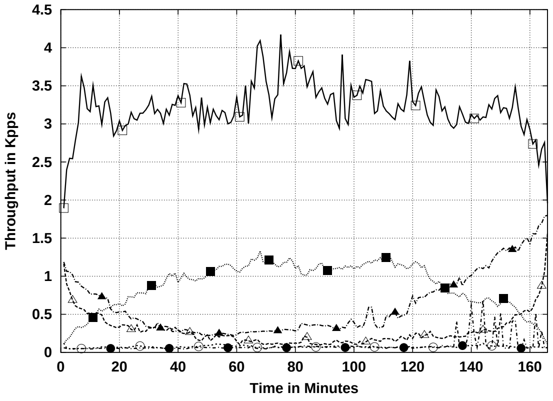


Figure 4.24: Breakdown of the packet throughput time series for Leipzig-II outbound.

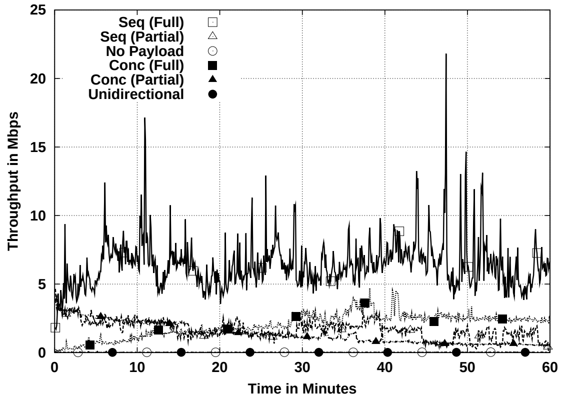


Figure 4.25: Breakdown of the byte throughput time series for Leipzig-II outbound.

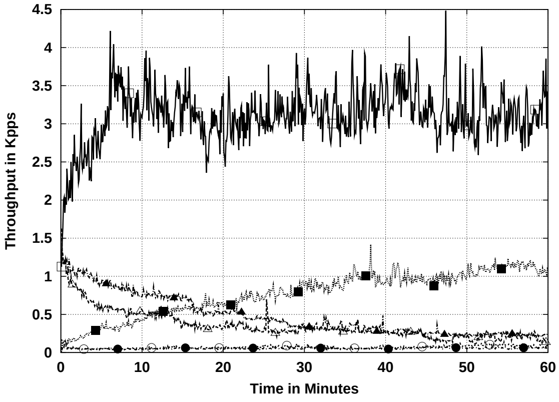


Figure 4.26: Breakdown of the packet throughput time series for Leipzig-II outbound.

University of Leipzig is mostly a consumer of content from the rest of the Internet. Sequential connections that were fully captured exhibit some sharp spikes in three short time intervals. A closer look revealed that only a few large connections with small round-trip times (in particular, three connections in the first spike in minute 78) created this sudden increase in throughput. As in the inbound direction, partially-captured sequential connections are only significant for the first and the last few minutes of the trace. Concurrent connections also show the same pattern, but partially-captured connections exhibit a steady increase for the last 50 minutes of the trace. Interestingly, the separation between the time series for fully and partially captured sequential connections is much larger for packets than for bytes. This suggests that the packets in this direction are very small, and mostly consist of acknowledgment segments that do not have a payload. This is another confirmation of the content-consumer nature of the university of Leipzig.

Figures 4.25 and 4.26 illustrate the impact of scale on the throughput time series for the Leipzig-

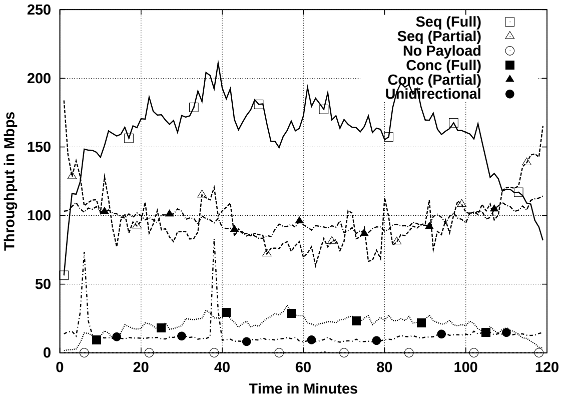


Figure 4.27: Breakdown of the byte throughput time series for Abilene-I Ipls/Clev.

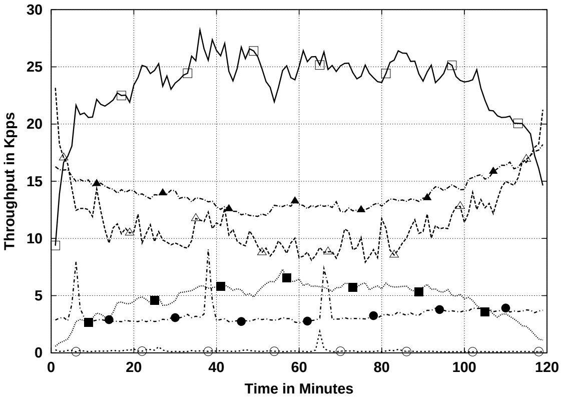
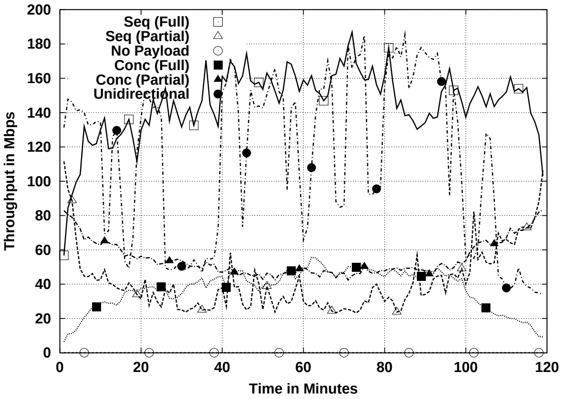


Figure 4.28: Breakdown of the packet throughput time series for Abilene-I Ipls/Clev.

II outbound trace. These plots have a scale of 5 seconds, and only the first 60 minutes (rather than entire 166 minutes) are shown to reduce the amount of over-plotting. Both byte and packet throughputs are clearly more bursty at this scale. The largest spikes of time series for fully-captured sequential connections are even larger (and therefore narrower) than those in the 1-minute time series. For example, the spike in the eleventh minute reaches 17 Mbps in the 5-second scale, while the corresponding region in the 1-minute scale plot in Figure 4.23 did not go above 9 Mbps. Decreasing the scale provides a better picture of the burstiness of traffic, but it increases over-plotting, and does not change the overall view (*i.e.*, the relative magnitude of the different time series). The same lesson holds for packet arrivals, but notice that the largest byte throughput spikes do not appear to have corresponding packet throughput spikes. This shows that a relatively small number of full packets sent in short periods created the observed throughput spikes (and not a large number of small packets).

The structure of the throughput time series for the Abilene-I trace is remarkably different. Figure 4.27 shows the time series of byte arrivals at a 1-minute scale for the Abilene-I traffic sent from Indianapolis to Cleveland. As in the Leipzig-II case, fully-captured sequential connections account for the largest percentage of the traffic. However, bytes from partially-captured sequential connections are much more significant here, with a mean throughput that is roughly half of the mean throughput for fully-captured sequential connections. While we still observe much larger throughputs in the first and last few minutes of the time series, the middle part still accounts for a very large number of bytes. This is in sharp contrast to the Leipzig-II trace, and cannot be explained by the duration of the trace, which is almost as long (2 hours *vs.* 2 hours and 46 minutes). Note also that the partially-captured connection time series is almost as bursty as the time series for fully-captured connections.



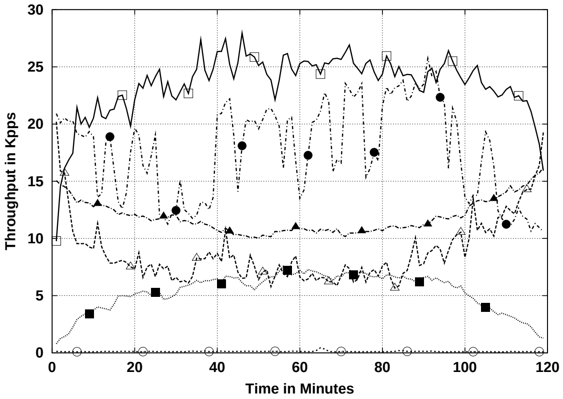


Figure 4.29: Breakdown of the byte throughput time series for Abilene-I Clev/Ipls.

Figure 4.30: Breakdown of the packet throughput time series for Abilene-I Clev/Ipls.

Concurrent connections in this direction of the Abilene-I trace show a surprising structure. The number of bytes in concurrent connections that were partially-captured was much larger than the number of bytes in connections that were fully-captured. This suggests that concurrent connections in this trace tend to have extremely long durations. Both time series are much smoother than those for sequential connections, and trace boundaries have very little impact on them. Connections with no payload carried an insignificant number of bytes, but, unlike the Leipzig-II trace, unidirectional traffic is non-negligible. Rather than some malfunction or malicious activity, this is explained by asymmetric routing in the Abilene backbone. Only one direction of these connections goes through the measured link, and hence these connections appear in our trace as unidirectional. We also observe two major throughput spikes at the 6th and the 38th minutes that could also be explained by transient routing changes, but malicious traffic cannot be ruled out without further analysis. Both spikes reach throughputs as high as 350 Mbps when the time series is examined at the 5-second scale.

The packet throughput time series for the Abilene-I trace shown in Figure 4.28 has a similar structure, in which partially-captured connections also account for a large percentage of the trace. It is interesting to note that fully-captured concurrent connections carry a larger percentage of packets than bytes, so packets in these connections are likely to be small. We also observe a third spike in the time series for unidirectional connections that did not show up in the byte throughput time series, and a smaller spike in the "no payload" time series.

The reverse direction, Cleveland to Indianapolis, of the Abilene-I trace offers a rather different view in Figure 4.29. Partially-captured sequential connections are much less significant in this case, although this time series still exhibits remarkable variability. Similarly, the number of bytes in partially-

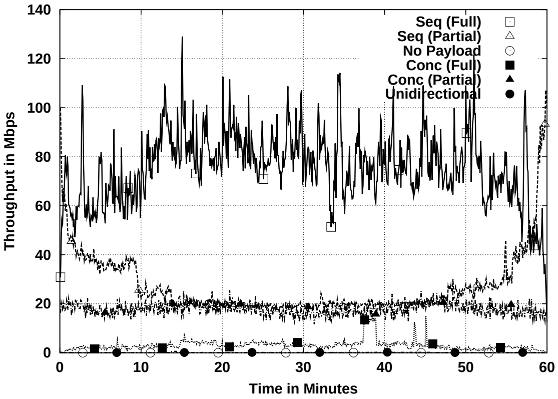


Figure 4.31: Breakdown of the byte throughput time series for UNC 1 PM inbound.

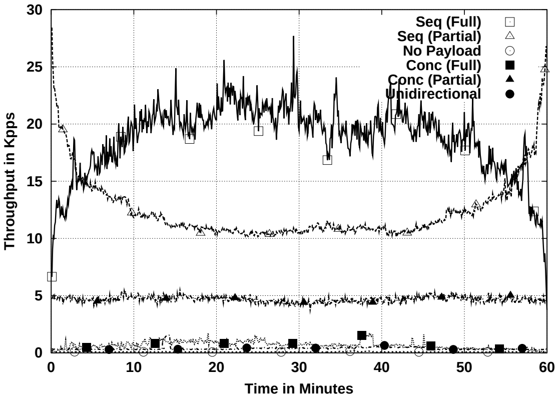


Figure 4.32: Breakdown of the packet throughput time series for UNC 1 PM inbound.

captured concurrent connections is much lower in relative terms, and quite close to the number of bytes in fully-captured concurrent connections. The most striking feature of this plot is the time series of unidirectional connections. The byte throughput of these connections shows enormous variability, and even reaches the magnitude of fully-captured sequential connections. This is either a strong indication of substantial instability in the routing of the Abilene backbone, or the existence of flows with extremely high throughput that only show up in one direction of the measured link. In any case, byte throughput is always above 50 Mbps, so part of the traffic that is routed asymmetrically did not experience any major routing changes.

The packet throughput time series from Cleveland to Indianapolis shown in Figure 4.30 offer yet another pattern in the breakdown of traffic per connection type. The sharp changes in the throughput of the time series of unidirectional traffic have a smaller magnitude, suggesting that large packets dominate traffic in this direction of the Abilene-I trace. Partially-captured concurrent connections carried significantly more packets than fully-captured connections. This is the opposite of the phenomenon in Figure 4.28 and can be explained by an asymmetry in the sizes of the ADUs of the connections. This asymmetry results from connections with a majority of data segments in the same direction and a majority of acknowledgments in the other direction.

The byte throughput time series for the UNC 1 PM trace in the inbound direction resembles that of Leipzig-II (which is also an edge trace). Figure 4.31 shows that fully-captured sequential connections carry the vast majority of the bytes, although the relative percentage of bytes in partially captured connections is larger. This is can be explained by the shorter duration of this trace (1 hour). The most significant difference, however, is in the time series for partially-captured concurrent connections.

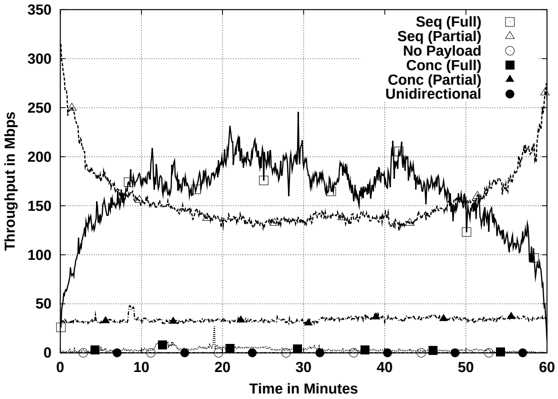


Figure 4.33: Breakdown of the byte throughput time series for UNC 1 PM outbound.

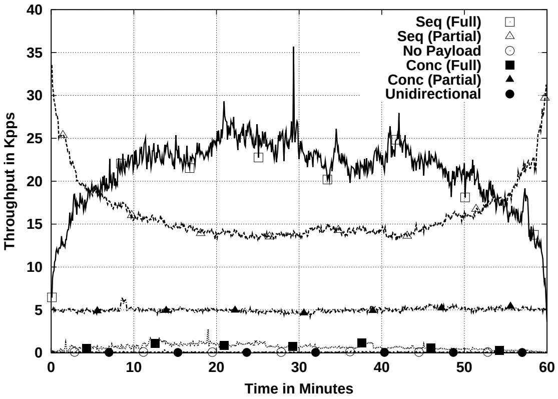


Figure 4.34: Breakdown of the packet throughput time series for UNC 1 PM outbound.

In this case, we find a very stable throughput of 20 Mbps without any clear boundary effects. This is similar to the type of concurrent traffic found in Abilene-I. Fully-captured concurrent connections show an interesting jump between the 37th and the 39th minutes, and a couple of spikes around the 45th minute. This could be explained by a single connection with significant throughput (10 Mbps). Packet throughput time series are similar, but we observe a significantly higher percentage of packets in partially-captured sequential connections. As the analysis of the other direction will suggest, this is due to the presence of many pure acknowledgment segments.

The byte throughput time series for the outbound direction of UNC 1 PM are shown in Figure 4.33. They are remarkably different from those of the Leipzig-II trace, where throughput on the inbound direction (created by local users downloading content from the Internet) was much higher than the throughput in the outbound direction. We observe the opposite here. The mean overall utilization in the outbound direction is much higher than the inbound direction, 325 Mbps versus 100 Mbps. Also, partially-captured sequential connections are much more significant. The obvious explanation is the presence at UNC of `ibiblio.org`, a popular repository of software and other content. Hosts outside UNC retrieve large amounts of data from the `ibiblio.org` servers, making the load in the outbound direction of the UNC link much higher than the load on the inbound link. Furthermore, `ibiblio.org` clients often download large objects, and this requires long connections that are more likely to be only partially captured. This provides a good explanation for the extreme boundary effects in the first and last 10 minutes of the throughput time series. The high throughput in the stable region of this time series could be due to long connections that carry large amounts of data, although further analysis is needed to verify this claim.

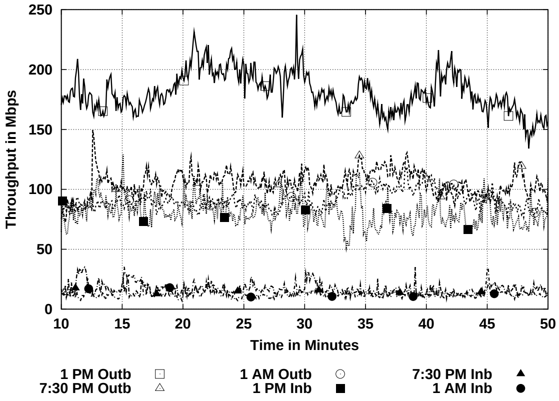


Figure 4.35: Breakdown of the byte throughput time series for the three UNC traces.

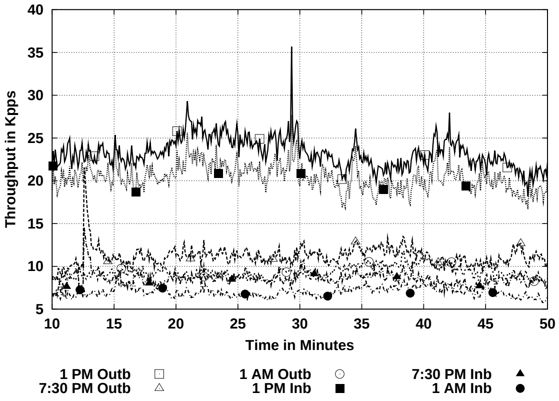


Figure 4.36: Breakdown of the packet throughput time series for the three UNC traces.

Concurrent traffic in the outbound direction appears similar to the inbound direction, showing remarkable load symmetry for partially-captured concurrent connections. We do not observe much variation in the time series for packet throughput (shown in Figure 4.34). Partially-captured sequential connections carried a smaller number of packets than bytes, and this agrees with the idea that large numbers of bytes are downloaded from `ibiblio.org`. These downloads show up as large data packets in the outbound direction and small acknowledgment packets in the inbound direction. In the most common case, a full TCP segment has a size of 1500 bytes, while an empty one (no payload) is only 40 bytes. This means that the ratio of bytes in a connection carrying a large file is 1500:40. Furthermore, since most TCP implementations acknowledge only every other data segment, we have a ratio of 3000:40 for bytes and a ratio of 2:1 for packets. A link that is dominated by large file downloads should show similar byte and packet ratios. If large file downloads from `ibiblio.org` were the only cause of the large fraction of bytes in partially captured connection, then we would expect similar ratios between the two directions of the UNC link. However, this is not so clear in Figures 4.31 to 4.34, suggesting that phenomena other than `ibiblio.org` also contribute to making UNC a source rather than a sink of content. As an example, file-sharing activity from campus dorms could also play a significant role. Since file-sharing implies both uploading and downloading, it usually tends to make the byte and packet ratios more balanced.

Network activity usually follows a cyclic daily pattern, in which traffic increases throughout the morning and decreases in the evening, being at its lowest during night hours. This diurnal pattern in the time series for the UNC traces is evident in Figures 4.35 and 4.36. These time series correspond to fully-captured sequential connections. Byte throughputs for the 1 PM traces are much higher in both directions and we observe that even the 1 AM trace has a large throughput in the outbound direction.

This suggests that content from UNC is downloaded throughout the day, although a diurnal pattern is still present. On the contrary, UNC clients are much less active later in the day. A similar plot (not shown) for partially-captured concurrent connections shows little reduction in throughput between the 1 PM and the 7:30 PM traces, and a reduction of only 15 Mbps in the 1 AM trace. The packet throughput time series illustrate again the dichotomy between the large data segments in the outbound direction, carrying UNC content, and the small segments in the inbound direction, carrying acknowledgments. The difference is substantially less significant later in the day.

4.2.2 Throughput Marginals

Plots of throughput time series provide a good overview of the coarse-scale characteristics of the traffic trace. However, they are not very practical for studying finer-scale features. Our traces are long enough that any plot at a scale of 1 second or below is dominated by over-plotting, and does not provide any useful information. This is specially true when the goal of the plot is to compare two time series that are already rather similar. Finer-scale differences can be of great importance for certain experiments. For example, two traces could have exactly the same average throughput, and appear identical at the 1-minute scale, but be completely different at the 1-second scale. One of them could show a sequence of sharp spikes and ditches, while the other one could remain completely smooth. If we were to expose a router queue to these two traces, we could obtain two completely different distributions of router queue length (and therefore of packet delay through the router). This would for example be the case if the spikes exceed the output rate of the queue (creating a backlog), while the smooth trace always remains below output rate. In the first case, packets would experience variable queuing delay, while in the second case no queuing delay would occur.

There are several ways in which we can compare traces at finer time scales. The most obvious one is to examine throughput for a limited period. While this approach is useful in some situations, it does not scale for comparing entire traces, especially as we decrease the scale and the number of possible periods to examine grows. In this dissertation, we will make use of two alternative methods for studying the throughput of our traces at finer granularities. Our first method is to examine the marginal distributions of throughput time series at a rather fine time scale, 10 milliseconds. Plots of the bodies of marginal distributions help us to understand the most common fine-scale throughputs in a trace, while plots of the tails of marginal distributions explore the episodes of highest throughput in a trace. We describe this type of analysis in the rest of this section. Our second method is to study the way throughput

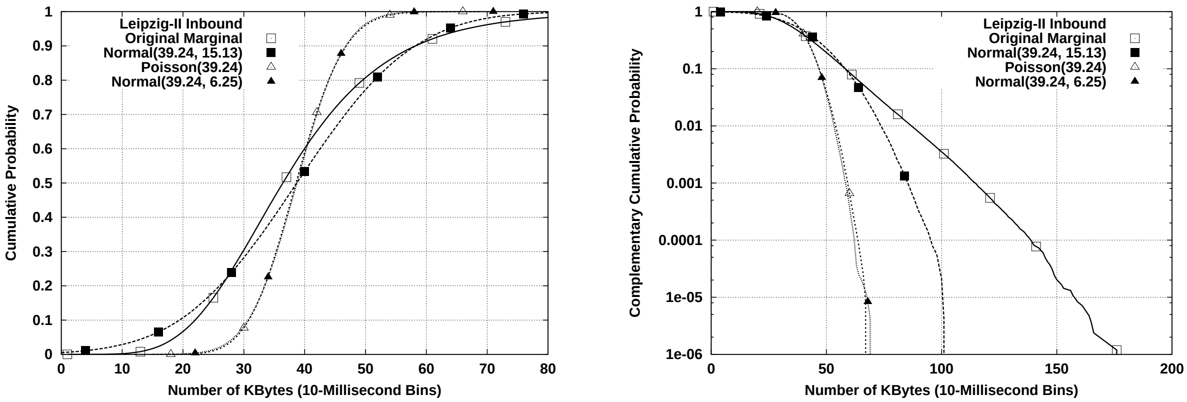


Figure 4.37: Byte throughput marginals of Leipzig-II inbound, its normal distribution fit, the marginal distribution of its Poisson arrival fit, and the normal distribution fit of this Poisson arrival fit.

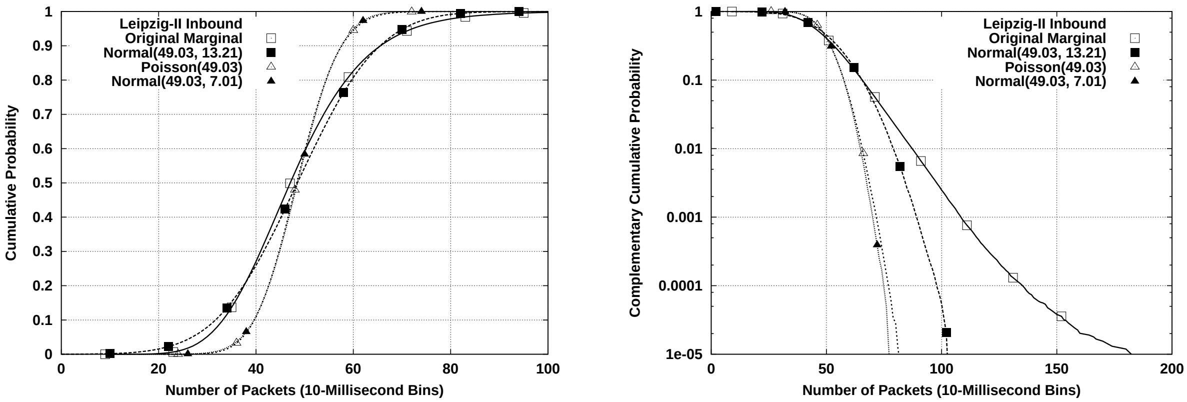


Figure 4.38: Packet throughput marginals of Leipzig-II inbound, its normal distribution fit, the marginal distribution of its Poisson arrival fit, and the normal distribution fit of this Poisson arrival fit.

variance changes with scale, which we will approach using the concepts of self-similarity and long-range dependence. We discuss this type of analysis in the next section.

Our analysis of throughput marginals examines the time series of throughput at the 10-millisecond time-scale, constructing the empirical distribution of the values of the time series. This empirical distribution assigns a certain probability to each observed value of the time series equal to the fraction that this value represents of the total set of values in the time series. As in previous cases, we will study the bodies of the marginal distributions using plots of CDFs and the tails using plots of CCDFs. For example, Figures 4.37 and 4.38 show respectively the marginal distribution of byte and packet throughput for the inbound direction of the Leipzig-II trace (depicted using solid lines marked by white squares). The CDF in the left plot provides a good overview of the body of the marginal distribution using linear axes. The CCDF in the right plot shows the tail of the distribution using a logarithmic y-axis. These

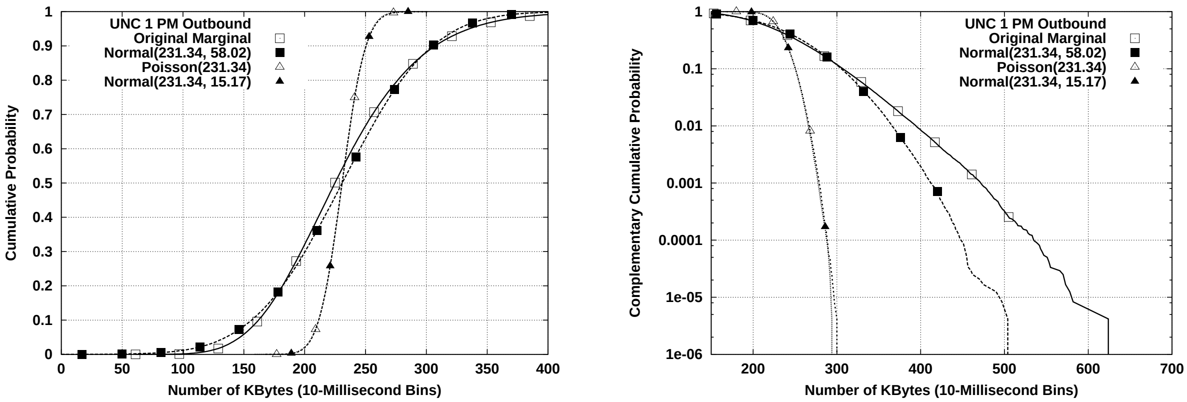


Figure 4.39: Byte throughput marginals of UNC 1 PM outbound, its normal distribution fit, the marginal distribution of its Poisson arrival fit, and the normal distribution fit of this Poisson arrival fit.

visualizations of the marginals are particularly useful for comparing multiple distributions, and we will use them extensively in Chapter 6.

As stated before, our goal with the analysis of marginal distributions is to understand fine-scale characteristics of throughput. We will use this type of analysis to determine whether our proposed traffic generation method results in synthetic traffic whose distribution of fine-scale throughputs is "realistic". By construction, and as explained in Chapter 5, the determination of this realism is accomplished by directly comparing the marginal distributions of an original trace and its synthetic version. This non-parametric analysis is consistent with other methods used in this dissertation.

We have also considered doing some parametric analysis of the marginal distributions of throughput time series. When modeling an arrival process, the first approach that comes to mind is the Poisson modeling framework. Poisson arrivals are very convenient from an analytical perspective, and concisely describe an arrival process using a single parameter. As pointed out by Floyd and Paxson [PF95], empirical studies do not support the use of this model, primarily because Poisson arrivals are far less "bursty" than Internet packet and byte arrivals. This important issue is discussed in the next section. In addition, we show here that Poisson arrivals have marginal distributions that are very far from the ones in our traces.

Given a throughput time series, we can fit a Poisson arrival model simply by computing the mean of the time series and using it as the rate of the Poisson model. From this fitted model, we can easily obtain a marginal distribution using Monte Carlo simulation. Figure 4.37 shows the marginal distribution of byte throughput in the inbound direction of the Leipzig-II trace, and the marginal distribution of the Poisson fit of this throughput process (depicted using a dashed line with white triangles). Both marginal

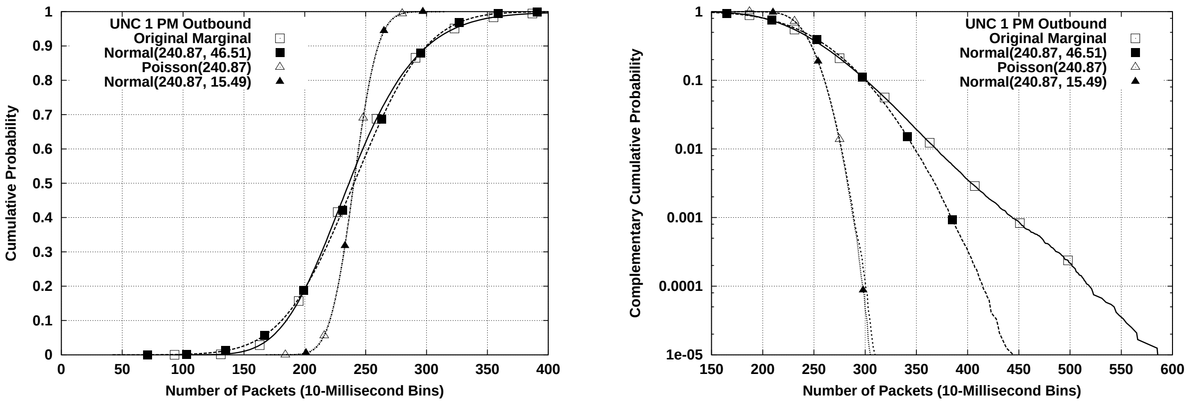


Figure 4.40: Packet throughput marginals of UNC 1 PM outbound, its normal distribution fit, the marginal distribution of its Poisson arrival fit, and the normal distribution fit of this Poisson arrival fit.

distributions have the same mean, 39.24 Kilobytes per 10-millisecond interval. As shown in the figure, the two marginals are very different, with the Poisson fit exhibiting a far narrower body. The standard deviation of the Poisson model is only 6.25, while the one for the real marginal distribution is 15.13, more than twice as large. In addition, the tail of the marginal distribution from Poisson arrivals is far lighter than the one from the trace. Intuitively, this means that the real traffic is far more aggressive on the network, consistently reaching far higher throughput values. Poisson arrivals are equally inadequate for modeling packet arrivals, at least in terms of their marginal distributions, as shown in Figure 4.38. Figures 4.39 and 4.40 repeat the same analysis for the outbound direction of UNC 1 PM. The plots confirm the poor fit from the Poisson arrival model, even for a trace with a throughput that is eight times higher. The same is true for every other trace examined in this dissertation.

The empirical results in Fraleigh *et al.* [FTD03] and the analysis in Appenzeller *et al.* [AKM04] support the idea that throughput values are normally distributed in Internet traffic, as long as sufficient traffic aggregation exists. If this were true, studying (and comparing) marginal distributions of throughput could easily be accomplished by looking at means and variances. Our analysis of the throughput marginals in our traces shows that they do resemble a normal distribution, but that this model is not completely satisfactory.

We can easily fit a normal model to the marginal distributions from our traces by computing their means and standard deviations. Figures 4.37, 4.38, 4.39 and 4.40 compare the real marginals and their fits, consistently showing the following two differences:

- The bodies of the marginal distributions from the real traces appear somewhat narrower. The

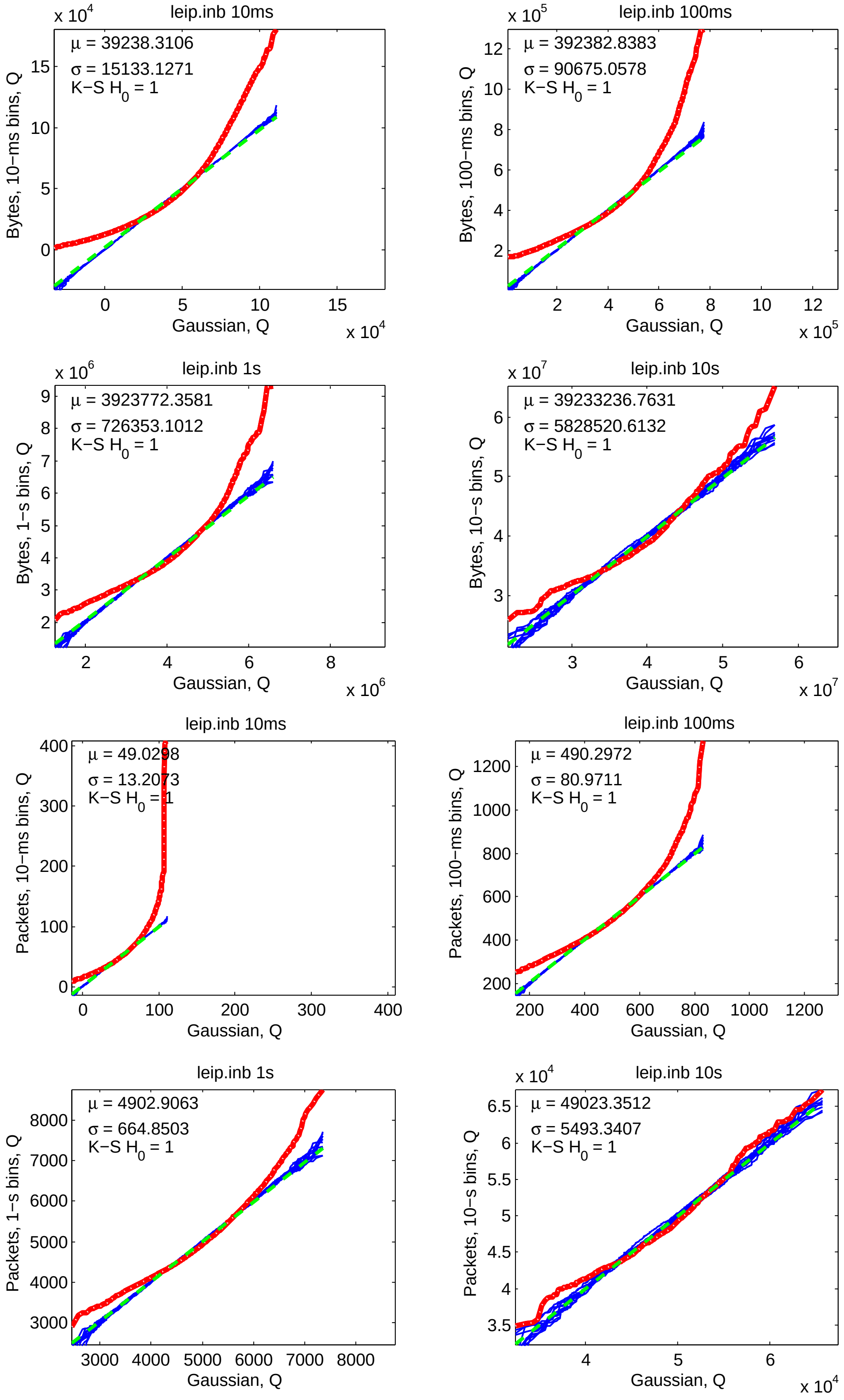


Figure 4.41: Quantile-quantile plots with simulation envelops for the marginal distribution of Leipzig-II inbound. The top four plots show byte throughput, while the four bottom plots show packet throughput.

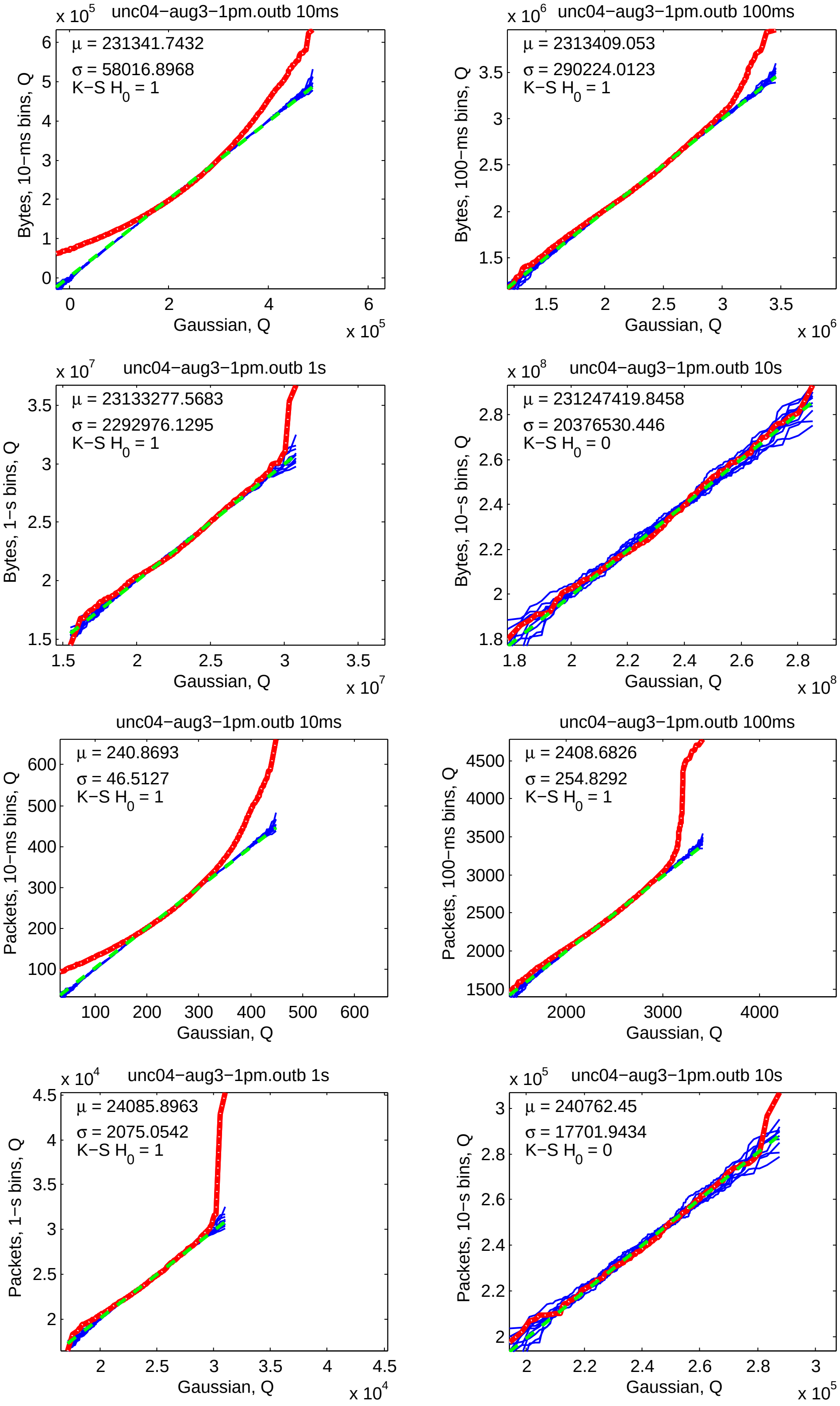


Figure 4.42: Quantile-quantile plots with simulation envelops for the marginal distribution of UNC 1 PM outbound. The top four plots show byte throughput, while the four bottom plots show packet throughput.

difference is slightly larger for packet throughput in the Leipzig-II trace.

- The tails of the marginal distributions from the real traces are substantially heavier. The largest values for the Leipzig-II traces are 75% larger, while those for the UNC 1 PM trace are 20-25% larger.

These deviations are present in every one of our traces, showing that throughput marginal distributions deviate from the normal distribution systematically.

The deviation from normality of the empirical marginal distributions is statistically significant. First, every marginal distribution from the traces fails the Kolmogorov-Smirnov test of normality [NIS06]. Second, every Quantile-Quantile (Q-Q) plot [NIS06] shows a clear departure from normality. This is true not only for the 10-millisecond time-scale, but also for the 100-millisecond, the 1-second time series, and even for the 10-second time series in some cases. We illustrate this type of analysis again using the throughput marginals of Leipzig-II inbound and UNC 1 PM outbound. The plots in Figures 4.41 and 4.42 show Q-Q plots for different time-scales, where the quantiles of the data and the theoretical normal distribution are compared using a thick line with white dots. If the data were normally distributed, the Q-Q line would closely follow the the dashed 45 degree line. This is clearly not the case, but the Q-Q plot does not provide any sense of statistical significance. To address this deficiency, the plots also show simulation envelopes, depicted using thin, dark-gray lines, following the methodology in Hernández-Campos *et al.* [HCMSS04]. They are easiest to see in the 10-second time series, the ones with the least over-plotting. Each line in the envelope corresponds to a distribution constructed by sampling the theoretical normal distribution as many times as values were present in the empirical marginal distribution. The envelope therefore captures the natural variability of the normal distribution for the given sample size. If the Q-Q line comparing the empirical marginal and the theoretical normal distribution is outside this envelope, the deviation from normality is considered statistically significant. This is clearly the case for every marginal distribution in the plots, except the ones at the 10-second scale of Leipzig-II inbound.

The plots in Figures 4.41 and 4.42 also show the results of the Kolmogorov-Smirnov test (K-S), a formal test of normality. The third line in the inside legend, below the sample mean μ and the standard deviation ρ, shows the result of the formal test. The null hypothesis (non-normality) can only be rejected for the marginals of the Leipzig-II throughput at the 10-second time-scale. The plots show a $H_0 = 0$ when the null hypothesis can be rejected, and a $H_0 = 1$ when it cannot be rejected. Given these results, assuming normality to study the marginals of our traces (and those of their synthetic versions) is of dubious value. We will restrict ourselves to comparative plots of CDFs and CCDFs for comparing throughput marginals.

Note that we are not arguing that our finding of pervasive deviations from normality invalidates earlier studies based on the assumption of normality in throughput marginals. From our analysis, the bodies of the marginals are close enough to the normal distribution that assuming normality can provide a useful simplification. As long as significant deviations from normality in the tails have little or no effect on the reasoning, assuming normality makes analytical studies more treatable and even more intuitive.

Our finding of non-normality in our traces is consistent with the observation by Sarvotham *et al.* [SRB01]. These authors demonstrated that deviations of the throughput marginal from normality can be explained by the presence of an *alpha* component in Internet traffic. Alpha traffic is composed of connection with high throughputs that transfer large amounts of data. In contrast, connections with moderate or low throughputs and connections with moderate or small amounts of data to transfer are considered *beta* traffic, whose throughput marginal is normally distributed. Intuitively, a traffic generation method should be able to reproduce both the alpha and the beta components of Internet traffic. Sarvotham *et al.* also proposed to consider traffic bursty when its throughput marginal deviates from normality. This is an alternative (and complementary) view of traffic burstiness, which is more commonly associated with long-range dependence in the arrival process, as we will discuss next.

4.2.3 Throughput Self-Similarity and Long-Range Dependence

A remarkable characteristic of Internet traffic is its high variability in throughput across a wide range of time scales, and how that variability changes as scale increases. If we plot the number of packets or bytes that arrive at a network link, say every 1 or 10 milliseconds, we observe a highly variable process where the number of arrivals is constantly changing. If we plot these arrivals at a coarser scale, say every 100 milliseconds or 1 second, this high variability does not decrease significantly. In contrast, Poisson arrivals exhibit a rapid decrease in variability as we increase the scale of the time series. For this reason, it is often said that Internet traffic has a “very bursty” arrival process, far more variable than that of call arrivals in a phone network. Starting with the work of Leland *et al.* [LTWW93], traffic burstiness has usually been characterized using the theoretical framework of statistically self-similar processes. This framework provides some powerful methods to study traffic burstiness and quantify its strength.

The motivation behind the study of traffic burstiness is the observation that an increase in the burstiness of traffic results in a more demanding network workload. For example, Erramilli *et al.* [ENW96] demonstrated that router queues exhibit dramatically heavier distributions of queue lengths as the burstiness of the input packet arrival process increases. Numerous measurement studies, *e.g.*, [WTSW97, ZRMD03, PHCMS05, PHCL$^+$], have examined Internet traffic and consistently observed highly bursty arrivals that appear self-similar for scales between a few milliseconds and tens of seconds. It is therefore expected that representative synthetic traffic reproduces this high burstiness. In this dissertation, we employ well-known methods to assess the self-similarity of real and synthetic traffic, and verify that our traffic generation methods can reproduce the level of burstiness in Internet traffic.

The term self-similarity comes from the study of fractal objects. Fractals are geometrical constructs that appear similar a different scales. The most famous example of fractal is the Mandelbrot set, whose cardioid shape repeats itself as we zoom into the set. In this fashion, a second-order self-similar time series shows a similar pattern of variation at different time-scales. For this reason, self-similarity is also known as scale-invariance. Some authors talk about "traffic scaling" or simply "scaling" to refer to the observed self-similarity in network traffic.

Quantitatively, the change in the arrival variance for a self-similar time series of bin counts X^t is proportional to t^{2H-2}, where $t \geq 1$ represents scale as the aggregation of arrival counts, and H is known as the Hurst parameter. For example, the variance in bin counts in a Poisson process is proportional to $\frac{1}{H} = t^{2(\frac{1}{2})-2}$. That is, a Poisson arrival process has $H = 0.5$. A stationary, long-range dependent process has $0.5 < H < 1$. The closer the value of the Hurst parameter is to 1, the slower the variance decays as scale (t) increases, and the traffic is said to be increasingly more *bursty* (than Poisson arrivals). The slow decay of the arrival variance in self-similar traffic, as scale increases, is in sharp contrast to the mathematical framework provided by Poisson modeling, in which the variance of the arrivals process decays as the square root of the scale (see [LTWW93, PF95]). This quantitative characterization of self-similarity provides us with the right framework to compare real and synthetic traffic, assessing the validity of the traffic generation process in terms of the burstiness of the packet/byte arrival process.

Self-similarity also manifests itself as Long-Range Dependence[13] (LRD) in the time series of arrivals. This means that there are non-negligible correlations between the arrival counts in bins that are far apart. A common way of studying these correlations is to compute the autocorrelation $\rho(k)$ of a time

[13] Long-range dependence is sometimes referred to as *long memory.*

series, where k is the autocorrelation lag. The autocorrelation at lag k,

$$\rho(k) = \frac{\sum_{i=1}^{n-k}(X_i^t - \overline{X}^t)(X_{i+k}^t - \overline{X}^t)}{\sum_{i=1}^{n}(X_i^t - \overline{X}^t)^2},$$

is the correlation between a time series and a shifted version of itself, where the i-th value in the original time series becomes the $i+k$-th value in the shifted time series. The autocorrelation function $\rho(k)$ of a long-range dependent time series decays in proportion to $k^{-\beta}$ as the lag k tends to infinity, where $0 < \beta < 1$. The Hurst parameter is related to β via $H = 1 - \beta/2$, so the closer the value of the Hurst parameter is to 1, the more slowly the autocorrelation function decays. In contrast, Poisson processes are short-range dependent, *i.e.*, their autocorrelation decays exponentially as the lag increases.

The concepts and definitions of self-similarity and LRD assume that the time series of arrivals is *second-order stationary* (also called *weakly stationary*). Loosely speaking, this means that the variance of the time series (and more generally, its covariance structure) does not change over time, and that its mean is constant (so the time series can always be transformed into a zero-mean stochastic process by simply subtracting the mean). The intuitive interpretation of this concept is that the time series should not experience any major change in variance, which would be associated with a fundamental change in the nature of the studied process. For example, a link usually used by 1,000 hosts that suddenly becomes used by 10,000 hosts (*e.g.*, due to a "flash crowd") would show a massive throughput increase, and much higher variance, which would make it non-stationary. These types of major changes are outside the scope of LRD analysis. They represent a coarse-scale feature of the time series which should be studied using other methods (*e.g.*, trend analysis using SiZer [CM99]).

Traffic is certainly not second-order stationary at the scales at which time-of-day effects are important. For example, a 24-hour trace is usually non-stationary due to the sharp decrease in network utilization at night. The number of sources at night is far smaller, which decreases variance, violating the second-order stationarity assumption. Trying to estimate the Hurst parameter of a 24-hour trace that exhibits a time-of-day effect results in a meaningless number. In our traffic generation work, we will estimate Hurst parameters (and other measures of self-similarity) for traces that are second-order stationary. Our traces have moderate durations, between 1 and 4 hours, which greatly diminishes the impact of time-of-day variations. We also carefully examined the packet and byte arrival time series of our traces and found no evidence of sharp changes that could be associated with second-order non-stationary.

Estimation of Hurst parameters is not a trivial exercise. Besides ensuring that no significant second-order non-stationarity is present in the data, common estimation methods are very sensitive to outliers and trends in the data, as pointed out by Park *et al.* [PHCL$^+$]. These difficulties motivate some preprocessing of the studied time series (*e.g.*, detrending) or to employ robust methods. In this dissertation, we will make use of wavelet analysis to study the scaling properties of real and synthetic traffic. We will follow the analysis method of Abry and Veitch [AV98] and make use of their Matlab implementation of the method. In general, we will compute what is called the *wavelet spectrum* of the time series of packet and byte counts in 10 millisecond intervals. This is also referred to as the *logscale diagram* in some works[14]. The *wavelet spectrum* provides a visualization of the scale-dependent variability in the data (see Figure 4.43 for an example). Briefly, a logscale diagram plots the logarithm of the (estimated) variance of the Daubechies wavelet coefficients, the *energy*, as a function of the logarithm of the scale $j = log_2(t)$, where t is the time scale and j is known as the *octave*. The Daubechies wavelet coefficients come from a decomposition of the time series in terms of the Daubechies wavelet basis, which is a collection of shifted and dilated versions of a mother Daubechies wavelet (a function) [Wal99]. Intuitively, this decomposition is similar to the Fourier transform, which decomposes a time series in terms of sinusoidal functions. The wavelet transform also performs a decomposition but it uses a compact support, so it can represent localized features (sinusoidal functions have infinite support). Besides this property, the benefit of the wavelet transform is its robustness to trends in the data, which can easily confuse other types of analysis, such as the variance-time plot [LTWW93]. Wavelet analysis is robust to moderate non-stationarities.

For processes that are long-range dependent, the wavelet spectrum exhibits an approximately linear relationship with a positive slope between energy and octave. For Internet traffic, the region where this linear scale relationship begins is generally on the order of a few hundred milliseconds (4th to 6th octave for 10-milliseconds time series). An estimate of the Hurst parameter H along with a confidence interval on the estimate can be obtained from the slope of the wavelet spectrum, $H = \frac{slope+1}{2}$. See the book edited by Park and Willinger [PW00] for a more complete overview of long-range dependence in network traffic, and papers by Veitch *et al.* [AV98, HVA02] and Feldmann *et al.* [FGHW99, Fel00] for more detail on traffic analysis using wavelets.

Figure 4.43 shows the wavelet spectra of the time series of packet throughputs for fully-captured sequential connections in the Leipzig-I trace. As explained in Section 4.2.1, this type of connection is responsible for the overall burstiness of the traffic in our trace[15]. For comparison, Figure 4.43 also

[14] The rationale for choosing the term "logscale diagram" can be confusing, since it is applicable to any kind of plot in which one or more axes show a logarithmic transformation of the data. The term "wavelet spectrum" is more specific and seems more appropriate and has become the standard in the literature.

[15] The only exception is the Abilene-I trace in the direction from Cleveland to Indianapolis, where routing asymmetries are responsible for most of the burstiness.

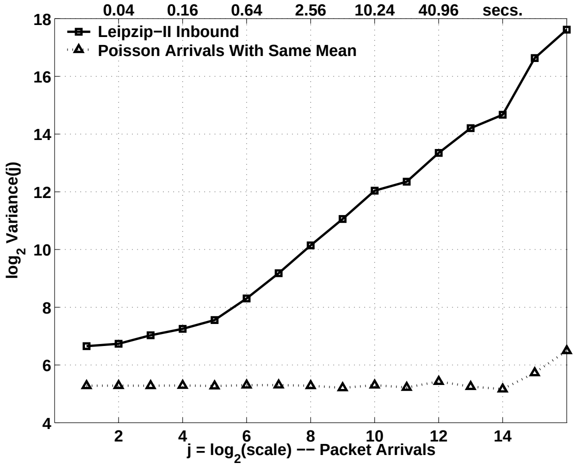


Figure 4.43: Wavelet spectra of the packet throughput time series for Leipzig-II inbound and its Poisson arrival fit.

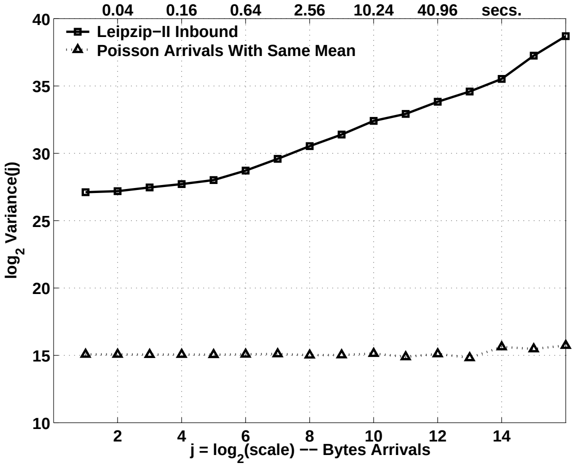


Figure 4.44: Wavelet spectra of the byte throughput time series for Leipzig-II inbound and its Poisson arrival fit.

plots a simulated time series of Poisson arrivals with the same mean (38.94 packets per 10-millisecond bin[16]). Note that only the middle 150 minutes of the Leipzig time series were used, eliminating the non-stationarity created by the boundaries of the trace. The plot shows the variance of the wavelet coefficient (or energy) as a function of the octave. The first octave comes from the dyadic aggregation of 10-milliseconds bins, so it represents the energy at the 20-millisecond scale. The second octave comes from the dyadic aggregation of the bins aggregated in the previous octave, so it represents the energy at the 40-millisecond scale. The same dyadic aggregation is used for every successive scale, so octave 12 represents the energy at the 10 milliseconds times 2^{12} scale, *i.e.*, at the 40.96-second bins. To make the plot more readable, we added labels on top of the plot with the scale given in seconds. Due to the nature of the wavelet basis, the exponential decay of the autocorrelation in a short-range dependent process results in a wavelet spectrum with a slope of zero. On the contrary, the decay in a long-range dependent process is slower than exponential, and results in a wavelet spectrum with a positive slope. The wavelet spectrum of Leipzig-II has a positive slope that indicates long-range dependence, while the synthetic Poisson time series does not show such a trend (it is short-range dependent). Note also that the height of the curves is rather different. This is because the overall variance of the Poisson arrivals is smaller. The standard deviation of the aggregate packet throughput time series was 12.96 while that of the synthetic Poisson arrivals was 6.23. The estimated Hurst parameters were 0.940 (with confidence interval [0.931,0.949]) for the Leipzig-II trace and 0.496 (with confidence interval [0.487, 0.505]) for the synthetic Poisson arrivals.

[16] Note that the simulated time series had exactly this mean, but the number of packets in each bin was always an integer number.

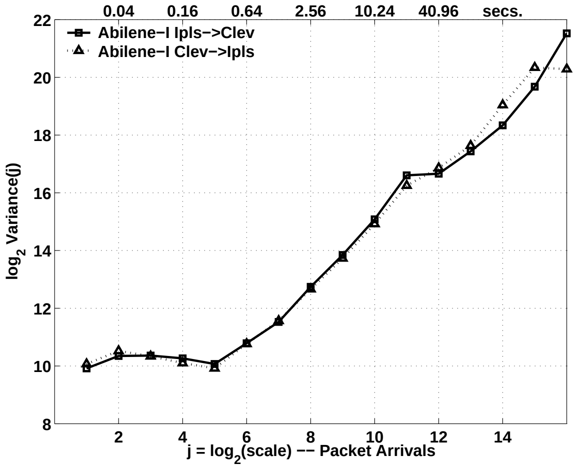


Figure 4.45: Wavelet spectra of the packet throughput time series for Abilene-I.

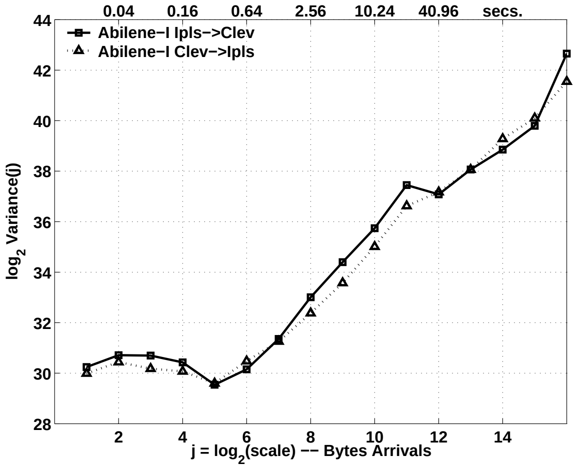


Figure 4.46: Wavelet spectra of the byte throughput time series for Abilene-I.

The same qualitative results hold for byte arrivals, as illustrated in Figure 4.44. Here the mean number of bytes per 10-millisecond bin for Leipzig was 34,400, and the standard deviation of the trace was 14,000, while the standard deviation of the synthetic Poisson arrivals was only 188. The estimated Hurst parameters were 0.941 (with confidence interval [0.932,0.950]) for Leipzig-II and 0.496 (with confidence interval [0.487, 0.505]) for the synthetic Poisson arrivals.

Figure 4.45 shows the wavelet spectrum of the packet throughput time series for Abilene-I (in the two directions: Indianapolis to Cleveland and Cleveland to Indianapolis). While the overall impression is similar to that of the previous figures, we find a change in slope after the 11th octave. Note that both directions exhibit similar long-range dependence. The estimated Hurst parameters were quite high: 1.016 (confidence interval [1.005, 1.027]) for the Indianapolis to Cleveland trace, and 1.009 (confidence interval [0.998, 1.019]) for the opposite direction. Byte throughput for the same trace shown in Figure 4.46 is qualitatively similar. The estimated Hurst parameters were 1.169 (confidence interval [1.158, 1.180]) and 1.046 (confidence interval [1.035, 1.057]). Both are significantly above 1, so some non-stationarity is present in the trace.

Another example of this type of analysis is given in Figures 4.47 and 4.48. For UNC 1 PM, these diagrams show a large separation between the two directions, that translates into significantly different Hurst parameters. The entire set of Hurst parameters for the traces considered in this dissertation is shown in Tables 4.1 and 4.2.

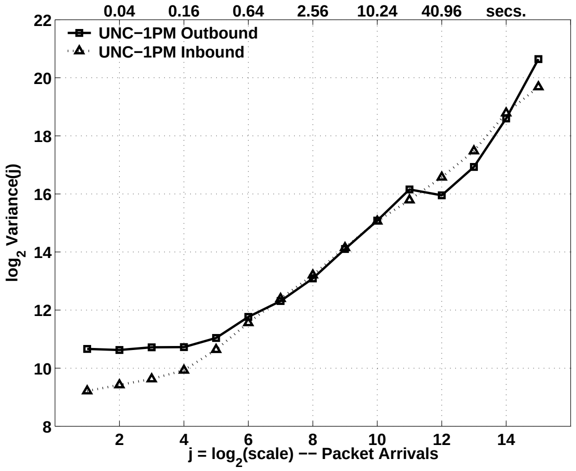


Figure 4.47: Wavelet spectra of the packet throughput time series for UNC 1 PM.

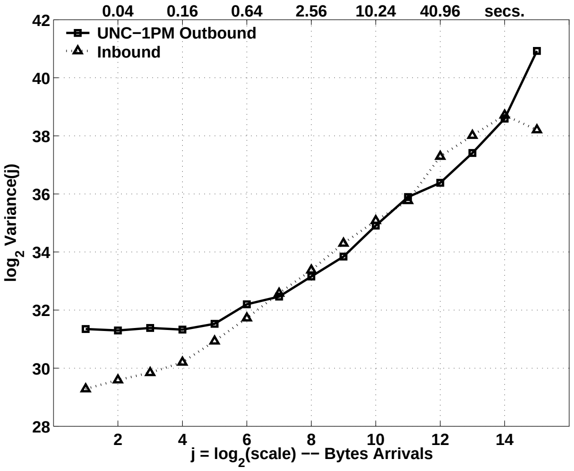


Figure 4.48: Wavelet spectra of the byte throughput time series for UNC 1 PM.

Trace	Estimated Parameters
Leipzig-I Inbound	H=0.940356 C.I.=[0.931459, 0.949254]
Leipzig-I Outbound	H=0.968425 C.I.=[0.959527, 0.977322]
Abilene-I Ipls/Clev	H=1.016014 C.I.=[1.005242, 1.026786]
Abilene-I Clev/Ipls	H=1.008771 C.I.=[0.998000, 1.019543]
UNC 1 PM Outbound	H=0.890024 C.I.=[0.872508, 0.907541]
UNC 1 PM Inbound	H=0.926588 C.I.=[0.909072, 0.944105]
UNC 1 AM Outbound	H=0.906053 C.I.=[0.888537, 0.923569]
UNC 1 AM Inbound	H=0.932574 C.I.=[0.915058, 0.950091]
UNC 7:30 PM Outbound	H=1.001424 C.I.=[0.983908, 1.018940]
UNC 7:30 PM Inbound	H=0.981452 C.I.=[0.963935, 0.998968]

Table 4.1: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of five traces.

Trace	Estimated Parameters
Leipzig-I Inbound	H=0.941176 C.I.=[0.932278, 0.950073]
Leipzig-I Outbound	H=1.019947 C.I.=[1.011049, 1.028844]
Abilene-I Ipls/Clev	H=1.169007 C.I.=[1.158236, 1.179779]
Abilene-I Clev/Ipls	H=1.045921 C.I.=[1.035149, 1.056692]
UNC 1 PM Outbound	H=0.820944 C.I.=[0.803428, 0.838460]
UNC 1 PM Inbound	H=0.925690 C.I.=[0.908174, 0.943206]
UNC 1 AM Outbound	H=0.906226 C.I.=[0.888710, 0.923742]
UNC 1 AM Inbound	H=0.957370 C.I.=[0.939854, 0.974887]
UNC 7:30 PM Outbound	H=0.963306 C.I.=[0.945789, 0.980822]
UNC 7:30 PM Inbound	H=0.970991 C.I.=[0.953474, 0.988507]

Table 4.2: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of five traces.

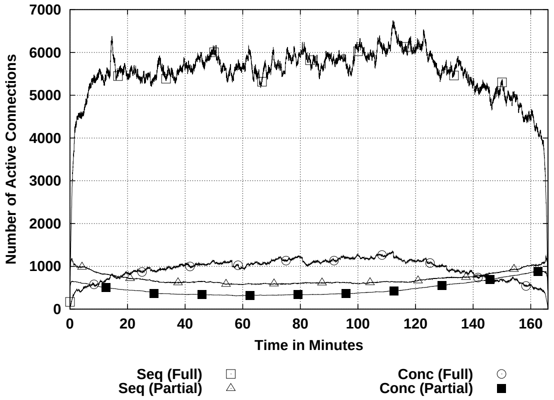


Figure 4.49: Breakdown of the active connections time series for Leipzig-II.

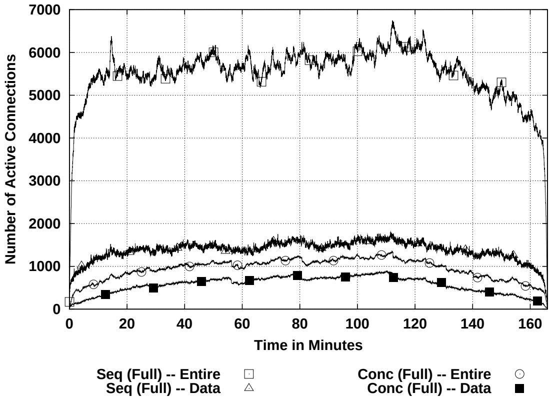


Figure 4.50: Impact of the definition of active connection on Leipzig-II.

4.2.4 Time Series of Active Connections

Another important metric for describing the workload of a network is the number of connections that are simultaneously active. The feasibility of deploying mechanisms that must maintain some amount of state for each connection is highly dependent on this metric. For example, stateful firewalls can selectively admit packets belonging to connections started from a protected network, and not those packets from connections that originated somewhere else on the Internet. This kind of filtering requires to maintain state for every connection observed in the recent past. Similarly, network monitoring equipment often reports on the number of connections and their aggregate characteristics, and tries to identify *heavy-hitters* that consume large amounts of bandwidth. This also requires per-connection state. A good example of this type of monitoring is Cisco's NetFlow [Cor06]. The performance of other mechanisms, such as route caching, may also be affected by the number of active connections. Evaluating these types of mechanisms and their resource consumption requirements can only be accomplished using synthetic traffic that is realistic in terms of the number of connections that are simultaneously active.

One important difficulty when analyzing the time series of active connections is the way connection start and end times are defined. The most obvious way to define connection start and end times is to consider the first and the last segment of a connection as the boundaries of the connection. Figure 4.49 shows the time series of active connections in 1-second intervals using this technique for the Leipzig-II trace. As in the throughput time series in Figures 4.21 and 4.22, the number of active connections from fully-captured sequential connections is much larger than the number of active connections for the other types of connections.

As the focus of our work is on the effect of source-level behavior, we can also use an alternative

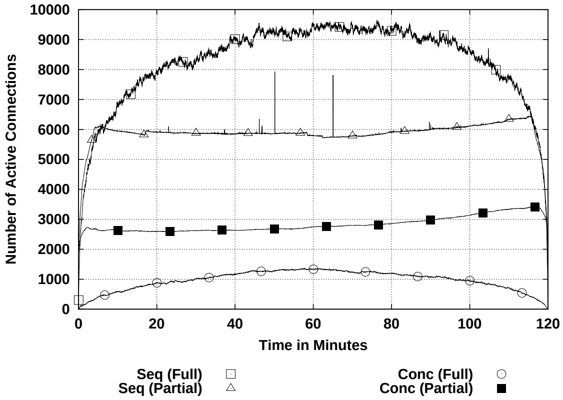


Figure 4.51: Breakdown of the active connections time series for Abilene-I.

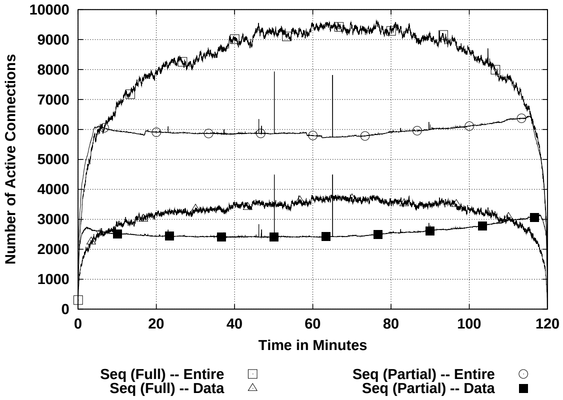


Figure 4.52: Impact of the definition of active connection on Abilene-I.

definition in which a connection is considered active as soon as it sends the first data segment, and inactive as soon as it sends the last data segment. Interestingly, these two definitions result in quite different time series. Figure 4.50 compares the time series for fully-captured sequential and concurrent connections (the time series for partially-captured connections changed very little). The average number of active connections is much smaller when only the data exchange portion of TCP connections is considered. The main cause of this difference is the presence of significant quiet times between the last ADU and connection termination. Figure 3.28 in the previous chapter showed the distribution of this quiet time. In some cases, we also observe quiet time between connection establishment and the first ADU. The duration of connection establishment and connection termination is generally very short (around two round-trip times), but we have observed numerous cases in which losses and TCP implementation problems[17] lengthened them substantially. We believe the second definition, considering only duration between data segments, is more useful for studying the realism of synthetic traffic, since connection establishment and termination create very little network load when compared to the actual exchanges of data. Furthermore, congestion control plays little role when no data is being exchanged. We will use this second definition of active connection in the rest of this work.

The breakdown of the active connection time series for Abilene-I is shown in Figure 4.51. Partially-captured sequential connections are far more significant for this trace, reaching 2/3 of the average number of fully captured sequential connections. We also note that the time series exhibits surprisingly small variability except for a few very small spikes in the middle. Finally, and in contrast to the breakdown of byte throughput for Abilene-I, the number of active connections from partially-captured concurrent connections is less than half of the number of active connections from partially-captured sequential

[17] For example, some implementations send several reset segment after a lossy connection termination, and these segments are often separated by long period of inactivity.

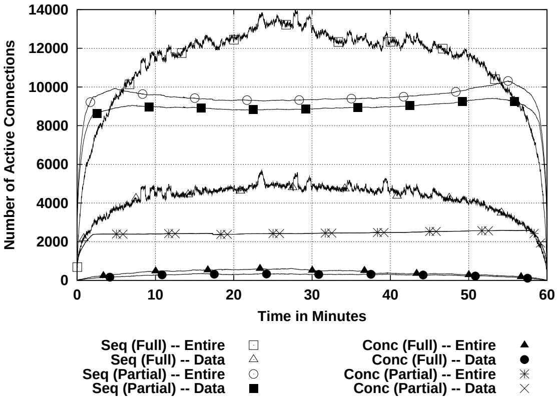


Figure 4.53: Breakdown of active connections time series for UNC 1 PM using both definitions of active connection.

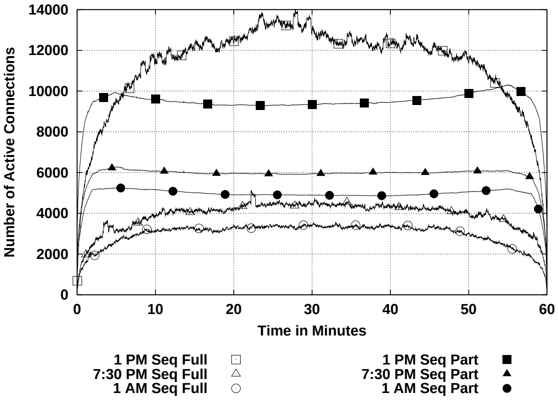


Figure 4.54: Impact of the time-of-day on the active connections time series for the three UNC traces.

connections.

Figure 4.52 illustrates the impact of the definition of active connection. The time series from both fully- and partially-captured sequential connections decrease considerably when only the data exchange part of the connection is considered. Note also that the spikes in the time series of partially-captured connections are not affected by the change in the definition. This is interesting when we observe that the magnitude of the variability of the other time series did decrease significantly.

The largest number of active connections was found in UNC 1 PM as shown in Figure 4.53. This is surprising given that Abilene-I carries more bytes and packets, and should be explained by the differences in the mix of applications that drives the traffic in these two links. The plots shows that fully- and partially-captured sequential connections are affected in a very different way by the definitions of active connections. While the number of active connections for fully-captured sequential connections decreases very significantly, the number for partially-captured ones is almost the same. This can be explained by long connections that were active throughout the entire duration of the trace.

Finally, Figure 4.54 studies the impact of the time of the day on the time series of active connections for sequential connections. The number of fully-captured sequential connections is more sensitive to the definition of active connection than the number of partially-captured sequential connections.

4.3 Summary

The first part of this chapter presented our approach for introducing realistic *network-level parameters* in our traffic generation methodology. In particular, we considered how to measure three basic network parameters that have a major impact on the throughput of a TCP connection: round-trip time, receiver window size, and loss rate. As in our analysis of source-level behavior, we focused on the efficient analysis of segment headers for extracting these network parameters, and evaluated the accuracy of our chosen measurement methods using testbed experiments.

Our discussion on measuring round-trip time considered the classic SYN estimator, and proposed a novel technique based on computing one-side transit times (OSTTs). Our technique has two main advantages. First, it is applicable to connections observed both on the edges and on the core of the network. In either case, it provides us with a way to measure the distance, in terms of network delay, between the monitoring point and the end hosts taking part in each connection. Second, OSTT-based estimation provides a number of samples proportional to the number of data segments on a TCP connection, unlike the single sample that can be obtained using the SYN estimator. This provides a better way to understand the inherent variability in round-trip times. It also served us to study the impact of delayed acknowledgments on path round-trip time estimation from segment headers. We clearly showed that delayed acknowledgments substantially inflate estimates of round-trip time that rely on non-robust statistics like averages and maxima. For this reason, we favor the use of minima or medians to estimate path round-trip time, which were proved to be highly accurate in our testbed experiments.

We also studied the empirical distributions of round-trip times in our collection of five traces. We can highlight several observations. The edge traces from UNC and Leipzig showed between 20% and 35% of connections with very short round-trip times below 20 milliseconds. In contrast, the backbone trace from Abilene showed less than 1% of connections with these small round-trip times. Our analysis of the total number of bytes carried in connections with a given round-trip time revealed that Leipzig-II had a far larger fraction of bytes (10%) carried in connections with round-trip times above 500 milliseconds. The distributions of round-trip times did not only differ substantially on their range, but also on their shapes, even among those collected on the same site. For example, the UNC 1 PM trace showed only 15% of connections with round-trip times above 100 milliseconds, while this percentage became 25% and 38% for UNC 7:30 PM and 1 AM respectively.

The second parameter we considered is the maximum size of the receiver window, which, in combination with the round-trip time, puts a hard limit on the maximum throughput of a TCP connection. This parameter is straight-forward to measure, since each TCP segment contains a field with the size of the receiver window at the time of its sending. Taking the maximum of the observed receiver windows provides an accurate way of measuring the largest receiver window supported by an endpoint, even for connections that grow their limit some time after the connection is opened. We used this technique to study the distribution of maximum receiver window sizes in our traces, and found a large fraction of connections with a small maximum. Between 45% and 65% of the connections had maximum receiver window sizes below 20 KB, which is well below the 64 KB limit.

The last network parameter that we studied was the segment loss rate. Loss has a substantial impact on TCP connections. First, losses force the endpoints to retransmit segments to maintain a reliable communication. Second, TCP endpoints use losses as the signal of congestion, and react to them by lowering their sending rate. For these two reasons, even a small number of losses can have a dramatic effect on a TCP connection. Measuring loss rates purely from segment headers must necessarily be based on the same mechanisms used by TCP endpoint to detect losses: retransmissions and duplicate acknowledgments. We proposed a technique to measure the loss rate of data segments using these signals, where differentiating between losses before the monitoring point, detected using duplicate acknowledgments, and losses after the monitoring point, detected using retransmissions. Our evaluation using testbed experiments showed that our technique is reasonably accurate. The experiments also illustrate the impact of lost acknowledgments, which increase data segment loss rates, and variability introduced by simulating losses using *dummynet* 's dropping mechanism. We also studied the loss rates in our traces, and found that between 92.5% and 96.2% of the TCP connections experienced no losses. However, connections with one or more losses accounted for 46% (Leipzig-II) to 78% (UNC 1 AM) of the total bytes in traces, and connections with loss rates above 1% (*i.e.*, moderately high) accounted for 8% (Abilene-I) to 34% (UNC 1 AM) of the total bytes.

The second part of this chapter described our approach for comparing real and synthetic traffic using several *network-level metrics*. The goal of such a comparison is to evaluate how closely synthetic traffic generated on a closed-loop manner can reproduce the aggregate characteristics of real traffic. This type of comparison concerns itself with the *extrinsic* characteristics of the generated traffic, which were not a direct input to the traffic generators. On the contrary, evaluating how well source-level properties and network-level parameters are preserved by our traffic generation method and its implementation focuses on *intrinsic* characteristic of the generated traffic, which are the input to the traffic generation system. We first discussed how to study the time series of packet and byte throughputs, using plots of time series at a coarse scale, tens of seconds. This broad view was specially useful to identify major trends and features in the traffic. We used this approach to study the composition of our traces, finding that sequential connections are mostly responsible for the features of the time series, being the aggregate throughput for concurrent connections generally smooth. We further differentiate between traffic from connections for which we observed every packet between TCP connection establishment and termination, uncovering substantial boundary effects in the UNC traces and to some extent in the Abilene-I trace. We also showed that the fraction of the total throughput from unidirectional connections is generally negligible. The only exception is Abilene-I, where routing asymmetries explain the finding that 1/4 of total Cleveland-to-Indianapolis bytes were carried in connections whose packets appear in only one direction of the trace.

The second way in which we proposed to examine throughput was to construct the marginal distributions of the time series at a fine-scale (10 milliseconds). While marginals ignore dependency structure, their interpretation in networking terms is intuitive. Plots of the body of the marginal distribution provide an overview of the range of fine-scale throughputs in a trace, while plots of the tail of the marginal distribution make the highest (fine-scale) throughputs stand out. The analysis of our traces showed that Poisson arrivals cannot be used to model neither packet or byte throughputs. The bodies of the marginal distributions from our traces are between 2 and 3 times more variable that the ones from Poisson arrivals with the same mean. We also showed that the marginal distributions from our traces have statistically significant departures from normality, which are most prominent on the tails. This was demonstrated using two methods, Q-Q plots with simulation envelopes and the Kolmogorov-Smirnov test of normality. Both methods were applied to scales of aggregation between 10 milliseconds and 10 seconds. While the distributions became closer to normality as scale increased, only a few of them were statistically consistent with the normal distribution at the 10 second scale. For this reason, our analysis of marginal distribution will rely on CDFs of the bodies and CCDFs of the tails, rather than making assumptions about the underlying statistical distribution.

Our third type of analysis of throughput focused on the long-range dependence of traffic. We employ the wavelet analysis for this purpose, which has been shown to be robust and accurate in the literature. This method provides both an overview of the way in which variability changes with scale using wavelet spectra plots, and a state-of-the-art estimator of Hurst parameter with confidence intervals. Our discussion illustrated how clearly wavelet spectra and Hurst parameter estimates differentiate between the short-range dependence in Poisson arrivals and the long-range dependence in our traces. Our traces show remarkably high Hurst parameter estimates, well above 0.9 for both packet and byte throughput.

Finally, the chapter introduced the plot of the time series of active connections. This type of analysis is essential to validate the realism of traffic generation for certain experiments where per-connection state is important. Our analysis considered two definitions of active connections: a connection was considered active between the arrivals of its first and last segments, or between the arrivals of its first and last segments that carried application data, *i.e.*, not control segments. We demonstrated that these two definitions have a dramatic impact on the number of active connections. We will favor the latter definition (data active connections) for our evaluation in Chapter 6, since the focus of our modeling is the source-level behavior in terms of useful data exchanges. Our discussion of active connections also considered the effect of trace boundaries, revealing a large fraction of active connections from partially-captured connections.

CHAPTER 5

Generating Traffic

Today's scientists have substituted mathematics for experiments, and they wander off through equation after equation, and eventually build a structure which has no relation to reality.

— Nikola Tesla (1857–1943)

Reality is merely an illusion, albeit a very persistent one.

— Albert Einstein (1879–1955)

This chapter discusses the use of the data acquisition and modeling methods presented in the two previous chapters to generate traffic in network experiments. In addition, it discusses the overall methodology we have developed for validating our traffic generation approach. We will distinguish between validating the method itself, and studying how closely the generated traffic approximates real traffic for properties not directly incorporated in the method. In this chapter, we consider the validation of the method itself, which means to verify that the source-level properties and network-level parameters of the traffic are preserved by the traffic generation method. The study of other properties is left for the next chapter.

5.1 Replaying Traces at the Source-Level

Our approach to traffic generation is illustrated in Figure 5.1. Given a packet header trace $\mathcal{T}_h$ collected from some Internet link, we first use the methods described and evaluated in Chapters 3 and 4 to analyze this trace and describe its content. This description is a collection of connection vectors $\mathcal{T}_c$. Each vector describes the source-level behavior of one of the TCP connections in $\mathcal{T}_h$ using either the sequential or the concurrent a-b-t model. In addition, each vector includes the relative start time of each connection, and its measured round-trip time, TCP receiver window sizes and loss rate. The basic

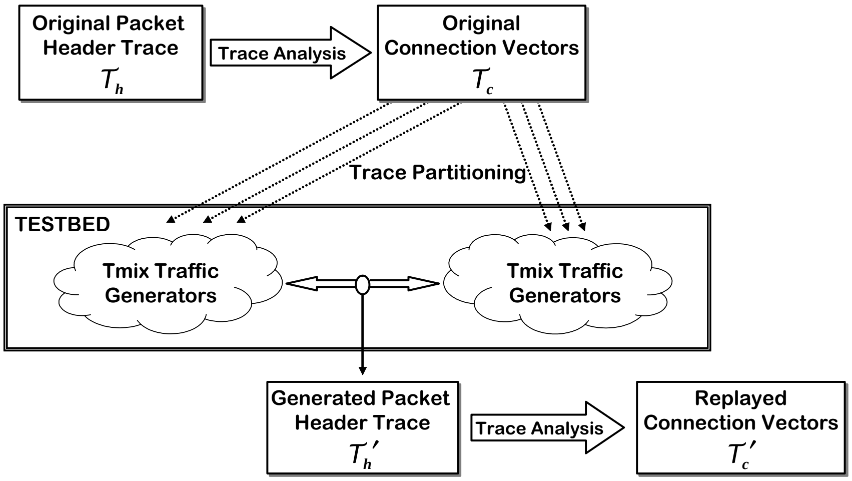


Figure 5.1: Overview of Source-level Trace Replay.

approach for generating traffic according to $\mathcal{T}_c$ is to replay each connection vector. For each connection vector, the replay consists of starting a TCP connection, carefully preserving its relative start time, and reproducing ADUs and inter-ADU quiet times. We call this traffic generation method *source-level trace replay*, and we have implemented it in a network testbed. Source-level trace replay in our environment implies the need to first partition $\mathcal{T}_c$ into disjoint subsets and then assign each subset to a pair of traffic generators. Partitioning is important in our environment, since the high throughput and large number of simultaneously alive connections in our real traces prevents us from using a single pair of traffic generators. We provide further details on our partitioning method in 5.1.1.

The goal of the direct source-level trace replay of $\mathcal{T}_c$ is to reproduce the source-level characteristics of the traffic in the original link, generating the traffic in a *closed-loop* fashion. Closed-loop traffic generation requires to simulate the behavior of applications, using regular network stacks to actually translate source-level behavior (the input of the generation) into network traffic (the output of the generation). In our implementation, described in Section 5.1.2, this is accomplished by relying on the standard socket interface to reproduce the communication in each connection vector. This is a closed-loop manner of generating traffic in the sense that it preserves the feedback mechanisms in the TCP layer, which adapt their behavior to changes in network conditions, such as in congestion. In contrast, packet-level trace replay, which means to directly reproduce $\mathcal{T}_h$, is an open-loop traffic generation method where TCP and lower layers are not used, and the traffic does not adapt to network conditions.

A new packet header trace $\mathcal{T}_h'$ can be obtained from the source-level trace replay of $\mathcal{T}_c$. Our validation of the traffic generation method is then based on analyzing this trace using the same methods used to

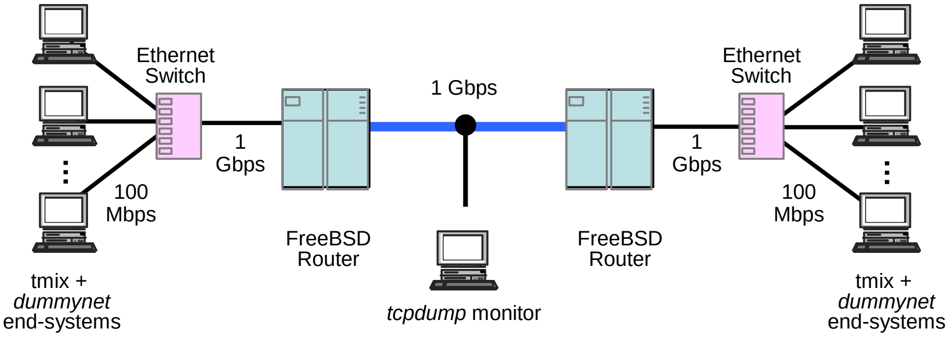


Figure 5.2: Diagram of the network testbed where the experiments of this dissertation were conducted.

transform $\mathcal{T}_h$ into $\mathcal{T}_c$. We then compare the resulting set of connection vectors $\mathcal{T}_c'$ with the original $\mathcal{T}_c$. In principle, they should be identical, since $\mathcal{T}_c$ represents the invariant source-level characteristics of $\mathcal{T}_h$. Section 5.2 studies the results from the source-level trace replay of three traces, assessing how closely $\mathcal{T}_c'$ approximates $\mathcal{T}_c$. $\mathcal{T}_h'$ is necessarily different from $\mathcal{T}_h$. Besides the stochastic nature of network traffic, this is because $\mathcal{T}_h'$ is generated according to $\mathcal{T}_c$, which is a simplified description of the source-level behavior and network parameters in the original trace $\mathcal{T}_h$. It is however important to understand the difference between $\mathcal{T}_h$ and $\mathcal{T}_h'$ in order to understand to what extent $\mathcal{T}_c$ describes the original traffic. Chapter 6 is an in-depth study of this question.

5.1.1 Trace Partitioning

The focus of our traffic generation work is the generation of wide-area traffic in a closed-loop manner. This type of generation process requires to drive a large number of connections by simulating the behavior of the applications on the endpoints. For example, the experiments presented in the latter part of this chapter involve several millions of TCP connections, behaving in the manner specified by as many connection vectors. At any given point in time during the generation, tens of thousands of connections are active. Given CPU, memory and bus speed limitations, a single pair of traffic generators cannot handle such loads, so we generate traffic in our experiment in a distributed fashion. Experiments are conducted in the environment illustrated in Figure 5.2. The goal of the experiment is to generate traffic on the link between the two routers. Traffic is generated by 42 traffic generators, 21 on each side of the network. This type of topology is usually known as the "dumbbell" topology.

Each pair of traffic generators (one on each side) is responsible for replaying the source-level behavior of a (disjoint) subset of the connection vectors in $\mathcal{T}_c$. In our experience, assigning connection vectors to subsets in a round-robin fashion works well. While the resulting subsets are far from being completely balanced, this simple partitioning technique results in subsets that can be easily handled by a pair of traffic generators. We carefully collected statistics on CPU and memory utilization from our source-level trace replay experiments, and found that no pair of traffic generators was ever overloaded. For the results in this dissertation, CPU utilizations were never above 60%, and usually well below that. The use of network connections involves allocating and deallocating pieces of memory known as "mbufs" for buffering purposes. No request for this type of memory was ever denied for the experiments reported in this dissertation. While larger traces than the ones we use in this dissertation could certainly overload our specific environment, our approach is fully scalable, in the sense that $\mathcal{T}_c$ can be partition into an arbitrary number of subsets. This means that the number of traffic generators can increase as much as necessary to handle the replay of any trace without running into resource constraints. This is obviously true as long as no individual connection requires more resources than those provided by an entire traffic generator end host.

5.1.2 Conducting Experiments

We have developed a traffic generation tool, *tmix* , which accurately replays the source-level behavior of an input set of connection vectors using real TCP sockets in a FreeBSD environment. In addition, we make use of a modified version of *dummynet* [Riz97] to apply arbitrary packet delays and packet drop rates to the segments in each connection[1] Our version of *dummynet* , that we will call *usernet* in the rest of this text, implements a user-level interface that can be used by *tmix* instances to assign per-connection delays and loss rates read from the input set of connection vectors. Finally, a single program, *treplay*, is used to control the setup of the experimental environment, configure and start *tmix* instances (assigning them a subset of $\mathcal{T}_c$ and a traffic generation peer), and collect the results.

Tmix is a user-level program that receives a collection of connection vectors as input, and generates traffic according to their source-level behavior. Figure 5.3 illustrates the relationship between *tmix* and the network layers in the traffic generation end host in which a *tmix* instance runs. *Tmix* instances rely on the standard socket interface to create a connection, send and receive ADUs, and to close the connection. The socket interface is an Application Programming Interface (API) that enables user-level programs, such as *tmix* , to communicate with other end host using a programming abstraction similar to a file. Calls to the socket interface are translated by the kernel into requests to use the process-to-

[1] We thank the members of the FreeBSD project in general, and in particular the creator of *dummynet* , Luigi Rizzo, for their outstanding work. Our empirical work would not have been possible without their generous efforts.

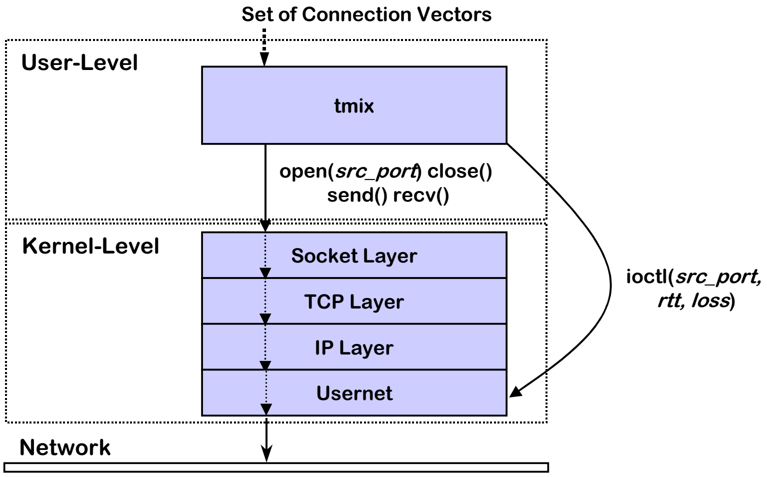


Figure 5.3: End-host architecture of the traffic generation system.

process communication service provided by the transport layer (TCP). The transport layer itself uses the host-to-host communication service provided by the network layer (IP), and the network layer uses the link layer (Ethernet in our case) to handle the network interface and create physical packets.

Usernet

Our experiments also require a special simulation service, *usernet* , which is a modified version of *dummynet* , that provides a highly scalable way of imposing per-connection round-trip times and loss rates. These per-connection round-trip times and loss rates are directly controlled from the user level by *tmix* instances. This requires a direct communication between the *tmix* instance and the *usernet* layer that is not directly supported by the network stack. In order to overcome this difficulty, we use a covert communication channel: the source port number of each replayed connection. By having *tmix* assigning specific source port numbers to each connection, we can then use `ioctl` calls to modify a table at the *usernet* layer that maps source port numbers to round-trip times and loss rates. When a segment is received by *usernet* (from the higher layer), *usernet* can appropriately use the source port number to decide which network parameters should be applied. Source port numbers are unique for each active connection in the same end host, and they are always present in TCP segments[2]. The user-level program, *i.e.*, the *tmix* instance, has therefore to keep track of the (dynamic) source port number that is used for each new TCP connection it opens. Using this technique, *usernet* can determine the delay and loss rate that should be applied to each segment simply by reading an entry in a table indexed by source port

[2] Fragmentation takes place below the *usernet* layer. Figure 5.3 can be confusing in this regard, since fragmentation does take place at the IP layer. *Usernet* is actually embedded in the IP layer.

number, so the lookup time is $O(1)$. The number of source port numbers is small (2^{16}), so this table does not require too much kernel memory (524 KB). No special infrastructure was required to accurately replay the receiver window sizes measured for each connection. This is because these parameters can be directly modified by *tmix* instances using a FreeBSD system call. This approach has worked very well in our experiments.

An alternative solution using traditional *dummynet* would be to use the programmable API of `ipfw`, which makes it possible to add new *dummynet* rules from a user-level program. The idea would be to add a new rule for each connection, again using the source port number to map delay/loss to individual connections. However, this will mean an $O(n)$ lookup cost for each segment, where n is the number of rules, since the current implementation of `ipfw` searches through the rules in a sequential fashion. Given the large number of connections that each end host handles during the experiments, this per-segment lookup is unacceptable.

Another way of introducing per-connection round-trip times was used by Le *et al.* [LAJS03]. This study used random sampling from a uniform distribution whose parameters were be set at the start of the experiment. As seen in Section 4.1.1, the uniform distribution is not a good approximation of real round-trip times. A later refinement enabling sampling from an empirical distribution was rather inflexible, since it required to modify the *dummynet* source code and recompile it for each experiment. The use of *usernet* , which is fully controllable from the user level, is far more convenient.

Replaying an a-b-t Connection Vector

Two instances of *tmix* can replay an arbitrary subset of $\mathcal{T}_c$ by establishing one TCP connection for each connection vector in the trace, with one instance of the program playing the role of the connection initiator and the other instance playing the role of the connection acceptor. To begin, the connection initiator opens the connection and performs one or more socket writes in order to send exactly the number of bytes specified in the first ADU a_1. The other endpoint accepts the connection and reads as many bytes as specified in the ADU a_1. For efficiency, the size of these read and write operations was chosen to be a multiple of the MSS in our Ethernet testbed (1,460 bytes). We made no attempt to actually measure and reproduce the size of the I/O operations in the original connections. The impact of this simplification is likely to be small, given the results in Section 3.4.

One important issue is how to synchronize the two endpoints (*i.e.*, instances of *tmix*) of the connection to replay exactly the same connector vector. This is accomplished by having the first ADU unit in each generated connection include a 32-bit *connection vector id* in the ADU's first four bytes. Connection vector ids are assigned to each connection vector prior to the traffic generation, and they are unique. Since this id is part of the content of the first data unit, the acceptor can unambiguously 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. It also makes it possible to generate traffic without introducing any control traffic into the experiment, *i.e.*, traffic comes only from the replay of connection vectors, and from any need to manage the behavior of the *tmix* instances.

One important design consideration in the implementation of our traffic generation approach is the assumption of independence among flows. While this is not completely realistic, the level of aggregation at which we generate traffic makes it a reasonable approach (see Hohn *et al.* [HVA02] for a related discussion). This assumption makes traffic generation fully scalable, since $\mathcal{T}_c$ can be partitioned into an arbitrary number of subsets. As long as there are enough traffic generation hosts, we can replay traffic from arbitrarily large traces.

5.1.3 Data Collection

We obtain two types of data from each experiment. First, we collect a new packet header trace $\mathcal{T}_h'$, which can be directly compared with the original packet header trace $\mathcal{T}_h$ and analyzed with our methods to extract a new set of connection vectors $\mathcal{T}_c'$. This new set can be directly compared to $\mathcal{T}_c$. Second, *tmix* instances create a number of logs. Some *tmix* logs can be used to verify that the traffic generation host did not run out of resources during traffic generation, and they successfully replayed their subset of $\mathcal{T}_c$. Other *tmix* logs report on the performance of the TCP connections in the experiments. This includes connection and epoch response times and the list of uncompleted connections with a description of their progress by the end of the experiment.

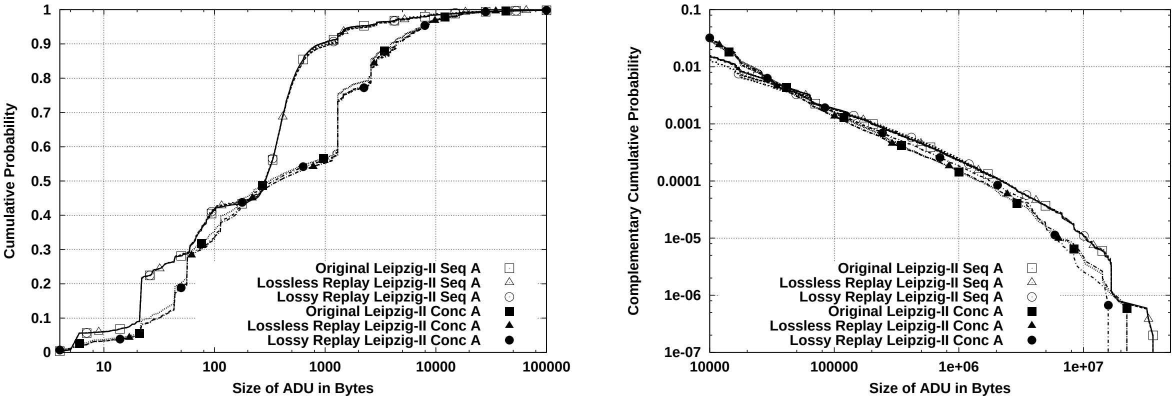


Figure 5.4: Bodies and tails of the A distributions for Leipzig-II and its source-level trace replays.

5.2 Validation of Source-level Trace Replay

In this section, we consider the source-level trace replay of the three packet header traces: Leipzig-II, UNC 1 PM, and Abilene-I. The first goal is to study how well the replay experiments preserve the source-level input, which is the collection of connection vectors $\mathcal{T}_c$ extracted from the original trace $\mathcal{T}_h$. In principle, the characterization of source-level behavior using the a-b-t model represents characteristics of each connection that are invariant to network conditions, so the analysis of the generated trace $\mathcal{T}_h$ should result in a collection of connections vectors $\mathcal{T}'_c$ that is identical to $\mathcal{T}_c$. In practice, there are some practical limitations that make the two sets of connection vectors different. We will discuss the possible causes in this section, and present a statistical comparison of $\mathcal{T}_c$ and $\mathcal{T}'_c$.

The second goal of this section is to study the impact of introducing packet losses in the generated process. For this purpose, we conducted two source-level trace replays of each original trace. The *lossless replay* reproduced the a-b-t connection vector of each original connection, and gave each connection its measured round-trip time and TCP receiver window sizes. The *lossy replay* additionally applied its measured loss rate to each replayed connection. Differences between the lossless and lossy replays tell us about the robustness of both our source-level characterization and traffic generation tools in the presence of losses. These losses are completely absent from our experiments unless they are artificially introduced using *usernet* , as in the lossy replay.

5.2.1 Leipzig-II

The plots in Figure 5.4 compare the distributions of a-type ADU sizes, A, for the original set of connection vectors in Leipzig-II, and for the sets of connection vectors extracted from its lossless and lossy replays. In each plot, the three distributions marked with white symbols correspond to sequential connection vectors, and the ones marked with black symbols to concurrent connection vectors. The left plot shows the bodies of the distributions, using CDFs in log-linear axes. The right plot shows the tails of the distributions, using CCDFs in log-log axes. In general, there is an excellent agreement between the original distributions and those from the source-level replays.

The bodies of the distributions from sequential connections lie on top of each other, even if per-connection loss rates are used during the experiments. As discussed in 3.4, our ADU measurement algorithm can sometimes be inaccurate when one of the last segments of a TCP window is lost before the monitor. In this case, the loss is recovered after a timeout, which can create a quiet time between the consecutive segment that is long enough to unnecessarily split an ADU. This means that a sample a_i from one of the a-type data units in $\mathcal{T}_c$ becomes two samples a'_i and a'_{i+1} in $\mathcal{T}'_c$, such that $a'_i + a'_{i+1} = a_i$. The validation of the data acquisition methods in Section 3.4 demonstrated that *ADU splitting due to TCP timeouts* is possible, although its impact was small even when large data units and aggressive loss rates were used. The comparison of the Leipzig-II lossless and lossy replays, which represent much more realistic traffic, shows that ADU splitting due to TCP timeouts has very little impact in practice, at least for the relatively light distribution of loss rates in Leipzig-II. We can hardly observe any difference between the bodies of the A distributions when losses are added to the replay. The two bodies from the replay are also very similar to the body of the original distribution. The same is true for the tails, which do not show any significant difference. This analysis demonstrates that *tmix* can accurately reproduce the sizes of a-type data units in sequential connections, even when ADUs are large and when experiments are lossy.

There is also a very good match between the A distributions for concurrent connection vectors. In some regions, we notice somewhat thicker lines that come from small offsets of the curves. The tails of the A distribution for concurrent connections are also very similar, although the one from the lossy replay is slightly heavier for values below 5 MB, and slightly lighter for values above that. This could be explained by the inaccuracy discussed above, or by trace boundaries. In the latter case, losses reduce throughput, making the replay of lossy connections are slower than the replay of lossless ones. This means that some a-type ADUs may not have time to complete their transmission before the end of the experiment.

Figure 5.5 compares the distribution of b-type ADU sizes, B, for the connections vectors extracted from the original Leipzig-II trace and their lossless and lossy source-level replays. For sequential con-

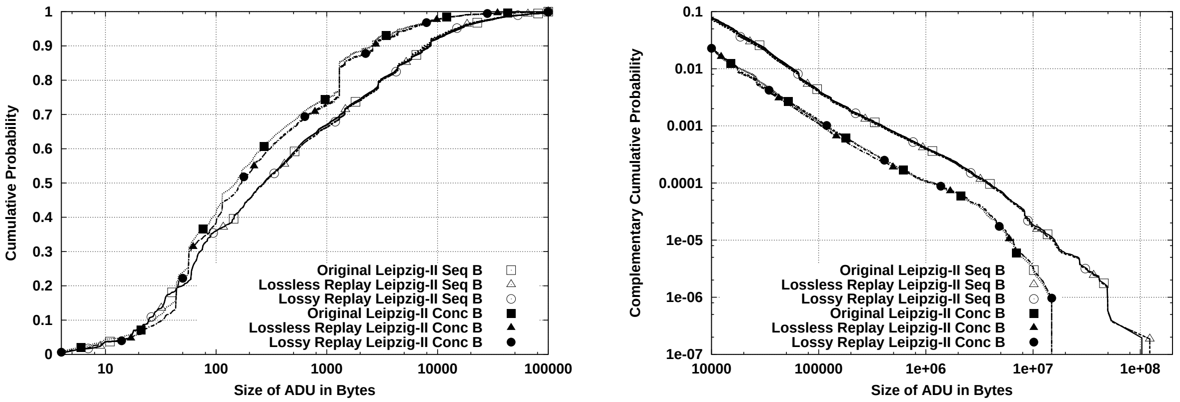


Figure 5.5: Bodies and tails of the B distributions for Leipzig-II and its source-level trace replays.

nection vectors, both the bodies and the tails are identical. For concurrent connection vectors, the distributions show slightly different bodies, but identical tails. The differences cannot be explained by the ADU splitting due to TCP timeouts. If so, we would see a difference between the distributions from the lossless replay and the ones from the lossy replay, but this is not the case. The source of the difference is an inherent problem with the replay of concurrent connections, the *misclassification of the replayed concurrent connections.* While *tmix* always replays a concurrent connection vector in the right way (*i.e.*, decoupling the two directions), the actual set of segments observed at the monitor may simply not have any pair of data segments that satisfy the concurrency test given in Section 3.3.3. In other words, the segments of a replayed concurrent connection may exhibit a fortuitous sequential ordering. As a consequence, the data analysis algorithm classifies as sequential some connections from the replay that were concurrent in the original trace. The sizes of the b-type ADUs in these misclassified connections are then absent from the B distribution for replayed concurrent connections. The small difference in the plot between the original and replayed distributions demonstrates that the number of misclassifications is relatively small, so the majority of the concurrent connections still exhibit concurrent behavior in the replays.

It is important to note that the probability of a misclassification decreases as the sizes of the ADUs increase, since the larger number of data segments makes finding a concurrent pair more likely. Therefore, misclassifications become less significant for the tails of the distributions, since the connections whose samples are in the tail have necessarily at least one large ADU (the one we see in the tail), and are less likely to be misclassified. There is no appreciable difference between the tails of the B distributions from concurrent connections, in agreement with our observation regarding the lower likelihood of misclassification for connections with large ADUs. Misclassified connections are described using the sequential a-b-t model, so they result in additional samples for the distributions that characterize se-

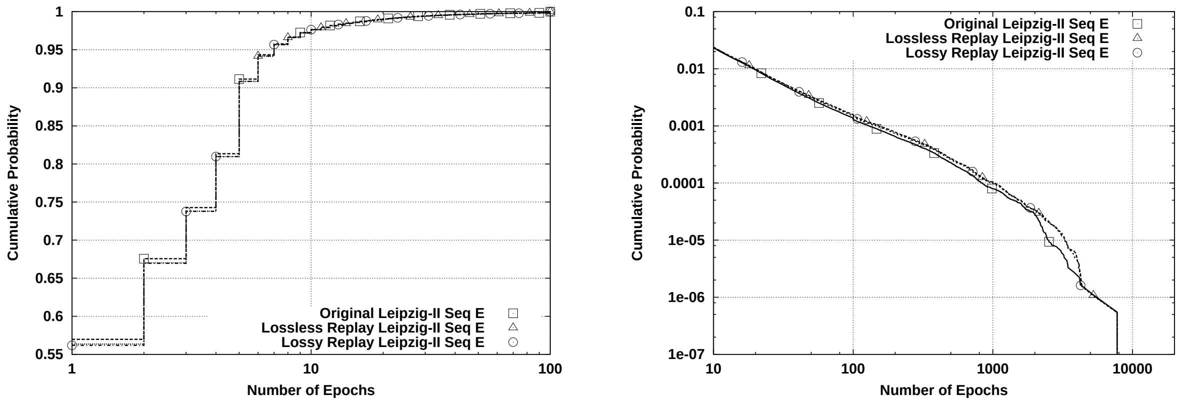


Figure 5.6: Bodies and tails of the E distributions for Leipzig-II and its source-level trace replays.

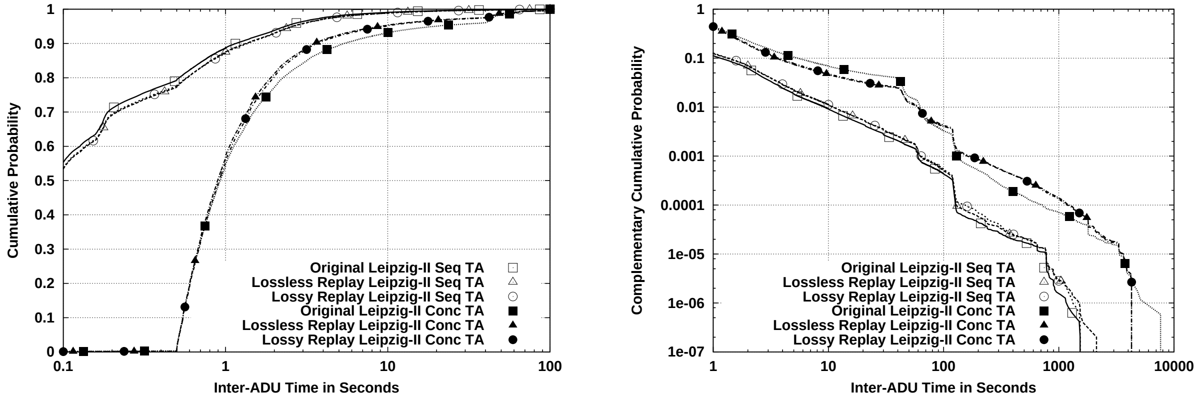


Figure 5.7: Bodies and tails of the TA distributions for Leipzig-II and its source-level trace replays.

quential connection vectors. These extra samples have a much smaller effect on the CDFs, since the number of samples from sequential connections is far larger anyway.

Figure 5.6 considers the distribution of the number of epochs E extracted from the original and from the generated packet header traces. The distributions from the replays are very similar to the original one. The small difference comes again from the small number of misclassified concurrent connections that were considered sequential. Misclassified connections add extra samples to E which slightly distort the distributions from the replays. There is a somewhat bigger difference in the far tail, for connection vectors between 1250 and 1500 epochs. This difference could be explained by misclassification and by trace boundaries (connections replayed more slowly than in the original that do not replay all of their epochs). We observe no difference between lossless and lossy replays in this part of the tail.

The next pair of plots, shown in Figure 5.7, examines the distribution TA of the quiet times on the acceptor side of TCP connections, *i.e.*, between a_i and b_i. The plot of the bodies shows a very good match between the original distribution and the ones measured from the replays of sequential connections. The slightly heavier distributions from the replays is due to a small simplification we made regarding the replay of quiet times. *Tmix* will replay the exact quiet times specified in each connection vector. However, as discussed in Section 3.3.1, when these quiet times are extracted from a packet header trace, the measured quiet time is the sum of two components. The first component comes from the quiet time q at the end host, and the second component comes from the delay d between the monitor and the endpoint. When *tmix* replays a quiet time, it remains quiet for the exact duration of the sum of these components, $q + d$. Given that the replay in the testbed uses *usernet* to reproduce the measured round-trip time of each connection, there is also a delay between *tmix* end hosts and monitor, so the analysis of the generated packet header trace results in quiet times of the form $q + 2d$. It would have been possible to eliminate this inaccuracy by subtracting d from the originally measured quiet times. The value of d is equal to half of the one-side transit time, although delayed acknowledgments and queuing can affect individual samples. We did not try to incorporate a correction for this *quiet time overestimation problem* in our experiments. Besides measurement difficulties, the extra delay becomes less significant in larger quiet times, for which d is far smaller than q. Larger quiet times are far more significant, since they are the ones that can increase the duration of TCP connections substantially.

There is also a good agreement in the tails of the TA distributions, although the distributions from the replays are slightly heavier than the original distributions. This is not explained by the previous overestimation of quiet times due the location of the monitor, because the magnitude of the quiet times in the tail is far larger than the magnitude of d. The source of this small mismatch is the misclassification of some concurrent connections. This is true for both the differences between the tails from sequential connection vectors and between the tails from concurrent connection vectors. It may seem counter-intuitive that the misclassifications makes both types of tails heavier, instead of making one type of tail heavier and the other one lighter. The explanation is that misclassifications move samples from concurrent connections to sequential connections. These moved samples satisfy at the same time the following two properties:

- They have a lighter tail than the tail of the samples left in connections correctly classified as concurrent in the analysis of the generated traffic. The removal of these samples therefore makes the shown distributions from concurrent connections in the replays heavier than the one in the original trace.
- They have a heavier tail than the tail of the samples that they joined in connections correctly classified as sequential in the analysis of the generated traffic. The addition of these samples

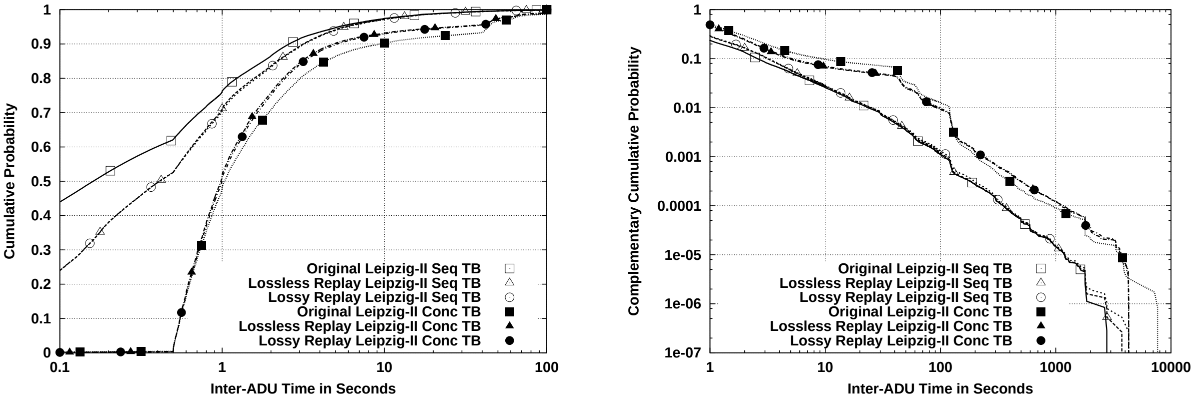


Figure 5.8: Bodies and tails of the TB distributions for Leipzig-II and its source-level trace replays.

therefore makes the shown distributions from the sequential connections in the replays heavier than the one in the original trace.

The distribution TB of quiet times on the initiator side of TCP connections, *i.e.*, between b_i and a_{i+1}, is compared for original and replayed traces in Figure 5.8. The bodies of the distributions show the same kind of mismatch that we discussed for the TA distributions. For values below a few seconds the TB distribution from the replay of sequential connections appears heavier that the original distribution. This is due to the overestimation of quiet times, which becomes less significant as the quiet time becomes larger. We can also observe that the difference in the shortest quiet time is larger for TB than for TA. The reason is not completely clear, but it is probably related to the absence of samples in TB from the large subset of connection vectors with only one epoch. The TB distribution from the replay of concurrent connections appears lighter than the original for values above one second. This is due to concurrent connection misclassification. The much larger number of samples in the distributions for sequential connections makes the impact of the misclassification very small.

Besides the replay of the source-level characteristics of the connections in Leipzig-II, our experiments also involved replaying the network-level parameters measured for each connection in $\mathcal{T}_c$. The left plot in Figure 5.9 compares the distributions of round-trip times extracted from the original and the generated packet header traces. The reproduction was very accurate for sequential connection vectors, and the three distributions exactly lie on top of each other. On the contrary, the distributions for the replayed concurrent connections show a strange jump in probability at 100 milliseconds. The reason for this anomaly, which changed the shape of the rest of the distribution, is unclear.

The right plot of Figure 5.9 compares the distributions of receiver window sizes. Note that the

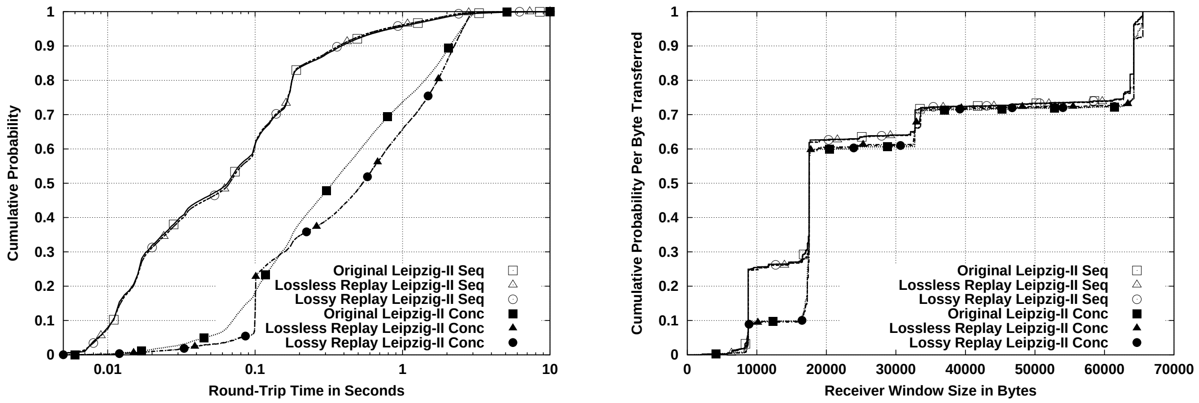


Figure 5.9: Bodies of the round-trip time and receiver window size distributions for Leipzig-II and its source-level trace replays.

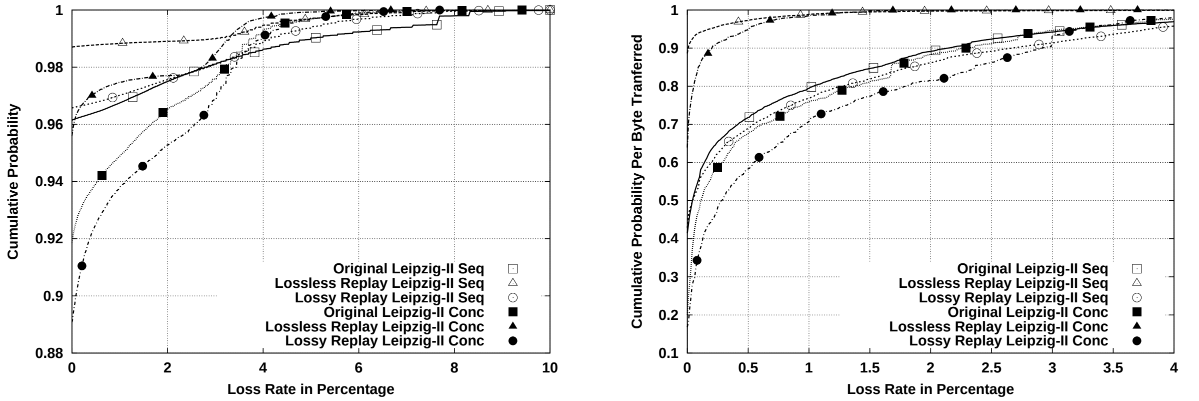


Figure 5.10: Bodies the loss rate distributions for Leipzig-II and its source-level trace replays, with probabilities computed per connection (left) and per byte (right).

probability was computed over the total number of data bytes transferred, to give a better sense of the amount of data associated with each receiver window size. There is an excellent match between the distribution obtained from the Leipzig-II trace and those from its two source-level replays.

The final comparison examines the distributions of loss rates. The left plot of Figure 5.10 shows the distribution of the measured loss rates for the original Leipzig-II trace and its replays. There is a reasonable match between the original and the lossy replays, especially for sequential connections. This is a good result given that *usernet* creates losses by generating random numbers in an independent manner. The small difference is probably explained by a sample size problem in short connections with non-zero loss rates, as discussed in Section 4.1.3, and by concurrent connection misclassification.

Note that we measured some non-zero loss rates in the lossless experiment, in which no artificial losses were introduced. This suggests some problem with the experimental environment, perhaps some network interfaces that were duplicating segments. Such duplicates confuse the loss rate measurement algorithm, which considers each retransmission a loss event[3]. If duplication is behind our observations, the impact on the experiments would be minimal. True loss slows down TCP, but duplication does not.

The right plot of Figure 5.10 shows the distributions of loss rates per byte, rather than per connection as in the left plot. The CDFs show the probability that each byte had of being carried in a connection with at most the given loss rate. For example, the CDFs for the original sequential connections shows that 80% of the bytes were carried in connections with a loss rate of 1% or less. The CDFs in the right plots are easier to read than those in the left plot, since they are far smoother. They are also more significant, since they pay more attention to the connections that carry more bytes, which are those than have a larger impact on the load of the network. There is a good match between loss rate distributions for the original and the lossy replay. Both the distribution from the replayed sequential connections and the one from replayed concurrent connections are slightly heavier than those from the original traces.

In general, we always observe heavier loss rates in the replays than in the original data. The explanation is the dropping of pure acknowledgment packets, which was discussed in Section 4.1.3. The analysis of the original trace considers only the loss rate of data segments, and not the combined loss rate of data and acknowledgment segments. However, the artificial dropping mechanisms in *usernet* that is used to create per-flow losses is applied to all of the packets in the connections. This means that both data segments and acknowledgment segments are dropped according to the original loss rates of data segments. The dropping of acknowledgment segments can increase the loss rate of data segments in the replay, because missing acknowledgments can trigger unnecessary retransmissions. Every retransmission is considered a loss event, and therefore we have an increase of loss rate in the replays, which makes the measured distributions of (data segment) loss rates heavier for the replays than for the original. It is certainly possible to modify *usernet* to apply the dropping rate to data segments only, but our experiments did not incorporate this refinement. It is somewhat unrealistic to use a biased dropping mechanism, so it would be better to refine the data acquisition algorithm to consider both data and pure acknowledgment losses. Measuring pure acknowledgment loss rates is far more difficult that measuring data segment loss rates. Endpoints may acknowledge every data segment, or every other data segment, and they do so using cumulative acknowledgment numbers, rather than individual sequence numbers as it is done for data segments. It is therefore more difficult to determine when an acknowledgment does not arrive as expected.

[3] This approach could certainly be refined using the IP ID field to distinguish duplications from retransmissions.

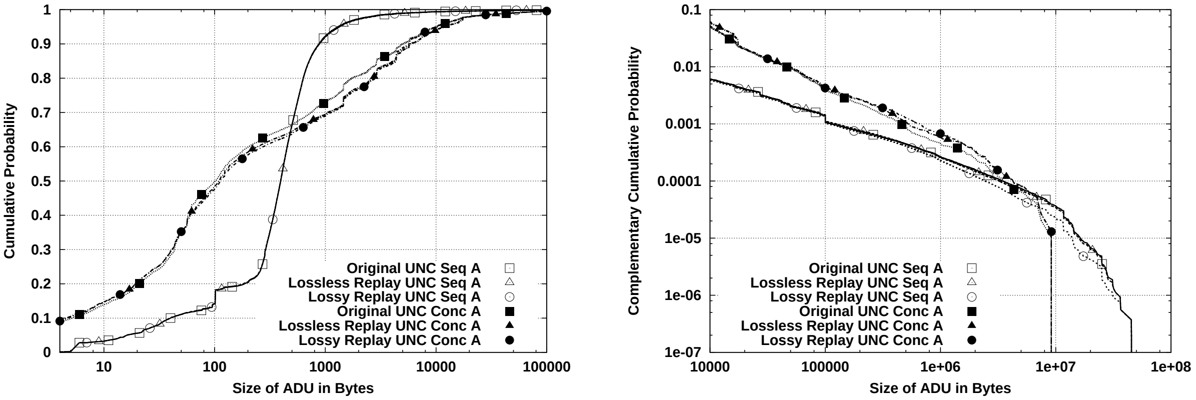


Figure 5.11: Bodies and tails of the *A* distributions for UNC 1 PM and its source-level trace replays.

5.2.2 UNC 1 PM

The second trace considered in our validation of the source-level trace replay approach is the UNC 1 PM trace. This trace is shorter than Leipzig-II (1 hour *vs.* 2 hours and 45 minutes) but it has much higher throughput, which results in a substantially larger number of samples in the distributions that we will examine in this section. Figure 5.11 compares the *A* distributions extracted from the UNC 1 PM and its lossless and lossy replays. The bodies of the *A* distributions from sequential connections reveal no difference between original and generated traces. The tail of the *A* distribution from the lossy replay is slightly lighter than the one from the original trace and the one from the lossless replay. This difference can be attributed to trace boundaries. Losses make the replay of some connections slower, which can easily result in some connections that do not have time to finish during the replay experiment. This effect is more important for the largest data units, those in the tail of the distribution, since they are the ones that require a substantial amount of time to complete their transmission even without losses.

Concurrent connections show a slightly worse match. This is due to the misclassification problem described in the previous section. As pointed out before, misclassifications are more likely to occur in concurrent connections with small ADUs. These connections have a small number of packets, making the observation of concurrent pairs less likely. As a result, the bodies of the distributions from the replays are slightly heavier, since some fraction of the small ADUs disappeared from the *A* distribution for concurrent connections. On the contrary, misclassifications had no visible impact on the *A* distribution for sequential connections. This is because the number of a-type ADUs in sequential connection vectors is much larger than the number of samples from misclassified connections. The tails of the *A* distributions for concurrent connections show a good agreement.

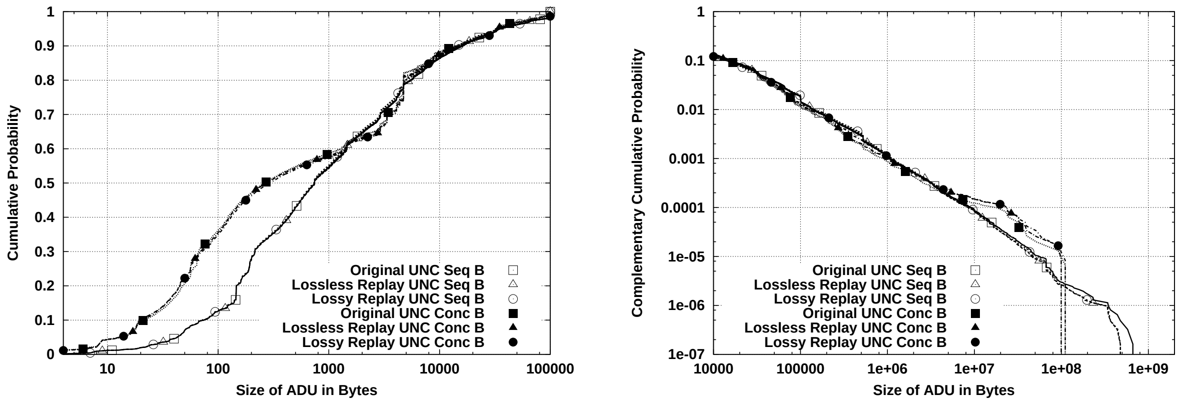


Figure 5.12: Bodies and tails of the B distributions for UNC 1 PM and its source-level trace replays.

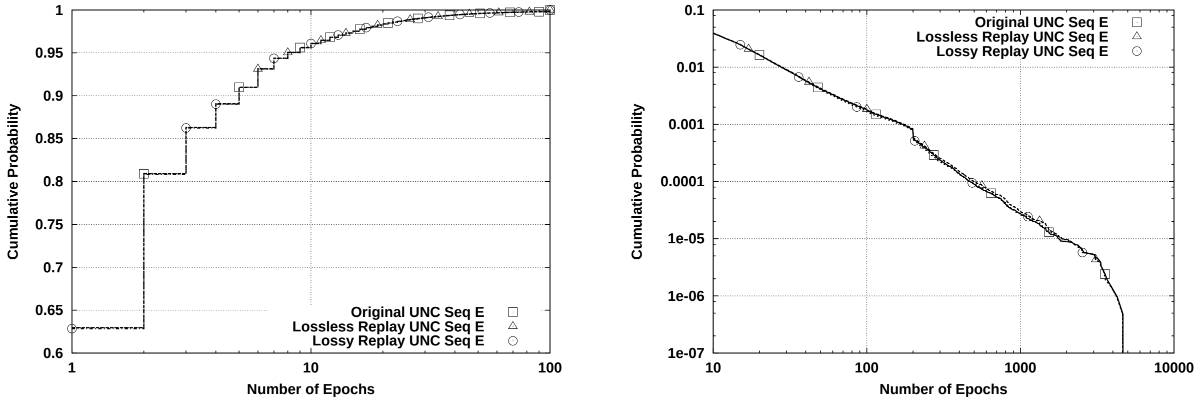


Figure 5.13: Bodies and tails of the E distributions for UNC 1 PM and its source-level trace replays.

The B distributions from the original UNC 1 PM traces and its replays are even closer, as Figure 5.12 shows. We can barely see any differences in bodies of the distributions from concurrent connections and no difference for those from sequential connections. The tails are also very similar, and the slight differences can be explained using the same arguments put forward in the discussion of the A distributions (*i.e.*, trace boundaries and misclassifications).

Figure 5.13 shows an excellent match between the number of epochs in sequential connection vectors measured from the UNC 1 PM traces, and those measured from the replays. The bodies of the distributions are identical, and the tails show only a very minor difference. We therefore observe a better agreement between original and replay for UNC 1 PM than for Leipzig-II (see Figure 5.6).

The plots in Figure 5.14 study the TA distributions. The bodies for sequential connections show an excellent match between the inter-ADU quiet time measured from the original UNC 1 PM trace, and those measured from the generated traces. The bodies for concurrent connections are also very similar.

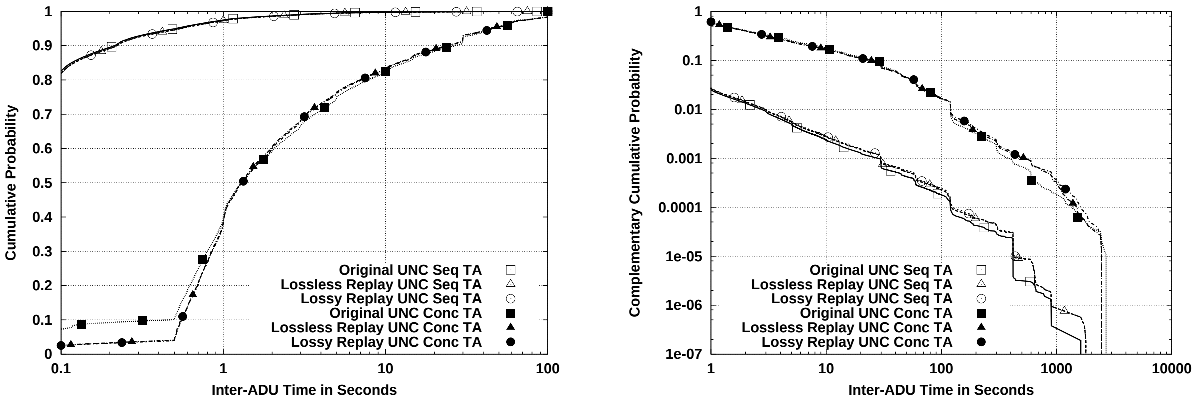


Figure 5.14: Bodies and tails of the TA distributions for UNC 1 PM and its source-level trace replays.

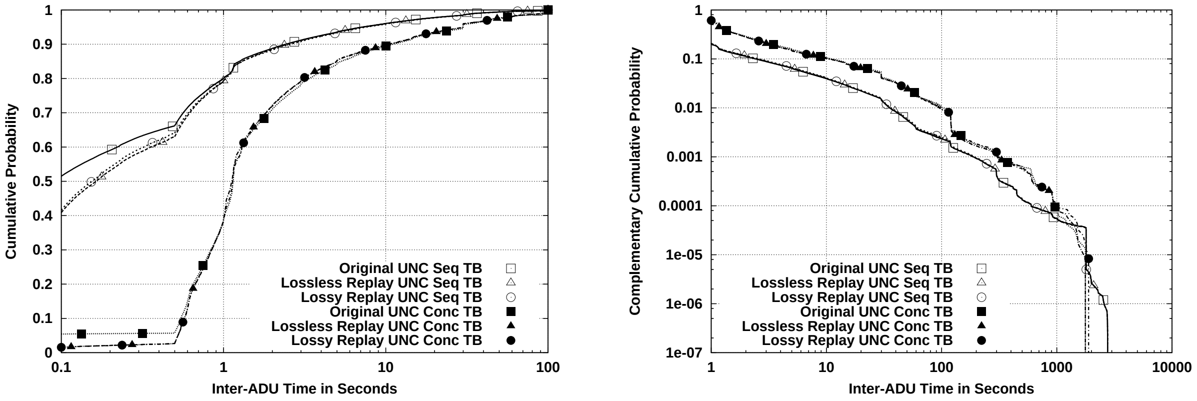


Figure 5.15: Bodies and tails of the TB distributions for UNC 1 PM and its source-level trace replays.

The small difference for the smallest values requires further investigation. We should not see these samples here because our only method for detecting inter-ADU quiet times in concurrent connections is to identify periods of inactivity above 500 milliseconds. We do not observe such a difference for Leipzig-II and Abilene-I. The tails of the distributions are very similar for sequential and concurrent connections. As it was also the case in the data from Leipzig-II shown in Figure 5.7, we observe slightly heavier tails from the replays, which can be explained by misclassifications.

Figure 5.15 shows the bodies and the tails of the TB distributions. Data from sequential connections shows an excellent match for values above 1 second, and even the far tail is very closely approximated. For values below 1 second, we observe that the replays have heavier distributions. This is explained by the quiet time overestimation problem discussed in the analysis of the Leipzig-II results. Concurrent connections also show an excellent match between original and generated traces. The artifact in the smallest inter-ADU quiet times that was observed for the TA distributions from concurrent connections

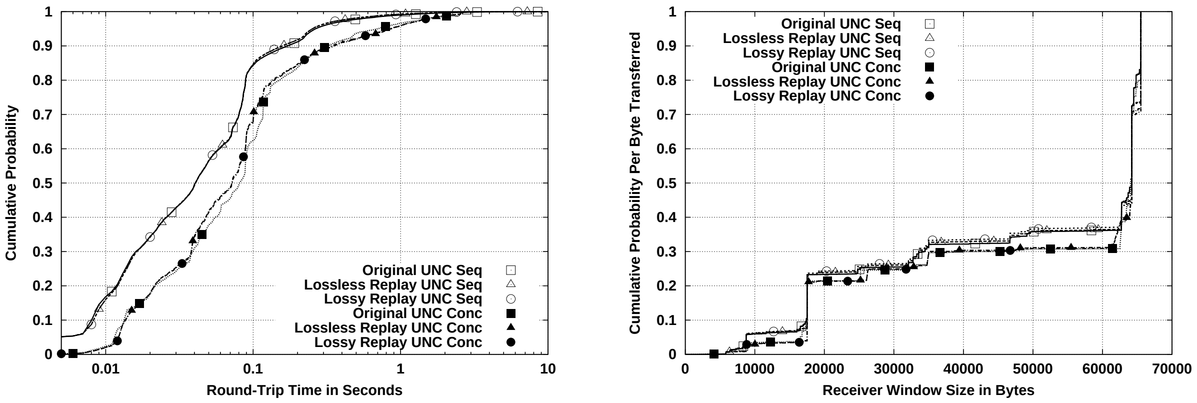


Figure 5.16: Bodies of the round-trip time and receiver window size distributions for UNC 1 PM and its source-level trace replays.

is also present in the TB distributions from concurrent connections.

The next four plots study how closely the replays of UNC 1 PM approximated the network-level parameters observed in the original plot. The left plot of Figure 5.16 shows the distributions of round-trip times. For sequential connections, there was no difference between the round-trip times obtained from the original trace and those obtained from its replays. For concurrent connections, there is only a very small difference, which we can attribute to concurrent connection misclassifications. The large masses of probability for 100 milliseconds observed in the Leipzig-II replays are not present in the UNC 1 PM replays.

Regarding the distribution of TCP receiver window sizes, the plot on the right in Figure 5.16 shows a good match between the original data and the one obtained from the analysis of the generated packet header traces. The tiny difference can again be explained by concurrent connection misclassifications, but it is clear that the replayed traffic accurately captured the use of TCP receiver window sizes.

Figure 5.17 studies the distributions of loss rates rates obtained from original and replayed traffic. As indicated in the analysis of the replays of Leipzig-II, matching loss rate is difficult given the use of independent packet dropping in *usernet* . Consequently, we can consider the approximation of the loss rates shown in the left plot of the figure reasonable, especially in the case of sequential connections, for which many more samples were available. In contrast to these per-connection loss rates, the right plot of the figure shows a substantially closer approximation when loss rate per bytes are considered. Note also that difference between distributions of loss rates for sequential and concurrent connections is far smaller in the case of probabilities per byte.

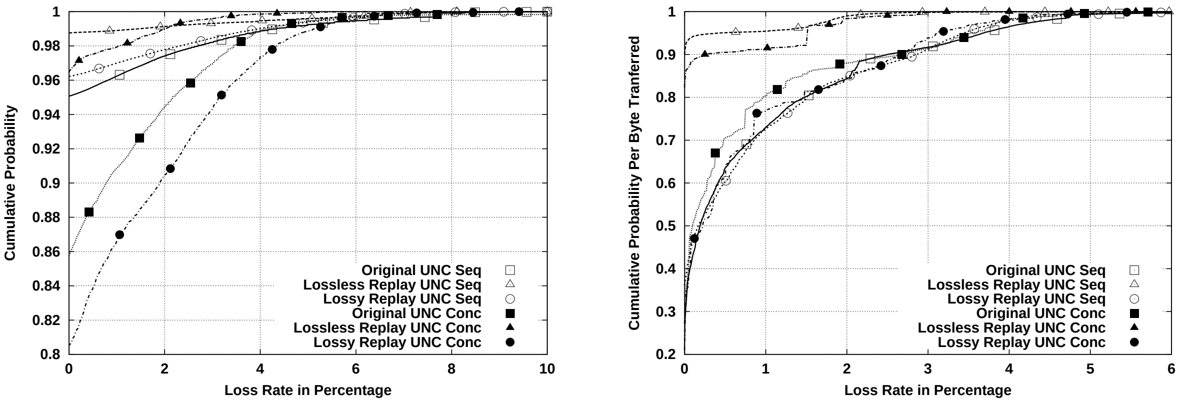


Figure 5.17: Bodies of the loss rate distributions for UNC 1 PM and its source-level trace replays, with probabilities computed per connection (left) and per byte (right).

5.2.3 Abilene-I

We conclude the validation of our source-level trace replay method by comparing the original Abilene-I trace and its lossless and lossy replays. This is the trace with the highest average throughput. Figure 5.18 shows that the A distributions measured from the replayed traces are very similar to those measured from the original trace. Given the completely different A distributions for sequential and concurrent connections, we would expect that any substantial number of misclassified concurrent connections would result in distributions from the replays that significantly diverge from the original distributions. The excellent approximation in this figure, and for the B distributions shown in Figure 5.18, suggest that the number of misclassifications was very small. We also observe a very good match for the tails where the only difference is found for the largest values. In some cases, the replay is slower than the original trace, so some of the largest ADUs may not have had enough time to complete. Adding losses to the replay experiment did not introduce any noticeable difference in the measured distributions, which confirms the robustness of the data acquisition and generation methods to the challenge of lossy environments.

The two plots in Figure 5.19 show that the original distribution of b-type ADU sizes is almost identical to the ones obtained from the lossless and lossy replays. This is true both for the bodies studied in the plot on the left, and for the tails studied in the plot on the right. It is quite difficult to find any region where the distributions differ. It is also clear that the addition of losses to the replay did not modify the sizes of the ADUs in the experiment.

Figure 5.20 shows that the bodies and the tails of the distributions of the numbers of epochs are closely approximated in the source-level replays. There is only a very slight difference in the far tail of the distributions. This could be attributed to a few connections that were replayed more slowly than

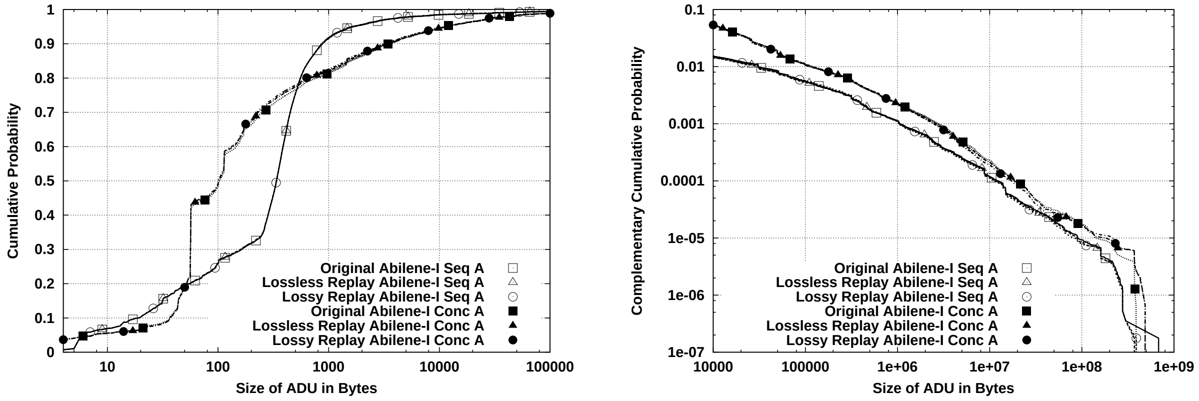


Figure 5.18: Bodies and tails of the A distributions for Abilene-I and its source-level trace replays.

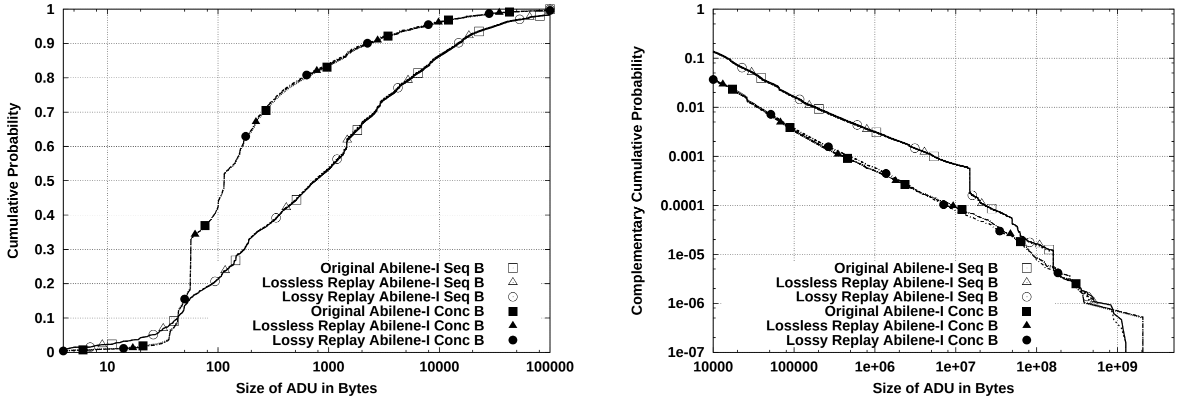


Figure 5.19: Bodies and tails of the B distributions for Abilene-I and its source-level trace replays.

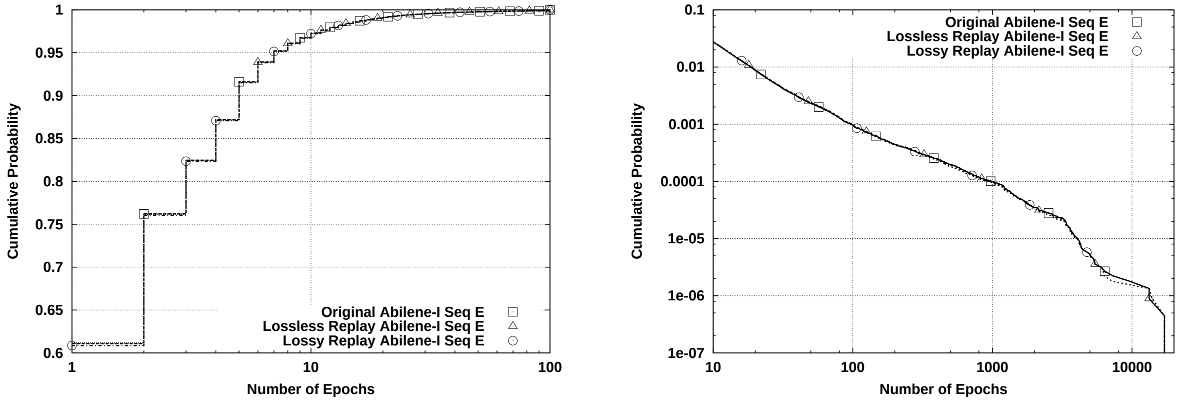


Figure 5.20: Bodies and tails of the E distributions for Abilene-I and its source-level trace replays.

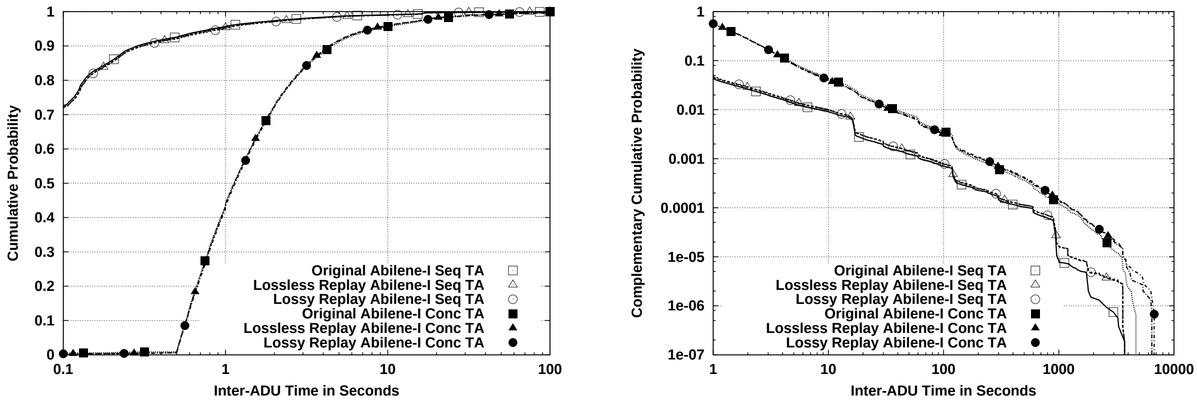


Figure 5.21: Bodies and tails of the TA distributions for Abilene-I and its source-level trace replays.

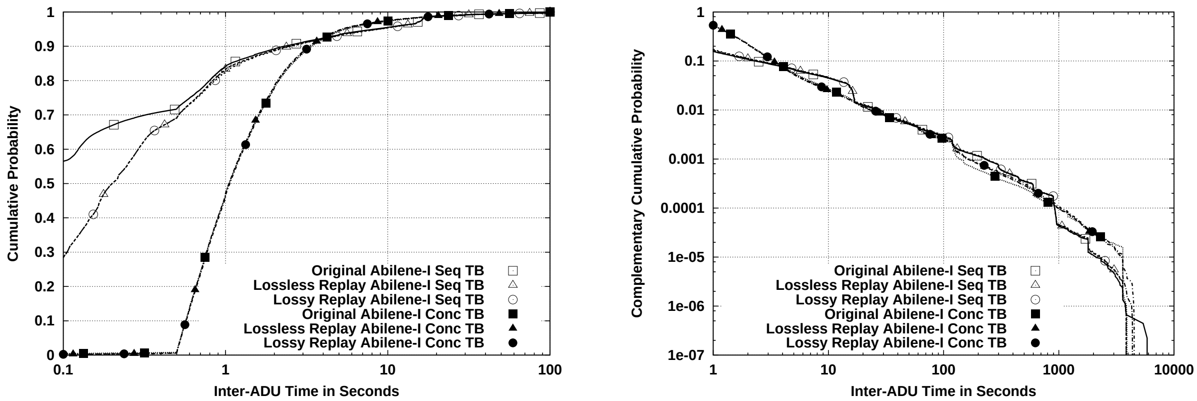


Figure 5.22: Bodies and tails of the TB distributions for Abilene-I and its source-level trace replays.

in the original trace, so they did not have time to complete all of their epochs. Another possibility is that a small number of concurrent connections with a large number of epochs were misclassified. The probabilities in the tail are so small, that even a few samples can create a visible difference.

The quality of the replay of quiet times between ADUs is studied in the next two figures. Figure 5.21 shows that the TA distributions are accurately approximated in the replays. This is true both for sequential and concurrent connections. We only observed a small difference in the far tail, where the replays show slightly heavier values for quiet times above 1000 seconds. As in the case of the E distributions, both experiment boundaries and concurrent connection misclassification can explain the difference.

Figure 5.22 examines the distribution TB of quiet times on the initiator side. As shown on the left, there is an excellent match between the bodies of the distributions from the original trace and those from the replays. The only difference is found in the distributions from sequential connections for quiet times

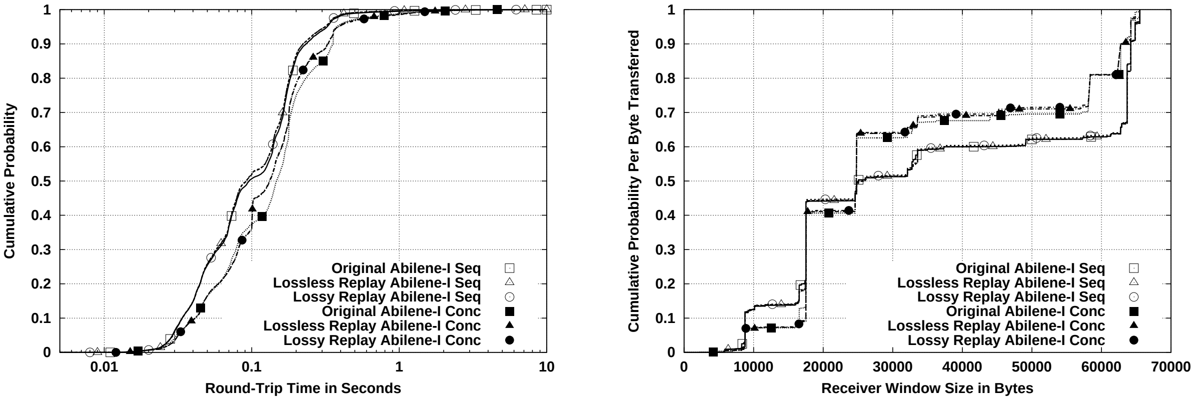


Figure 5.23: Bodies of the round-trip time and receiver window size distributions for Abilene-I and its source-level trace replays.

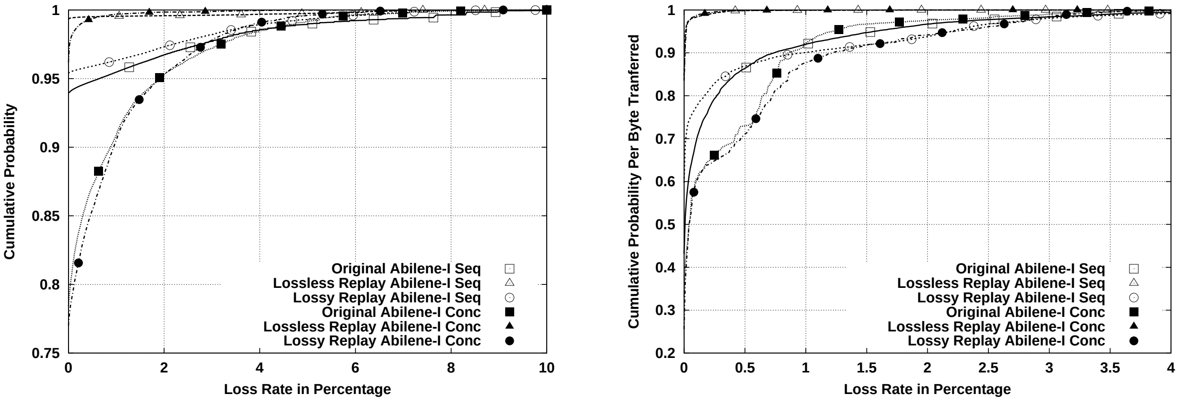


Figure 5.24: Bodies of the loss rate distributions for Abilene-I and its source-level trace replays, with probabilities computed per connection (left) and per byte (right).

below 1 second. The quiet times measured from the replays became increasingly heavier than those from the original trace as their magnitude decreased. This finding is consistent with inaccuracies due to the overestimation of quiet times, since end-host location has a larger impact on the measured quiet time as the magnitude of the application-level quiet time decreases. The tails of the distributions reveal an excellent approximation. It is also important to note that the distributions for concurrent connections do not show the unexpected values below 500 milliseconds that were observed for UNC 1 PM.

The analysis of the round-trip times in Figure 5.23 reveals an excellent match between the original and the replay distributions of round-trip times. The replay of concurrent connections exhibits the same artifact at 100 milliseconds encountered in the replays of the Leipzig-II trace, but the magnitude is far smaller. The distributions of receiver window sizes show very close approximations, with only a small divergence for concurrent connections, which can be easily explained by a small number of misclassifications.

The distribution of loss rates in the lossy replay is very close to the original distribution, as shown in Figure 5.24. The CDFs on the left plot show cumulative probabilities computed per connection, and they reveal a remarkably good match between the original and the lossy replay, both for concurrent and sequential connections. This is significantly better than in the cases of Leipzig-II and UNC 1 PM, which were studied in Figures 5.10 and 5.17. The better match is mostly explained by two characteristics of the original data. First, Abilene-I has the largest fraction of lossy connections, which more than doubles the one in Leipzig-II. This means a wider y-axis that reduces the distance between the distributions in the plot. Second, the heavier distribution of connection sizes in the Abilene-I trace means a larger number of packets, which makes the use of independent drops approximate the intended loss rates more accurately. The right plot shows a good match when the distributions of the per-byte loss rates are considered.

5.3 Summary

This chapter presented our traffic generation method, source-level trace replay. The first step in source-level trace replay is to transform a packet header trace into a set of connection vectors, which describe its source-level behavior using the sequential or the concurrent version of the a-b-t model. Connection vectors also include three network-level parameters, round-trip time, TCP receiver window size and loss rate. The actual traffic generation consists of replaying the characteristics of each connection vector in an accurate manner. We demonstrated the possibility of this approach using an implementation in a network testbed, which includes a distributed traffic generator, *tmix* , that can replay source-level behavior, and coordinate with a packet manipulation layer, *usernet* , to impose specific round-trip times and loss rates to each connection. The approach, and its implementation, was then validated by comparing the statistical characteristics of three traces and those of their replays. This comparison focused on how well the replay preserved the original parameters, *i.e.*, the source-level description and the network-level characteristics.

The validation results showed a good match between original traces and their replays, which confirms the highly accurate reproduction of source-level properties that can be achieved with our approach. The differences, which are shown to be small or nonexistent in every case, are due to the following causes:

- There is no guarantee that the replay of a concurrent connection exhibits measurable concurrency, *i.e.*, that a pair of concurrent data segments can be observed in the generated trace. This results in connections that are replayed as concurrent but classified as sequential in $\mathcal{T}_c'$, therefore adding spurious samples to the characterization of sequential connections, and removing samples from the characterization of concurrent connections. In general, this affects the comparison of concurrent connections more substantially, since the number of samples from concurrent connections is usually far smaller. This problem is inherent to the form of the concurrent a-b-t model used in this dissertation.

- Our measurement of quiet times tended to overestimate their durations, since it did not compensate for the delay between the end host and the monitor. This difference is only significant for the smallest quiet times, whose magnitude is similar to that of network delays. A possible refinement of our measurement method that would eliminate the overestimation of quiet times and make the replay of quiet times even more accurate, is to subtract the corresponding one-side transit time from each measured quiet time.

- *Usernet* uses independent dropping to simulate losses, and this is not completely accurate. Connections often have too few packets to converge to the intended loss rate per connection. If loss rates per byte are considered, the replay is shown to be very close to the original distribution. Achieving a close approximation of the original loss rate would involve some form of dependent dropping.

- Measured drop rates consider only data segments, but the loss rate simulation also drops pure acknowledgments with the same probability. This makes the distributions of loss rates in the lossy replays slightly above the intended values. Addressing this inaccuracy requires developing a measurement algorithm that can determine the loss rate of pure acknowledgments, which seems rather difficult, or modifying *usernet* to drop only data segments, which is a somewhat artificial solution.

The analysis of the validation results also served us to verify the robustness of our data acquisition and generation method to the introduction of losses with regard to the source-level characteristics. We found very little difference, if any, between the results from the lossless and lossy replays, which confirms the accuracy of the analysis even in the face of packet losses and reordering. TCP timeouts, which can sometimes confuse the heuristic used to split ADUs in the same direction, do not appear to have any significant effect.

CHAPTER 6

Reproducing Traffic

Sometimes the appropriate response to reality is to go insane.

— Philip K. Dick (1928–1982), Valis

Dissertations are not finished; they are abandoned.

— Frederick P. Brooks, Jr. (1931–)

This chapter examines the statistical characteristics of source-level trace replay experiments, comparing them to those of their corresponding original traces. As discussed in Chapter 5, and illustrated in Figure 5.1, a packet header trace $\mathcal{T}_h$ and its source-level trace replay can be compared at two levels. The first level is how well the set of connection vectors $\mathcal{T}_c$ extracted from $\mathcal{T}_h$ are preserved by the trace replay experiments. This means to collect a packet header trace $\mathcal{T}_h'$ from the replay and extract a new set of connection vectors $\mathcal{T}_c'$. Section 5.2 presented a comparison of $\mathcal{T}_c$ and $\mathcal{T}_c'$ for three traces. It demonstrated that the characteristics of $\mathcal{T}_h$ captured by $\mathcal{T}_c$ are accurately reproduced by the traffic generation method and its implementation. The second level at which traces and their replays can be compared is to directly extract statistics from $\mathcal{T}_h$ and $\mathcal{T}_h'$. If these statistics are *reasonably* close, we can say that the traffic generation method *reproduces* the original traffic using closed-loop traffic generation. This is the type of comparison discussed in this chapter. As we will show, source-level trace replay generally results in a good approximation of the statistical characteristics of the original traffic, which supports the use of the a-b-t model as a foundation for realistic traffic generation.

6.1 Beyond Comparing Connection Vectors

The main goal of this dissertation is to improve the state-of-the-art in closed-loop traffic generation by developing a better approach to source-level modeling. In particular, we presented in Chapter 3 the sequential and concurrent versions of the a-b-t model, which provide a first method for describing source-level behavior in an application-independent manner. We also discussed an efficient data acquisition algorithm for extracting a-b-t connection vectors from the packet headers of TCP connections. The first way in which we justified our source-level model was by examining connections from different applications, and demonstrating that their source-level descriptions in terms of a-b-t connection vectors properly captured their source-level behavior. The second way in which we can justify the model is to study the traffic generated using this model. If generated traffic is shown to closely approximate original traffic, this would strongly support the claim that the a-b-t model is a good description of source behavior. In other words, given that the statistical characteristics of $\mathcal{T}_h$ are obviously a function of source behavior, being able to generate a $\mathcal{T}'_h$ statistically similar to $\mathcal{T}_h$ would confirm the quality of $\mathcal{T}_c$ as a description of the original source behavior.

Comparing $\mathcal{T}_h$ and $\mathcal{T}'_h$ is however a subtle exercise. The actual replay of $\mathcal{T}_c$ necessarily requires choosing a set of network-level parameters, such as round-trip times and TCP receiver window sizes, for each TCP connection in the source-level trace replay experiment. The exact set of packets and their arrival times is a direct function of these parameters, as explained in Chapter 4. As a consequence, if we were to conduct a source-level trace replay using arbitrary network-level parameters, we would obtain a $\mathcal{T}'_h$ with little resemblance to the original $\mathcal{T}_h$. The replayed a-b-t connection vectors may be a perfect description of the source behavior driving the original connections, but the generated $\mathcal{T}'_h$ would still be very different from the original $\mathcal{T}_h$. To address this difficulty, the replay should incorporate network-level parameters individually derived from $\mathcal{T}_h$ for each connection. In Chapter 4, we described and evaluated methods for measuring three important network-level parameters: round-trip time, TCP receiver window size and loss rate. While this set of parameters is by no means complete, it does include the main parameters that affect the average throughput of a TCP connection, [PFTK98]. In this chapter, we examine the results of several source-level trace replay experiments, showing that the generated traffic is remarkably close to the original traffic. This is a strong justification of our source-level modeling approach, since it demonstrates that the closed-loop replay of a-b-t connection vectors provides a good approximation of the original traffic.

Incorporating network-level properties is important, but it is critical to understand the main shortcoming of this approach. The goal of our work is not to make the generated traffic $\mathcal{T}'_h$ identical to the original traffic $\mathcal{T}_h$, which could be accomplished with a simple packet-level replay. The goal is to develop a closed-loop traffic generation method based on a rich characterization of source behavior. Comparing $\mathcal{T}_h$ and $\mathcal{T}'_h$ is a means to understand the quality of traffic generation method, where quality is considered to be higher as the original trace is more closely approximated. By construction, traffic generated using source-level trace replay can *never* be identical to the original traffic. The statistical properties of original packet header traces are the result of multiplexing a large number of connections into a single link, and these connections traverse a large number of different paths with a variety of network conditions. It is simply not possible to fully characterize this environment and reproduce it in a laboratory testbed or in a simulation. This is both because of the limitations of passive inference from packet headers, and because of the stochastic nature of network traffic. Source-level trace replay can never incorporate every factor that shaped $\mathcal{T}_h$, and therefore differences between $\mathcal{T}_h$ and $\mathcal{T}'_h$ are unavoidable. Still, finding a *close* match between an original trace and its replay, even if they are not identical, constitutes strong evidence in favor of our a-b-t model and our data acquisition and generation methods. It also demonstrates the feasibility of generating realistic network traffic in a closed-loop manner that resembles a rich traffic mix.

Besides evaluating source-level trace replay by comparing original traces and their replays, this chapter also considers whether *detailed* source-level modeling is necessary to achieve high-quality traffic generation. This is accomplished by comparing traffic generated using $\mathcal{T}_c$ (*i.e.*, replaying connection vectors and network-level parameters) and traffic generated using a simplified version of $\mathcal{T}_c$ with *collapsed epochs*, which we will name $\mathcal{T}_c^{coll}$. Formally, given a sequential connection vector $C_i = (e_1, e_2, \ldots, e_n)$, $n \geq 1$, with epoch tuples of the form $e_j = (a_j, ta_j, b_j, tb_j)$, we define the version of C_i with collapsed epochs as

$$C_i^{coll} = ((\sum_{i=1}^{n} a_i, 0, \sum_{i=1}^{n} b_i, 0)).$$

The only a-type ADU size in the resulting connection vector is the total amount of data sent from the connection initiator to the connection acceptor, and the only b-type ADU size is the total amount of data sent from the connection acceptor to the connection initiator. No quiet time is part of a connection vector after collapsing its epochs. Similarly, given a concurrent connection vector $C_k = (\alpha, \beta)$, where

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

and

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

we define the version of C_k with collapsed epochs as

$$C_k^{coll} = ((\sum_{i=1}^{n_a} a_i, 0), (\sum_{i=1}^{n_b} b_i, 0)).$$

Traffic generated according to $\mathcal{T}_c^{coll}$ does not incorporate any internal source-level structure of connections, *i.e.*, epochs and inter-ADU quiet times are ignored. For this reason, we say that the collapsing of epochs "removes" detailed source-level modeling. Note however that even if epochs are collapsed, the total amount of data transferred in each direction does not change. The results in this chapter demonstrate that traffic generated using $\mathcal{T}_c$ is substantially closer to the original traffic than traffic generated using $\mathcal{T}_c^{coll}$.

The evaluation of source-level trace replay presented in this chapter examines the results of replaying five traces. These traces were first considered in Section 3.5: Leipzig-II, UNC 1 PM, UNC 1 AM, UNC 7:30 PM and Abilene-I. Our analysis compares the statistical characteristics of each of these traces and their replays using the following metrics:

- time series of byte throughput,
- time series of packet throughput,
- Body and tail of the marginal distribution of byte throughput,
- Body and tail of the marginal distribution of packet throughput,
- Wavelet spectrum (logscale diagram),
- Estimated Hurst parameter and its confidence interval, and
- time series of the number of active connections.

These metrics were introduced in Section 4.2. For each original trace, we compare four different replays, conducted using *tmix* and *usernet* in the testbed shown in Figure 5.2. The first replay is the *lossless replay*, which replayed the a-b-t connection vectors in $\mathcal{T}_c$, giving each TCP connection its measured round-trip time and TCP receiver window sizes. The second replay is the *lossy replay*, which was identical to the first one, but it also applied random packet dropping to each TCP connection according to its measured loss rate. The third replay, is the *lossless replay with collapsed epochs*, which replayed the a-b-t connection vectors after they had their epochs collapsed, and it also gave each connection its measured round-trip time and TCP received window sizes. The fourth replay is the *lossy replay with collapsed epochs*, which was identical to the third one but incorporated loss rates. We will often refer to the first two replays as *full replays* and to the second two replays as *collapsed-epochs replays*.

It is important to note that our method for incorporating losses into the experiments, random dropping according to the measured probability of loss per connection, is not consistent with closed-loop traffic generation. We are by no means suggesting that loss rates should be incorporated in this manner into regular networking experiments that require closed-loop traffic generation. In such experiments, losses should only be the result of congestion on network links and buffering limitations. If this is the case, the endpoints generating synthetic traffic can not only react to the network conditions (*e.g.*, reducing sending rates when congestion is detected), but also modify them (*e.g.*, reducing overall congestion thanks to the lower sending rates). This is the right approach to reproduce the essential feedback loop in TCP which should be used in empirical studies of TCP performance.

However, loss is an important factor in TCP behavior (see Section 4.1.3), so our lossy experiments should result in a $\mathcal{T}'_h$ that is closer to the original $\mathcal{T}_h$. By incorporating losses, we eliminate one possible cause of divergence between original and replayed traces which could confuse our assessment of our source-level modeling approach. Comparing lossless and lossy replays enables a more systematic evaluation of our traffic modeling and generation methods, and it also helps to understand the impact of loss rates on the generated traffic. Losses are shown to have only a minimal effect on some traces and for some metrics, but a much more substantial effect on others.

The analysis in this section confirms the high-quality of the synthetic traffic generated using source-level trace replay. Our analysis reveals some (mostly minor) differences between original traffic and replay traffic. While we put forward some hypotheses about the cause of these differences, their confirmation requires further analysis. This additional work, which would involve both analysis and experimentation, would certainly be enlightening. It would tell us more about the limitations of our approach, and even about the inherent limitations of testbed experimentation. However, we have chosen not to pursue this avenue here. As discussed above, our goal is not to generate a $\mathcal{T}'_h$ equal to $\mathcal{T}_h$, but to convincingly demonstrate the benefits of our closed-loop traffic generation method. We believe this chapter achieves this goal, so we do not present any further analysis beyond the comparison of five traces and their four types of source-level replays using a rich set of metrics.

6.2 Source-level Replay of Leipzig-II

6.2.1 Time Series of Byte Throughput

The first trace we consider in this chapter is Leipzig-II. It has a duration of 2 hours and 45 minutes, and its average throughput is relatively low. We will first consider the traffic received by Leipzig's hosts

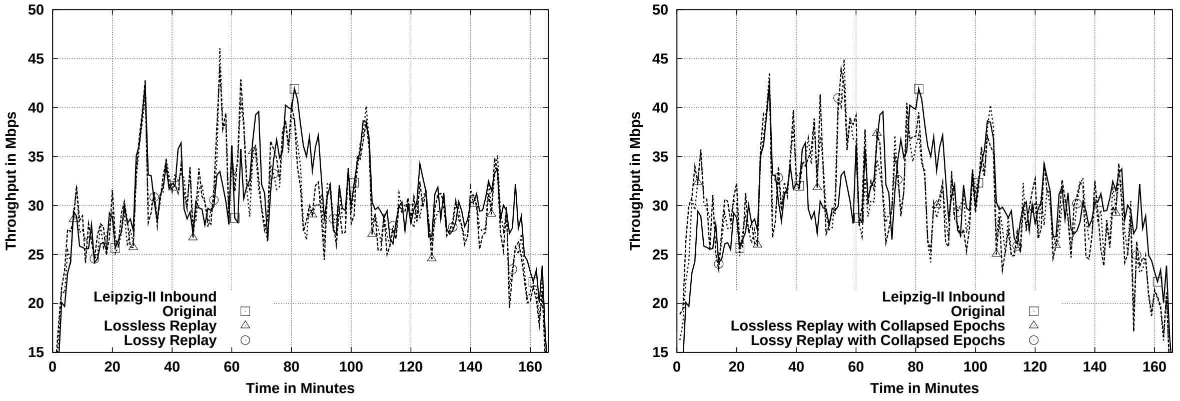


Figure 6.1: Byte throughput time series for Leipzig-II inbound and its four types of source-level trace replay.

from Internet hosts, *i.e.*, in the inbound direction with respect to the University of Leipzig. Figure 6.1 compares the original time series of byte throughput (solid line) and four different source-level trace replays (dashed lines). The plot on the left shows the full replays of $\mathcal{T}_c$ with and without imposing loss rates using *usernet* . The plot shows that the original time series is highly bursty[1], even when 1-minute bins are considered. Both replays closely approximate the original traffic, showing a strikingly good match in most regions. It also shows very little difference between lossless and lossy replays. This suggests that losses had a very moderate impact in the original trace, at least regarding the time series of byte throughput.

We also observe in the left plot of Figure 6.1 several major throughput spikes, *e.g.*, in minutes 25 and 105, that are very closely approximated by both replays. It is clear that the source-level nature of these spikes was accurately captured by our modeling approach. In a few other regions, the original and the replayed traces do not match so well. We have for example a spike in the throughput of the replays in minute 55 that was not present in the original traffic. This suggests that, for some number of connections active in that region of the trace, our model did not capture a significant limitation of throughput that was present in the original trace. This limitation could be at the source level or at the network level, but there is no way to know without further analysis. Given our traffic generation methods, we can however say that loss is very unlikely to be behind this difference, since both lossless and lossy replays show the same spike. We can also observe the opposite phenomenon in several locations, such as minutes 90 and 152, were we find ditches in the throughput of the replays. Here our measurement and modeling approach seems to be imposing an artificial limitation to the throughput of one or more connections.

[1] The term *bursty* does not have a unique meaning. In this paragraph, it simply refers to high variability. Some authors consider traffic more bursty as its long-range dependence becomes stronger [WP98], while others as its marginal distribution becomes less Gaussian [SRB01]. We make use of these more formal definitions, discussed in Chapter 4, in later sections.

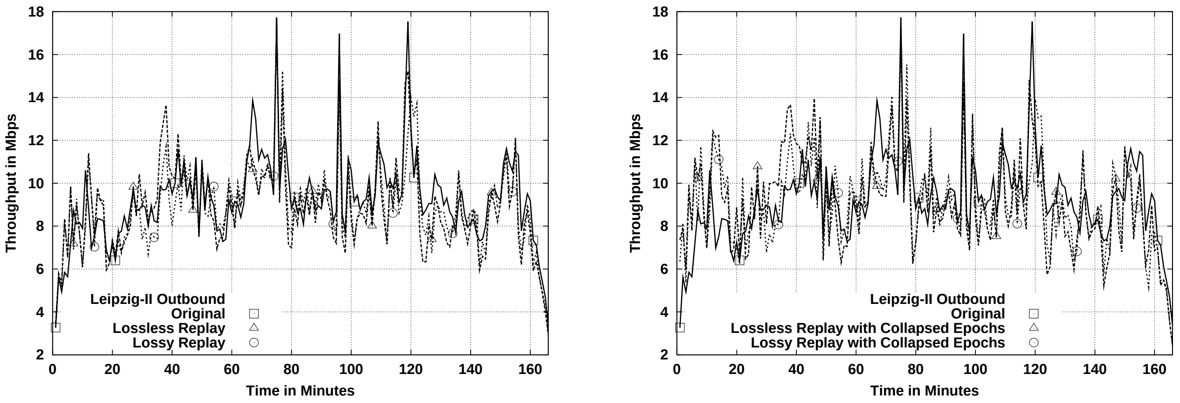


Figure 6.2: Byte throughput time series for Leipzig-II outbound and its four types of source-level trace replay.

While this suggests that further refinement is possible, the plot clearly shows that our approach result in an excellent approximation of the original byte arrival process and its overall burstiness.

The right plot of Figure 6.1 compares the original time series of byte throughput and the ones from the lossless and lossy replays with collapsed epochs. The approximation is also generally good, but the replays appear more bursty, which seems rather significant given the high level of aggregation (1-minute bins). The replays with collapsed epochs results in several new spikes in which the replay is well above the original throughput. This means that removing the source-level structure enabled artificially higher throughputs for some number of replayed connections. Despite these difficulties, it is important to note that the collapsed-epochs replay achieves a reasonably good approximation of original throughput with a much simpler source-level model. The collapsed-epochs replays could then be sufficient for some kinds of experimental studies in which only a good reproduction of the time series of byte throughput is required.

The time series of byte throughput in the outbound direction is studied in Figure 6.2. The comparison of the original and the full replays is found in the left plot. As we observed for the opposite direction, the time series from the replays closely track the original one, and losses do not have a significant impact. We find a number of sharp spikes and ditches from the original traffic that are well reproduced by the replays, *e.g.*, minutes 88, 97 and 143. We also find some artificial ones not present in the original, notably the spike in the replay on minute 38 and the ditch around minute 70. The right plot compares the original and the collapsed-epochs replays, which are again shown to be somewhat more bursty that the full replays throughout the trace.

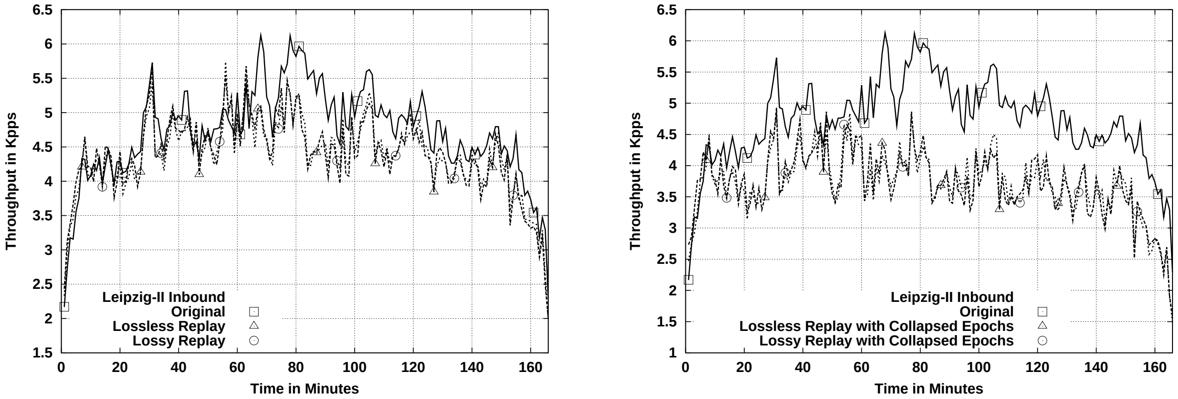


Figure 6.3: Packet throughput time series for Leipzig-II inbound and its four types of source-level trace replay.

6.2.2 Time Series of Packet Throughput

The analysis of the time series of packet throughput reveals larger differences between original and replayed traffic. Figure 6.3 shows the time series for the inbound direction. The comparison of the time series from the original trace and those from the full replays reveals a close approximation for the first 60 minutes, and a consistently lower packet throughput for the rest of the trace. The replays generally have between 2% and 5% less packets per 1-minute bin that the original trace, although they mostly track the original shape. The right plot shows that the collapsed-epochs replays result in far lower packet throughput for the entire trace, between 20% and 40% below the original. This clearly shows that the detailed modeling of source-level structure accomplishes a more realistic traffic generation in terms of the number of generated packets. The main reason is the modeling of epochs, which often increases the number of segments per connection. Replaying an epoch with non-zero ADU sizes necessarily involves sending two packets, even if the sizes of the ADUs are very small. An epoch involves a necessary exchange of data, so at least one packet is used to carry the ADU a_i from the initiator to the acceptor, and another one to carry the ADU b_i from the acceptor the initiator. This means for example that a connection with 10 epochs, and ADUs with a size of 100 bytes in both directions requires 20 packets to be fully replayed. On the contrary, the collapsed-epochs version of this connection can be replayed with a single pair of packets, since the 10 ADUs in each direction can fit into a single TCP segment (it is only 1,000 bytes). Another reason for the more realistic time series of packet throughput when the full replay is used is the modeling of quiet times. Quiet times between two ADUs sent in the same direction (see Section 3.1.2) can also result in a larger number of packets per connection, since they often prevent consecutive small ADUs from sharing packets.

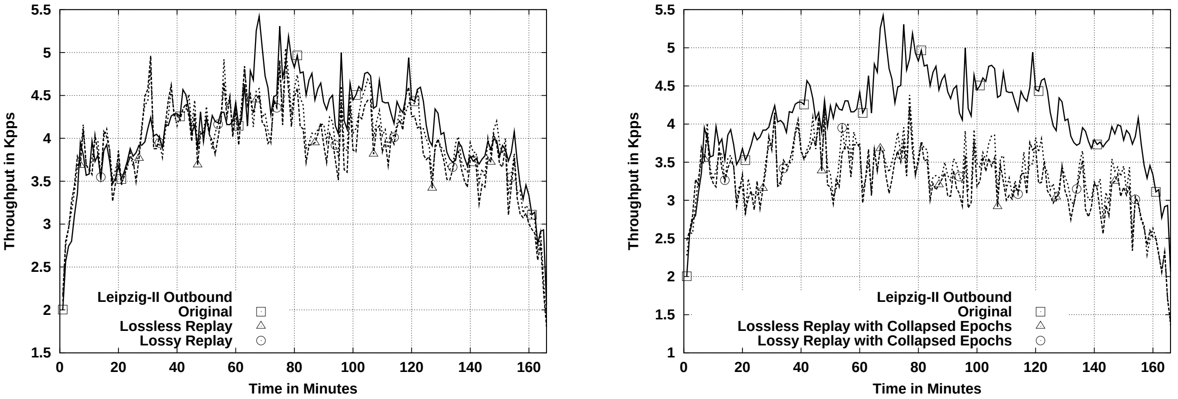


Figure 6.4: Packet throughput time series for Leipzig-II outbound and its four types of source-level trace replay.

While the results in Figure 6.3 convincingly demonstrate a substantially more realistic traffic generation with the full model, there is still some room for improvement. We can think of several possible refinements, which should improve the approximation. First, we made no attempt to model the Maximum Segment Size (MSS) supported by the path of each TCP connection. Instead of relying on the default size derived from Ethernet's MTU (1,500 bytes), as we do in our experiments, it seems possible to collect MSS information for each connection and extend *tmix* to make use of these measurements[2]. Connections replayed using smaller MSS values would frequently require more packets to be replayed. Second, the measurement techniques we used to determine ADU boundaries for data sent in the same direction rely on a constant inter-ADU quiet time threshold equal to 500 milliseconds. Some applications may be using smaller quiet times between their writes, which could result in a larger number of packets per connection. Simply reducing the threshold is problematic, since this would increase the number of spurious splits of ADUs due to network delays (rather than application behavior). To avoid this, we could make the inter-ADU quiet time threshold a multiple of the measured round-trip time. Given the typical distributions of round-trip times (see Section 4.1.1), this method would reduce the threshold for most connections and increase the sensitivity of the measurements. Another approach is to study segment sizes, using non-full segments to mark ADU boundaries. This would require some further refinement, since non-full segments can easily come from application writes which are not a multiple of the MSS. Two consecutive non-full segments are for example far more likely to mark a true ADU boundary.

The lesson is similar for the outbound direction results, which are shown in Figure 6.4. The left plot

[2] MSS is a system-wide constant in FreeBSD, so generating traffic that preserves per-connection MSS is not directly possible with our current implementation. However, there is a relatively simple way to extend our method to support per-connection MSS values. We could use a first step to group connections with the same MSS and then assign each group to a host configured with that MSS. Fortunately, only a few MSS values are common on the Internet, so it seems feasible to implement this extension without increasing the number of hosts.

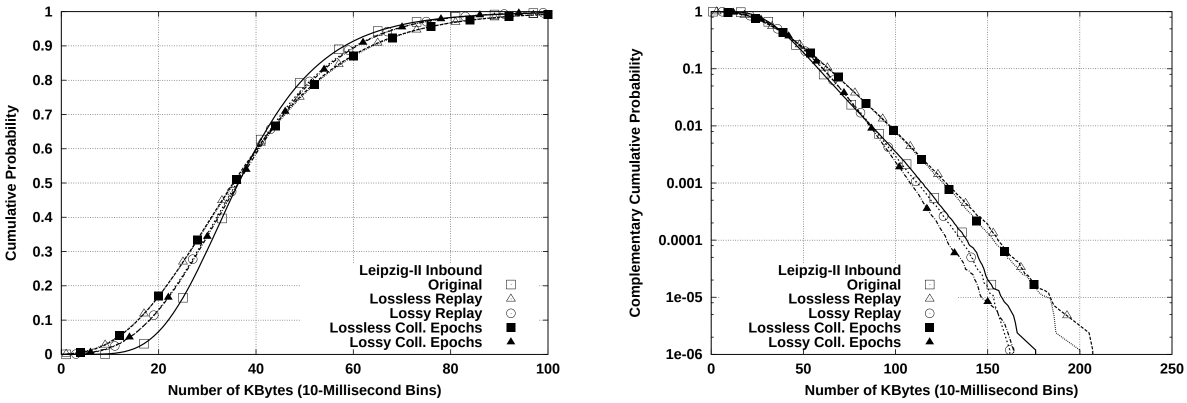


Figure 6.5: Byte throughput marginals for Leipzig-II inbound and its four types of source-level trace replay.

shows that the full replays are generally a good approximation to the original, but they exhibit a somewhat lower number of packets in some regions. On the contrary, collapsed-epochs replays consistently show a far lower number of packets.

The reader may be puzzled by the finding of very similar shapes for the inbound and outbound time series of packet throughput, which show spikes and ditches located at the same minutes. This is due TCP's acknowledgment mechanism, which forces TCP endpoints to at least send one acknowledgment for each pair of data segments received. As consequence, a connection that sends a large number of data segments in one direction, creating a spike in the time series, must necessarily receive a large number of acknowledgments in the opposite direction, creating a similar spike.

6.2.3 Marginal Distributions

One important limitation of the type of analysis in the previous section is the use of a relatively coarse level of aggregation (1-minute bins). The obvious question is whether the close match between original traffic and its source-level replays is also found at finer scales, which are arguably more important for some kinds of studies, such as router queuing evaluation. Given the highly bursty nature of the throughput time series, simply plotting the time series at finer levels of aggregation just makes the plots completely unreadable. In this section, we rely on a different kind of analysis to examine the difference between original and replayed traffic at a finer level of aggregation. Instead of the 1-minute bins used in the previous section, this section examines throughput using CDFs of the marginal distributions extracted from time series of 10-milliseconds bins. Section 4.2.2 further discusses the reasoning behind this type of analysis.

Figure 6.5 plots the marginal distributions of the byte throughput in the inbound direction, showing the data for the original time series and the four types of replay. The left plot shows the body of the marginal distributions using CDFs in linear axes. The right plot shows the tail of the marginal distributions using CCDFs in a logarithmic y-axis. The plot of the tail provides information about the 10-millisecond bins with the highest throughput, giving us a better sense of how well the most "aggressive" regions (*i.e.*, with the highest throughput) of the time series are reproduced by the replays. The vast majority of the plot comes from throughputs that are relatively uncommon, *e.g.*, half of the plot shows data from only 0.1% of the distribution. On the contrary, the plot of the body provides information about the most common bins, showing the entire distribution without focusing on any particular region. These two visualizations are complementary. The body plot shows the overall match, which is relevant for experiments in which producing a realistic range of fine-scale throughputs is important. The tail plots shows the extremal match, which is relevant for experiments in which reproducing the magnitude and frequency of peak throughputs is important. None of these plots says anything about the dependency structure of the time series, which is important and that we study in a later section using wavelets. While wavelets are a powerful analysis tool, marginals are far easier to interpret in networking terms.

The left plot shows the original data using a solid curve marked with white squares, and the replay data using dashed curves. The full replay experiments are marked with white symbols, and the collapsed-epochs replay experiments with black symbols. We can make several observations about this plot. The position of the original curve with respect to the replay curves defines two different regions in the plots. Below 40 KB, the distribution from the original data is slightly heavier than those from the replays. Above 40 KB, the distribution is slightly lighter. This means that the replays tended to be less concentrated around the central value than the original data, For example, the number of bins with 10 KB is negligible in the original data, but corresponds to between 2% and 5% of the bins in the replays. We could therefore say that the replays are somewhat more bursty, in the sense that we find more bins with small values and more bins with large values in the CDFs from the replays than in the CDFs from the original data. The exact reason is unclear, but we can make a hypothesis. We know from the previous section that the total number of bytes is similar in original and replay time series. This means that the presence of a larger number of bins with more bytes in the replay must necessarily be accompanied by a larger number of bins with fewer bytes to compensate. Connections in the replay are exposed to more homogeneous delays (primarily because round-trip times are fixed), which gives replayed connections a chance to achieve higher throughput. In the aggregate, and when considering such fine scales, the presence of one or a few replay connections with higher throughput than originally observed creates bins

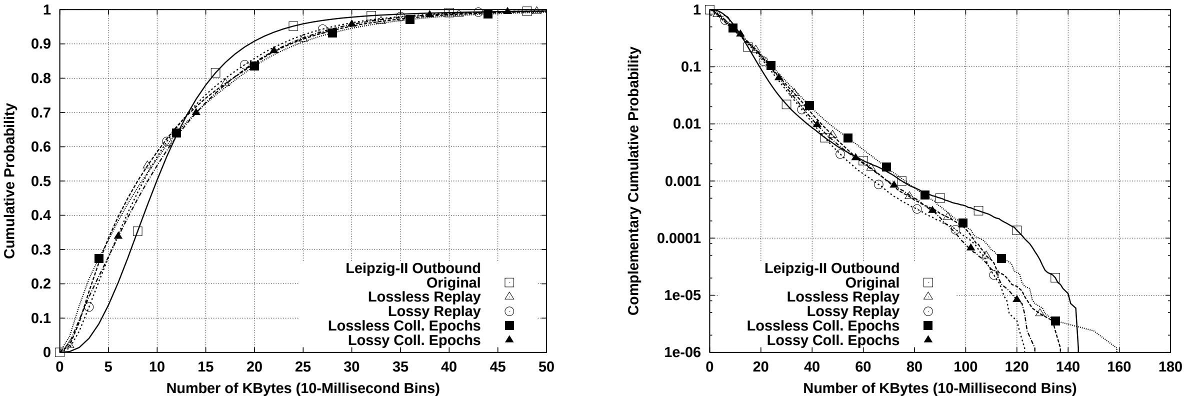


Figure 6.6: Byte throughput marginals for Leipzig-II outbound and its four types of source-level trace replay.

with more bytes, which are part of the upper portion of the body of the marginal distribution. Faster connections run out of data sooner, in turn creating bins with fewer bytes than originally observed, which show up in the lower portion of the body of the marginal distribution. Therefore, the somewhat milder conditions in the replay can explain the wider spread of marginal distributions from the source-level trace replay experiments.

Another observation from the plot of the bodies is that the collapsing of the epochs of the replayed connection vectors has no effect on the marginal distribution of byte throughput. This is an interesting finding, given that we did find a difference for the plots in Figure 6.1. It means that the slightly more bursty replays with collapsed epochs come from a less realistic correlation structure rather than from a fine-grain difference in the values of the bins. The plot also shows that the distributions from the lossy replays are slightly closer to the original than those from the lossless ones. This is evidence in support of the statement in the previous paragraph regarding the impact of more complex network dynamics, which make the highest throughput of many connections lower in the original trace. Adding losses has precisely this effect, making the marginal distributions from the replays closer to the marginal distribution from the original.

The analysis of tails in the right plot confirms the last observation. The plot of the body shows a lighter second half of the distribution. The plot of the tails shows heavier tails from the lossless experiments, and slightly lighter tails from the lossy experiments. The tail from the lossy full replay is actually an excellent fit of the original data. Lossless replays gave some connections the opportunity to reach higher throughputs, which in turn created bins with a larger number of bytes than in the original. Adding losses avoided this problem. In general, the results in Figure 6.5 are very reassuring.

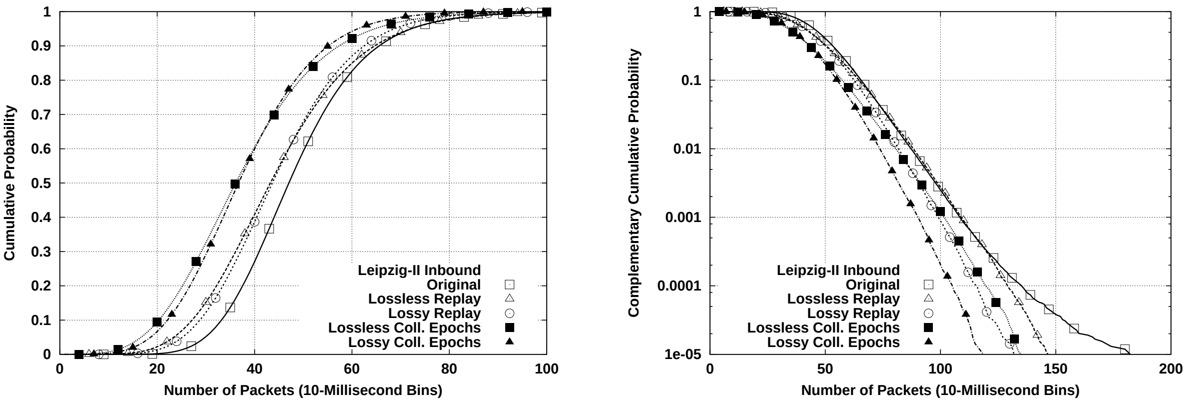


Figure 6.7: Packet throughput marginals for Leipzig-II inbound and its four types of source-level trace replay.

The marginal distributions for the time series of byte throughput in the outbound direction are shown in Figure 6.6. The bodies of distributions (left plot) exhibit a substantial tail, which makes them less Gaussian than distributions from the inbound data. As in the previous case, the range of bin sizes with a significant number of samples is wider for the replays than for the original. The relative difference seems slightly larger in this case, although the absolute difference is of the same magnitude. Lossy replays are again slightly closer to the original.

The tails of the marginal distributions shown in the right plot are not as close to a straight line as those found for the inbound direction. The shape of the tail is most complex for the original data, especially in the region above 90 KB. All of the replays achieve a good match below 90 KB, but are substantially lighter than the original above that value. The reason is unclear. The different shape can easily be due to the characteristics of a few connections (given the very small probabilities considered). The four replays result in similar tails.

As in the previous section, we follow our analysis of byte throughput with an analysis of packet throughput. The marginal distributions for the inbound direction are shown in Figure 6.7. The comparison of the bodies reveals a quite different result for packet throughput. In general, the distributions from the replays are significantly lighter than the distribution from the original. The difference is far larger for the collapsed-epochs replays. The reason was already discussed in the previous section. Collapsing epochs can often reduce the number of segments in a connection, since it enables connections to combine small ADUs from different epochs into a single ADU, increasing packet utilization. Our full replay, while much closer than the collapsed-epochs replay, is still lighter than the original. The possible extensions described in the previous section could improve the match further. Note also that the improvement when losses are used is quite minor, so retransmissions are not likely to explain the difference between

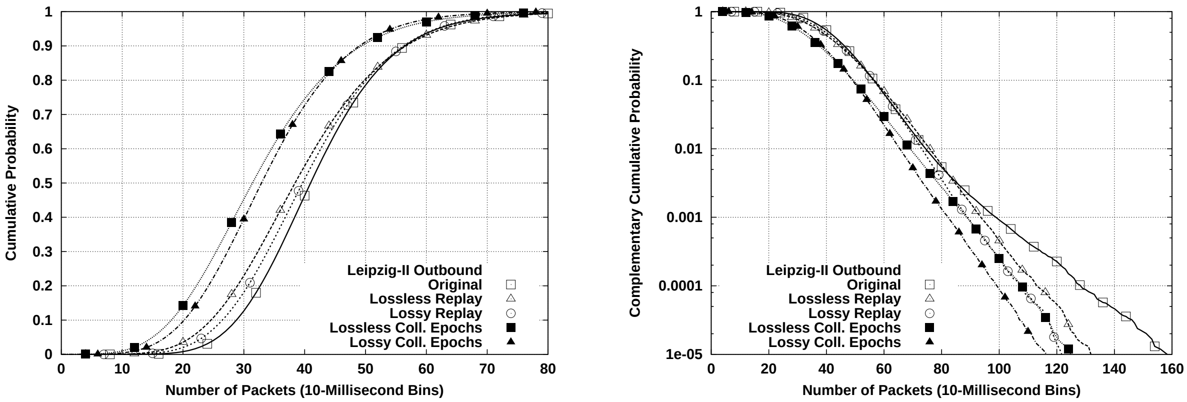


Figure 6.8: Packet throughput marginals for Leipzig-II outbound and its four types of source-level trace replay.

original and replay distributions.

The tails of the marginal distributions from the replays are lighter than those from the original data. Interestingly, the best match is achieved by the lossless replay with fully characterized epochs rather than by the lossy replay. The match is excellent below 10^{-4}. Above this value, the shape of the tail from the original data is less linear, which could be caused by a small number of connections with characteristics that we do not model well. Lossy replays result in significantly lighter tails, as expected given the loss-induced reduction in connection throughput.

Figure 6.8 shows the same analysis for the outbound direction. Collapsed-epochs replays again resulted in bodies that are substantially lighter than the body of the original distribution. In contrast, the full replay achieved a much closer approximation, even overlapping the original distribution for the largest values. Adding losses to the experiments made the replays only a bit closer to the original. This is a strong indication that source-level structure, and not loss/retransmission, is behind the differences between original and replay trace. We can distinguish two regions in the plot of the tails. Below 80 Kpps, the replays with fully characterized epochs provide an excellent match, while those with collapsed epochs result in significantly lighter tails. Above 80 Kpps, the slope of the tail from the original trace is far higher than the slopes of the tails from the replays.

6.2.4 Long-Range Dependence

Another way of looking at the time series of byte and packet arrivals is to study the characteristics of the time series for a wide range of time scales. This can be accomplished using scaling analysis tools,

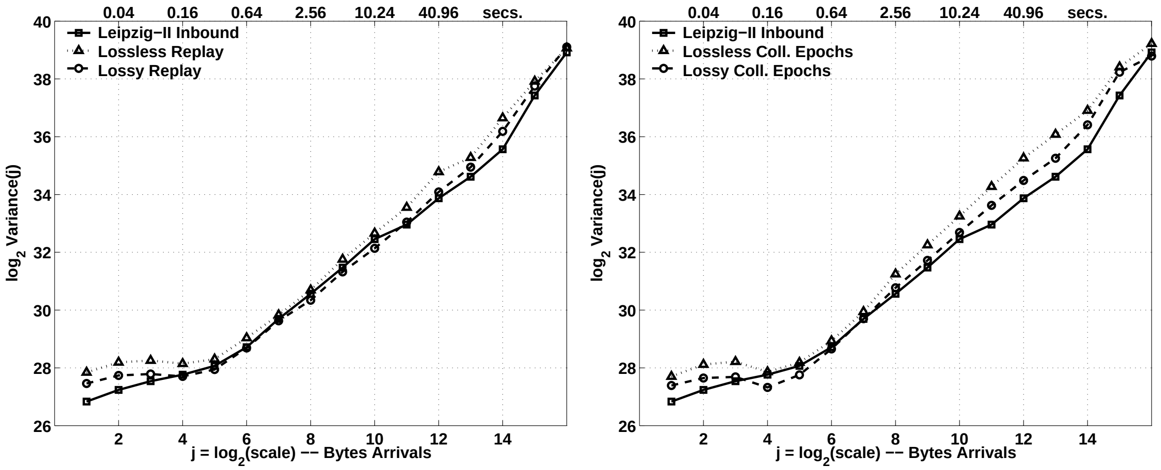


Figure 6.9: Wavelet spectra of the byte throughput time series for Leipzig-II inbound and its four types of source-level trace replay.

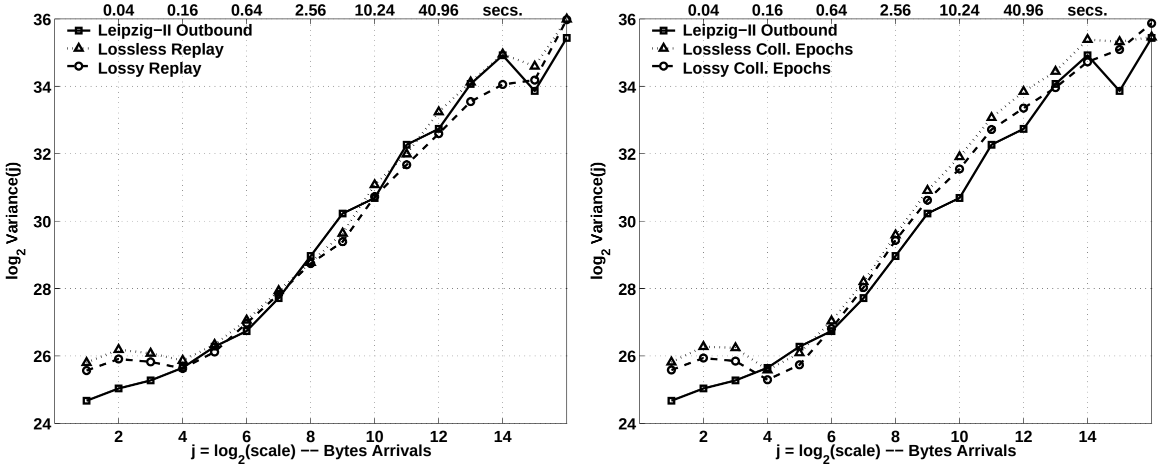


Figure 6.10: Wavelet spectra of the byte throughput time series for Leipzig-II outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
Leipzig-II	*H*	C. I.	*H*	C. I.
Original	0.9201	[0.8990, 0.9412]	0.9973	[0.9762, 1.0184]
Lossless Replay	0.9863	[0.9652, 1.0074]	1.0475	[1.0264, 1.0686]
Lossy Replay	0.9583	[0.9372, 0.9794]	0.9832	[0.9621, 1.0043]
Lossless Coll. Epochs	0.9986	[0.9775, 1.0197]	1.0473	[1.0262, 1.0684]
Lossy Coll. Epochs	0.9668	[0.9457, 0.9879]	1.0083	[0.9872, 1.0294]

Table 6.1: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of Leipzig-II and its four types of source-level trace replay.

such as the wavelet transform, which was introduced in Section 4.2.3. In this section, we use wavelet spectrum plots and Hurst parameters estimates to compare the scaling of the arrival processes found in original and replay traces. Figure 6.9 shows the wavelet spectra of the time series of byte arrivals in the inbound direction. The left plot reveals an excellent match between the original and the full replays. The linear region between octaves 6 and 14 is very similar in the three spectra. This tells us that the kind of long-range dependence found in the original and in the replay traces is very similar. If we equate burstiness to long-range dependence, we can say that the generated traffic faithfully reproduced the burstiness of the original traffic. The finest time scales show a somewhat larger difference between octaves 1 and 5. The spectrum of the original data starts at a lower energy level than the spectra of the replay data. It also shows a linear trend with an upward slope, which is far less clear in the replay data.

The exact cause of the small difference is not completely clear. Our additional experiments strongly suggest that it is due to more complex network-level characteristics in the Internet than in the network testbed. We conducted a large set of experiments (not reported here) which betrayed that the energy levels at the finest time scales are dominated by round-trip times and other network-level parameters[3]. The slightly better match achieved with the lossy replay is consistent with this claim. Further work on network-level modeling may help improve the match, but it is beyond the scope of this dissertation. The approximation seems acceptable for most experimental studies.

The wavelet spectra of the collapsed-epochs replays is similar to the wavelet spectrum of the original trace, as shown in the right plot of Figure 6.9. The spectra from the replays exhibits a slightly higher slope in the linear region, and a slightly worse approximation of the fine-scale region. The benefit of modeling source-level behavior is relatively small, in terms of scaling behavior, for this trace, but present nonetheless.

Figure 6.10 shows the analysis of the wavelet spectra of the time series of byte throughput in the outbound direction. One interesting observation is that the wavelet spectrum of the original is far from the expected straight line. This is due to the low mean throughput on this direction. A handful of connections can have a large impact in the aggregate throughput, which makes the aggregate less stable, showing a less clear scaling. The full replays are very close to the original in the scaling region, but show a larger gap at fine scales. The collapsed-epochs replays result in a slightly worse approximation.

Estimated Hurst parameters for the byte throughput time series are shown in Table 6.1. The original

[3] More specifically, we learned that the range of the distribution of round-trip times determines the knee of the spectrum, while the distribution of window size determines the level of energy at the finest scales. Related results from web traffic simulations can be found in [FGHW99].

trace exhibits a smaller estimated Hurst parameter than the replays. The estimate for the lossy replay is however within the confidence interval of the original for the outbound and very close to the upper bound for the inbound. In general, lossless replays have higher Hurst parameters than lossy replays, and the replays with collapsed epochs have somewhat higher Hurst parameters than the full replays. Note also that several estimated Hurst parameters for the outbound direction are above 1, with the lossless replay even having the lower bound of the confidence interval above 1. Non-stationarities, properly captured by the source-level trace replay, may be behind this extreme burstiness. It is important to note that non-stationarity, even if present, does not change the fact that our computation of wavelet energy and Hurst estimates is identical in all cases. This makes the comparative results meaningful, at least in relative terms.

Figure 6.11 shows the wavelet spectra for the time series of packet throughput in the inbound direction. As in the case of byte throughput, the spectra of the replays are quite similar to the spectrum of the original, especially in the linear region. The spectra of the collapsed-epochs replays are somewhat farther from the original spectrum than the ones from the full replays. The slope of the linear region is again higher for the collapsed-epochs replays, and the difference is also larger at the finest scales.

The analysis of the packet throughput in the output direction shown in Figure 6.12 reveals a close approximation of the original spectrum by the full replays. Collapsed-epochs replays are slightly worse. Note also that the spectrum of the original trace is smoother here than in Figure 6.10. The phenomenon that distorted the linear scaling in the original time series of byte throughput seems far less significant for the time series of packet throughput.

Table 6.2 presents the estimates of Hurst parameters and confidence intervals for the original and replay time series of packet throughput. The original and the lossy full replays have almost identical estimated Hurst parameters for the inbound direction, while the other replays show higher Hurst parameters. The estimated Hurst parameter of the lossy full replay is again the closest one to the original estimate for the outbound direction. It is somewhat lower than the original, but within the confidence interval. The other replays show significantly higher estimated Hurst parameters. Note also that the estimated Hurst parameters for the outbound direction do not go above 1 in this case.

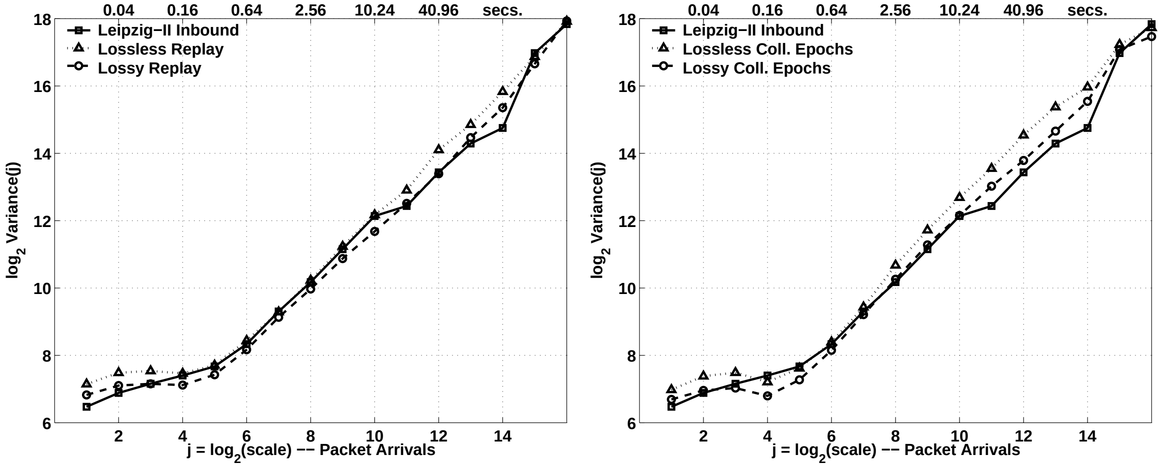


Figure 6.11: Wavelet spectra of the packet throughput time series for Leipzig-II inbound and its four types of source-level trace replay.

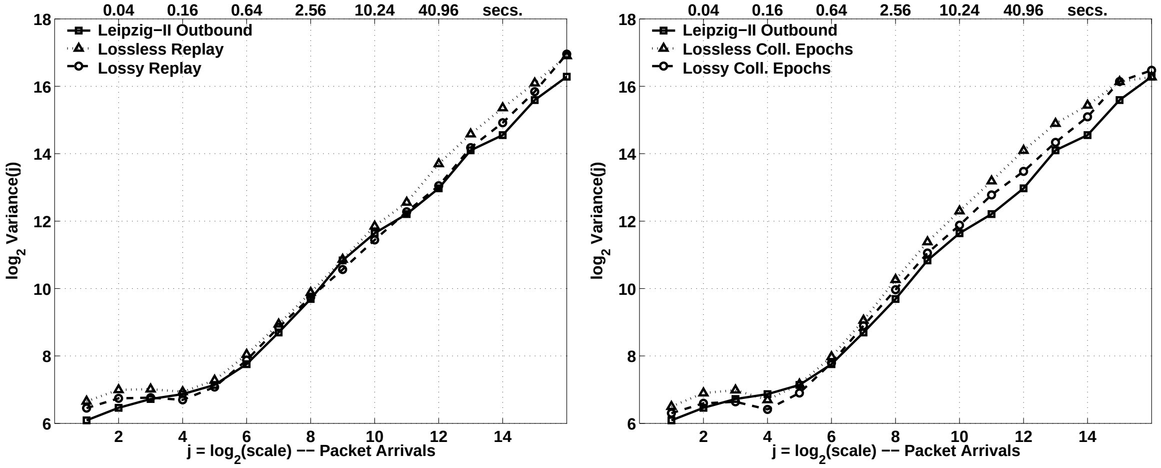


Figure 6.12: Wavelet spectra of the packet throughput time series for Leipzig-II outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
Leipzig-II	H	C. I.	H	C. I.
Original	0.9208	[0.8975, 0.9442]	0.9399	[0.9165, 0.9633]
Lossless Replay	0.9716	[0.9482, 0.9950]	0.9701	[0.9468, 0.9935]
Lossy Replay	0.9271	[0.9038, 0.9505]	0.9194	[0.8961, 0.9428]
Lossless Coll. Epochs	0.9883	[0.9649, 1.0116]	0.9925	[0.9692, 1.0159]
Lossy Coll. Epochs	0.9587	[0.9353, 0.9820]	0.9635	[0.9402, 0.9869]

Table 6.2: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of Leipzig-II and its four types of source-level trace replay.

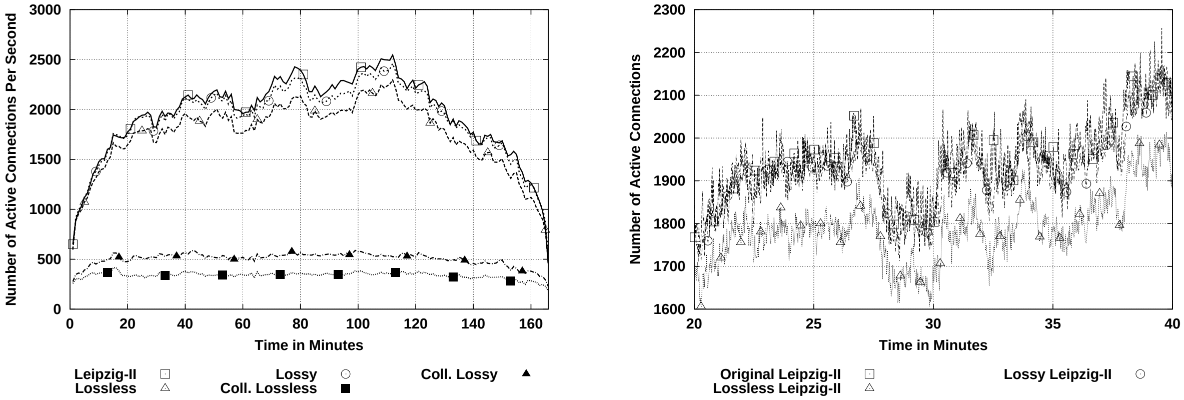


Figure 6.13: Active connection time series for Leipzig-II and its four types of source-level trace replay.

6.2.5 Time Series of Active Connections

The final metric we examine in this chapter to evaluate how closely original and generated traffic match is the time series of active connections. The left plot in Figure 6.13 shows the time series from the original trace using a solid line, and the time series from the four replays using dashed lines. The first observation from this plot is that the collapsed-epochs replays resulted in a strikingly lower number of active connections that the full replays. Since the number of connections replayed in both types of the replay is the same, this difference is due to the substantially shorter durations of the connections replayed with their epochs collapsed. The collapsing of epochs increases connection durations, because quiet times and epoch structure disappear. Epochs require at least one round-trip time to be replayed (see Section 3.1.1). As a result, the number of active connections is several times smaller in the collapsed epochs replays than in the original trace. On the contrary, the number of active connections observed in the full replays is far closer to the original.

The left plot of Figure 6.13 also provides a good illustration of the impact of replaying losses on the quality of the approximation. The number of active connections increases substantially when loss rates are used in the generation, both in the case of collapsed-epochs replays and full replays. However, it is clear from this plot that collapsing epochs has a far more substantial impact on the number of active connections than incorporating losses, at least for the Leipzig-II trace. Given how carefully our replay reproduced the main network-level parameters that affect TCP throughput (round-trip time, window size and loss rates), this result strongly suggest that traffic generated without any modeling of epoch structure and quiet time has an unrealistically low number of active connections.

While the lossless full replay achieves a reasonable approximation of the original time series, the

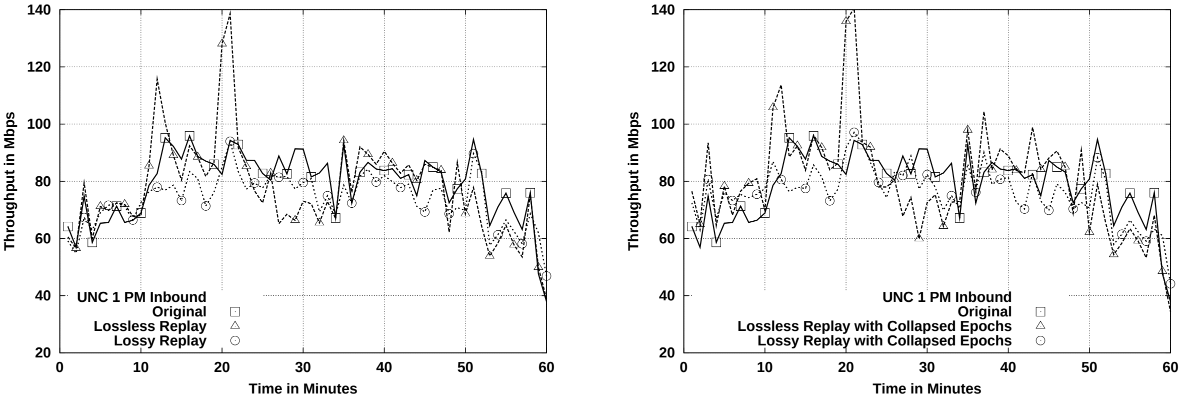


Figure 6.14: Byte throughput time series for UNC 1 PM inbound and its four types of source-level trace replay.

lossy full replay is almost a perfect match. The difference is always below 100 connections, which can be considered an outstanding result. It is clear that generating traffic using a combination of detailed source-level models and primary network-level parameters makes the number of active connections very realistic. Note also that this is not only true for the coarse scale (1 minute) at which the left plot of Figure 6.65 is displayed, but also at the finer scale (5 seconds) in the right plot. Notice for example how closely the replay tracks the significant variability in the original time series.

6.3 Source-level Replay of UNC 1 PM

6.3.1 Time Series of Byte Throughput

Figure 6.14 shows the time series of byte throughput for UNC 1 PM in the inbound direction, revealing a good match between original and replayed traces. Lossless replays with and without collapsed epochs are generally closer than lossy replays, which are often 10 to 20 Mbps below the original. However, lossless replays show large spikes (minutes 14 and 21) that are not found neither in the original trace nor in the lossy replays. The lossy replays are actually very close to the original in the neighborhood of these spikes (*e.g.*, between minutes 20 and 28). Interestingly, the time series for Leipzig-II shown in Figure 6.14 did not reveal a significant difference between lossless and lossy replays. Finding an explanation for this phenomenon requires further analysis, but this plot certainly justifies our choice of comparing the original trace to lossless and lossy versions of its source-level trace replay. Without a lossy replay, we would be tempted to conclude from the artificial throughput spikes in lossless replay that our source-level model is not properly reproducing an end-point limitation that was present in the original environment.

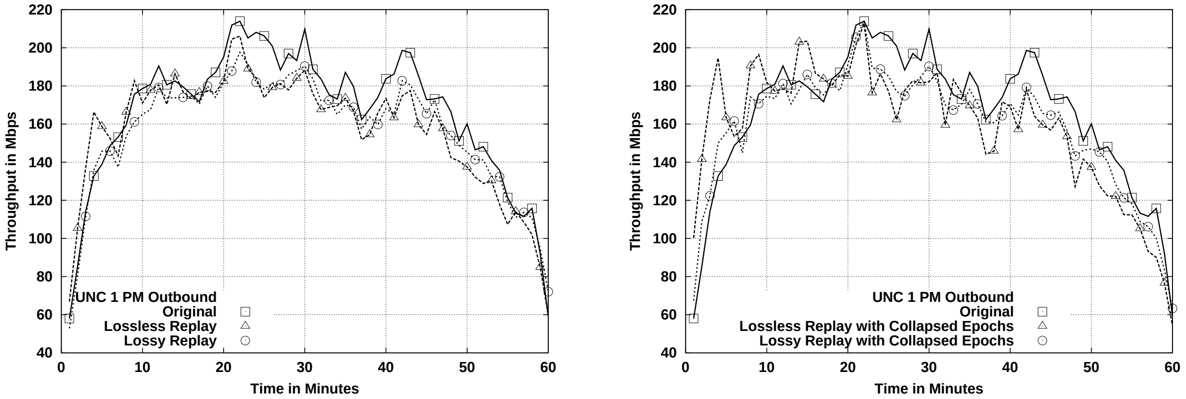


Figure 6.15: Byte throughput time series for UNC 1 PM outbound and its four types of source-level trace replay.

However, the lossy replay, by showing that adding losses eliminates this spikes, demonstrates that they are purely due a network-level parameter and not to a limitation of the a-b-t model. Once again, we are not naively advocating for incorporating open-loop losses into traffic generation experiments, but addressing a difficulty that significant loss can create when trying to understand how realistic our modeling of the traffic source is. Simply relying on a lossless replay can be misleading, as this example demonstrates.

As in the full replay case, the lossless collapsed-epochs replay shows two large spikes that are not present in the lossy collapsed-epochs replays. The general impression from the plot is that collapsing epochs moderately increases the burstiness of the replay. Note for example the larger spike in the minute 5, the spikes in minutes 36 and 44, and the large ditch in minute 29. The collapsed-epochs lossy replay is quite similar to the full lossy replays, but we find a few periods where the approximation of the original throughput is slightly worse. For example, the collapsed-epochs replay shows a drop of byte throughput in minute 40 that is not present in the full lossy replay.

Figure 6.15 reveals somewhat different lessons from the time series of byte throughput in the outbound direction of UNC 1 PM. Regarding the full replays shown in the left plot, we see that the lossless replay has only one significant spike above the original traffic. One reason behind this finding is that the much higher average byte throughput makes spikes due to a few connections far less significant in relative terms.

Both full replays are generally slightly below the byte throughput of the original trace. The reason is not completely clear, but it suggests that the replay has a somewhat lighter distribution of connection throughputs, which makes the aggregate throughput slightly lower. If the replay is continued beyond

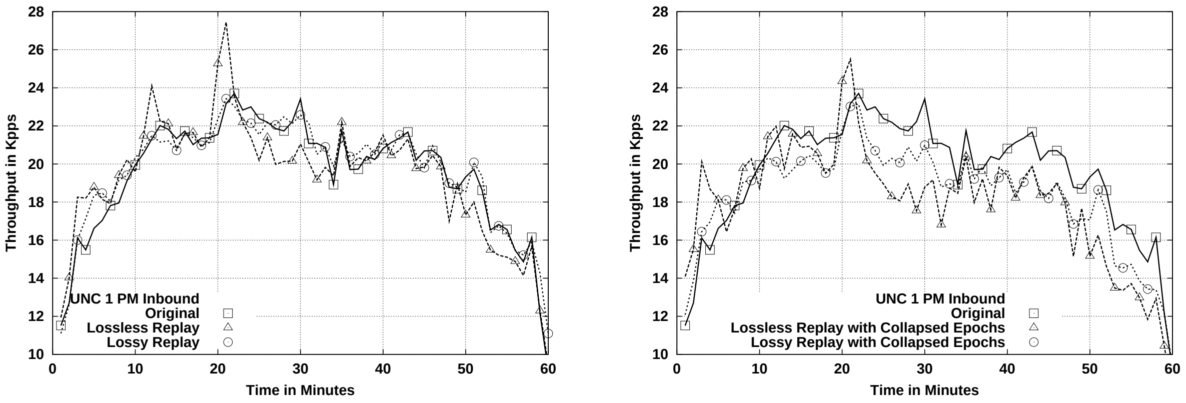


Figure 6.16: Packet throughput time series for UNC 1 PM inbound and its four types of source-level trace replay.

minute 60, we do observe connections that remain active for a few more minutes and transfer enough data to account for the difference between the time series. We examined the logs from the generator hosts and confirmed that no overload occurred during the experiments, so the cause seems to be some artificial limit on the throughputs of the connections in our replay. One cause could be the overestimation of quiet times discussed in Section 5.2.1. Another possible cause is that the replays did not take into account the specific MSS of each connection. Every connection was given the FreeBSD default value (1,460 bytes), which is the most common one on the Internet. However, it could be the case that a significant fraction of the segments were carried in TCP connections with a smaller MSS. These connections would then have higher control overhead, making their transferring of the same payload result in more bytes and therefore higher aggregate throughput. Given the small size of TCP headers, it is unlikely that the extra overhead would result in more than a few additional Mbps.

The results from the replays with collapsed epochs are similar, although we observe several additional spikes in the case of the lossless replay. The lossy replay does not show these spikes, but it is still below the original for most of the time series. Interestingly, it provides a closer approximation in some regions, such as between minutes 10 to 22. We can argue that this is an accidental improvement due to the artificially larger throughputs that a fraction of the connections achieves after their epochs are collapsed.

6.3.2 Time Series of Packet Throughput

The analysis of the packet throughput in the inbound direction shown in Figure 6.16 reveals a number of interesting characteristics. Both lossless replays show substantial spikes above the original packet throughput. This is consistent with the similar finding for byte throughput. We also observe that

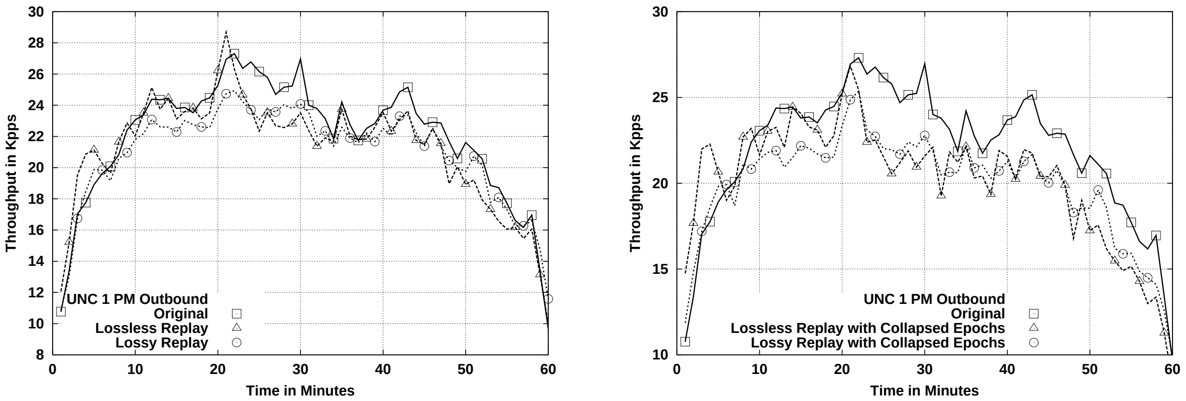


Figure 6.17: Packet throughput time series for UNC 1 PM outbound and its four types of source-level trace replay.

collapsed-epochs replays generated a substantially smaller number of segments than full replays. As in the case of the analysis of the Leipzig-II replay shown in Figure 6.3, the lack of detailed source-level modeling in the collapsed-epochs replays makes traffic less realistic in terms of the aggregate packet throughput. In contrast, the lossy full replay shows an excellent match for most of the time series. This result is different from the Leipzig-II one, where the full replays achieved a good approximation, but were still below the original packet throughput. Adding per-connection losses had a very minor impact on the Leipzig-II packet throughput, but the effect is substantial in the UNC 1 PM replay, where we observe increments of up to 2,000 packets per second. This result demonstrates the effectiveness of our source-level modeling method, and also justifies our effort to incorporate losses in the replay in order to study the realism of our modeling approach.

Figure 6.17 examines packet throughput in the outbound direction. Unlike the inbound direction, adding losses does not have a substantial impact here, and the aggregate packet throughput remains below the original trace even for the lossy full replay. As discussed in Section 6.2.2, this could be due to some limitations of our data acquisition algorithm in terms of how well it infers source-level characteristics, or to the use of the default MSS for all connections. As in previous cases, collapsed-epochs replays generate a substantially lower number of packets than full replays, which are far closer to the original packet throughput.

6.3.3 Marginal Distributions

The marginal distribution of byte throughput for the inbound direction of UNC 1 PM and its replays are shown in the Figure 6.18. The bodies of the distributions show that lossy replays provide a better

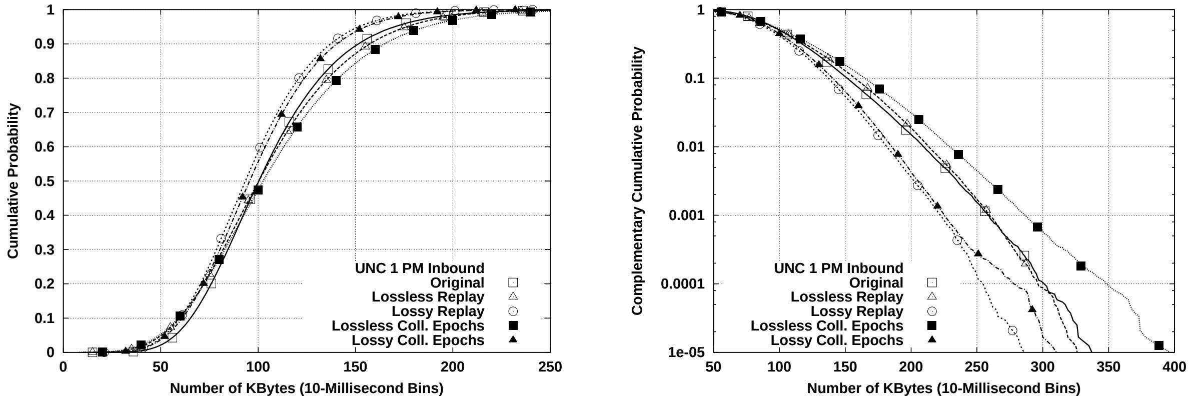


Figure 6.18: Byte throughput marginals for UNC 1 PM inbound and its four types of source-level trace replay.

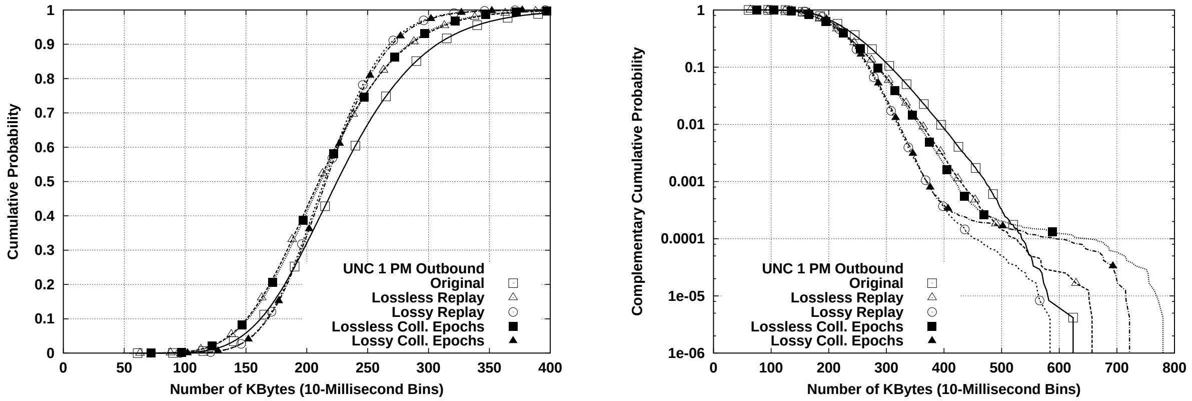


Figure 6.19: Byte throughput marginals for UNC 1 PM outbound and its four types of source-level trace replay.

approximation, although they are slightly heavier than the original. Interestingly, the analysis of the time series in Section 6.3.1 showed lower aggregate throughput from lossy replays, which seems inconsistent with the heavier bodies in the marginal distribution. The explanation is given by the plot of the tails of the marginals, which shows far lighter tails from the lossy replays. The way in which losses were incorporated in the experiments limited peak throughput substantially at the fine scales considered in the marginal plots. This is because the probability of artificial losses increases linearly with throughput, which is not generally true for real conditions. On the contrary, the lossless full replay reproduced the tail very accurately, demonstrating that the experimental environment and generation method are perfectly capable of reproducing the observed peak throughputs. It seems likely that further refinements in the implementation of per-connection losses, making them less open-loop, could make the tails closer to the original.

The marginal distributions in the outbound direction, which are shown in Figure 6.19, reveal a

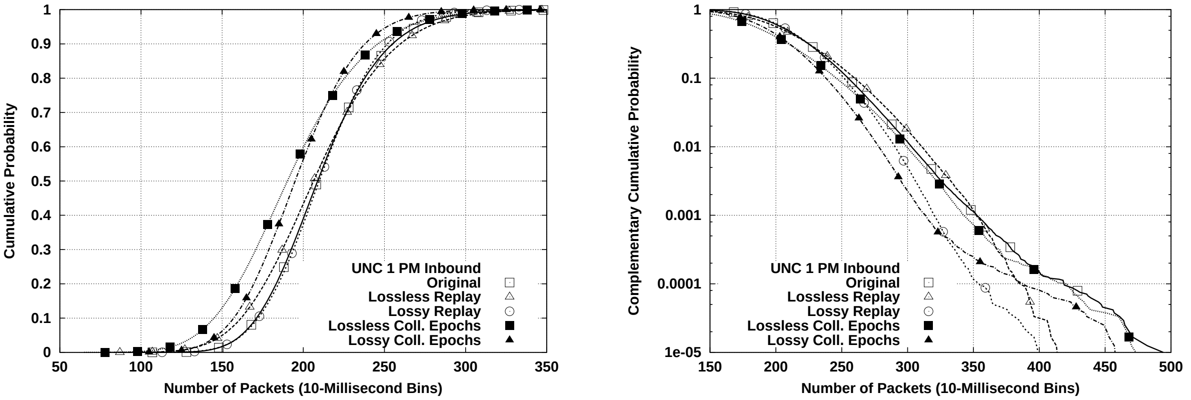


Figure 6.20: Packet throughput marginals for UNC 1 PM inbound and its four types of source-level trace replay.

somewhat worse approximation. We can distinguish three regions in the plot of the bodies. For values below 175 KB, lossless replays are lighter than the original, while lossy ones are heavier. Above 175 KB, all replays are lighter, which shows that the finding of lower aggregate byte throughput in Section 6.3.1 is due to overall lower throughputs at fine scales (rather than only to lighter tails). In the region after 175 KB, we can also observe that lossy replays are heavier below 275 KB and lighter above that. The marginal distributions from the lossy replays are less concentrated around the mean value, and are therefore somewhat more bursty, which is consistent with the similar finding for Leipzig-II (see Section 6.2.3).

Regarding the tails, we observe that for probabilities below 0.00075, the tail of original marginal is substantially heavier than the tails of the replay marginals. For probabilities above that, the collapsed-epochs replays show a major change in the shape of the distributions, being far heavier than the original for the largest values. We did not encounter a similar phenomenon in the Leipzig-II replays, where lossy collapsed-epochs replays always had a lighter tail than the lossless full replay. The number of 10-millisecond bins with very high throughput is larger for collapsed-epochs replays than for the full replays. Note that this artifact is only visible by looking at the tails of the marginals, and not at their bodies or at the time series of byte throughput.

The marginal distributions of packet throughput for UNC 1 PM inbound are shown in Figure 6.20. As observed for Leipzig-II, and as we may expect from 6.16, collapsed-epochs replays result in bodies that are significantly lighter than the body of the original marginal. Full replays are far closer, being the lossy full replay an excellent approximation of the original distribution. Interestingly, the tails reveal a rather different picture. Below 350 Kpps, the lossy replays have lighter tails than the original, especially in the case of the lossy full replay. Lossless replays closely approximate the original tail. Above 350

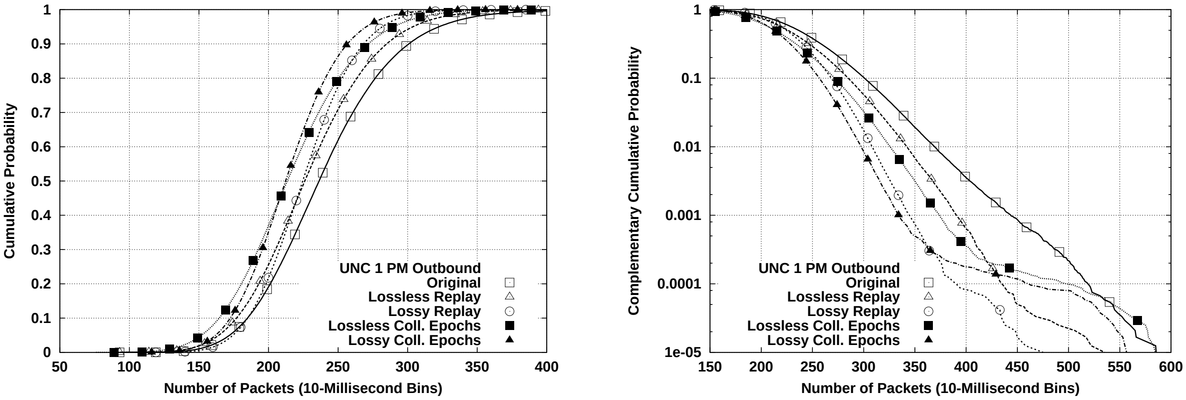


Figure 6.21: Packet throughput marginals for UNC 1 PM outbound and its four types of source-level trace replay.

Kpps, both full replays are lighter than the original, while the collapsed-epochs replays reproduce the probability of very high throughput bins accurately.

Figure 6.21 shows the marginal distributions of packet throughput in the opposite direction. All replays are lighter than the original, being the lossy full replay the closest one. The tails from the replays are also significantly lighter than the original tail. We also observe a similar change in the tail of the collapsed-epochs replays, which are very close to the original for the largest values.

6.3.4 Long-Range Dependence

The left plot of Figure 6.22 shows that the wavelet spectrum of the original byte throughput in the inbound direction is well approximated by both full replays for lower and medium octaves. The finding of this good match at the lower octaves differs from the result for the replay of the Leipzig-II trace, where this part of the wavelet spectrum was not so well approximated. The lossy replay shows less energy for octaves 8 and above, while there is a significant jump in the energy of the lossless replay for octaves 12 and above. In the right plot, the lossless collapsed-epochs replays shows substantially more energy for octaves above 4, while the lossy replay provides a better approximation.

For the outbound direction, the left plot of Figure 6.23 reveals a better approximation of the finest scales by the lossless full replay, while both full replays closely match the original spectrum at coarser scales. The right plot shows that both collapsed-epochs replays have less energy at the finest scales, with a rather sharp ditch for octaves 5 and 6 that was not present in the original. This ditch was far less pronounced in the full replays. Beyond the finest scales, the lossless collapsed-epochs replay

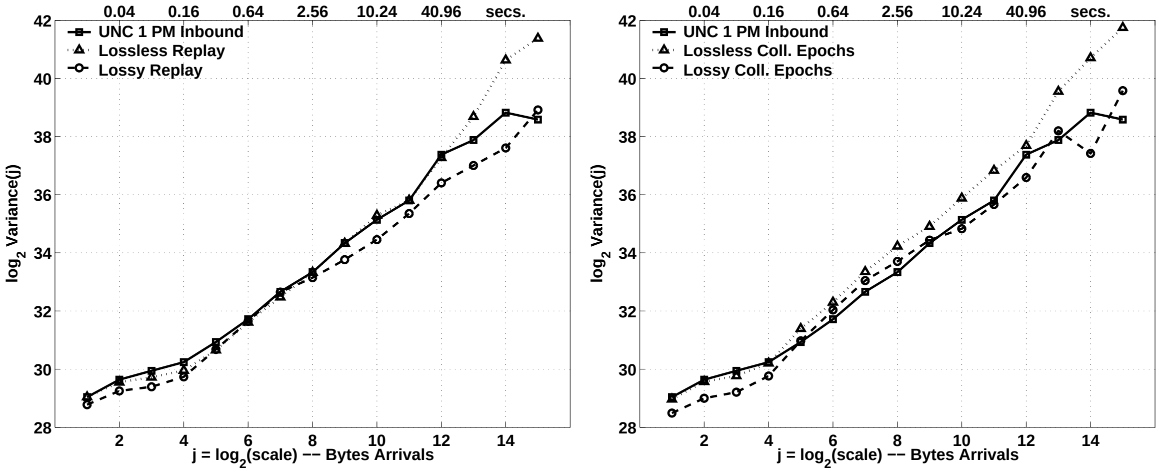


Figure 6.22: Wavelet spectra of the byte throughput time series for UNC 1 PM inbound and its four types of source-level trace replay.

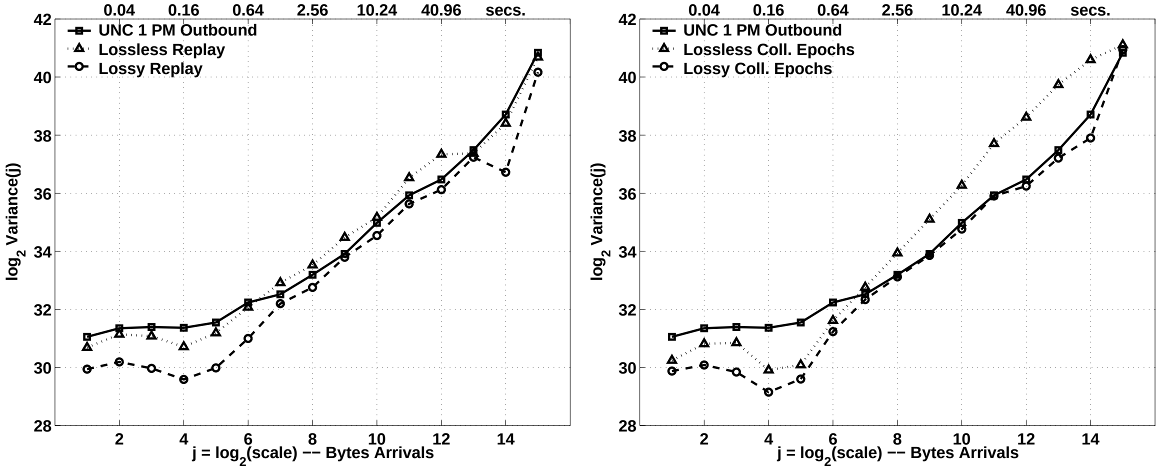


Figure 6.23: Wavelet spectra of the byte throughput time series for UNC 1 PM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 1 PM	*H*	C. I.	*H*	C. I.
Original	0.9557	[0.9113, 1.0002]	0.9717	[0.9272, 1.0161]
Lossless Replay	0.9632	[0.9188, 1.0077]	0.9585	[0.9141, 1.0030]
Lossy Replay	0.9118	[0.8674, 0.9563]	0.9306	[0.8861, 0.9750]
Lossless Coll. Epochs	0.9521	[0.9077, 0.9966]	1.0170	[0.9726, 1.0615]
Lossy Coll. Epochs	0.8441	[0.7996, 0.8885]	0.8657	[0.8212, 0.9101]

Table 6.3: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of UNC 1 PM and its four types of source-level trace replay.

is a poor match of the original, while the lossy one provides a close approximation. This high impact of losses in the collapsed-epochs replay, far larger than in the full replay case, suggests a significant interaction between loss and long-range dependence when traffic is not generated according to a detailed source-level model. In other words, endpoints that generate traffic according to less realistic models (without epochs) are artificially more aggressive than Internet sources. This makes them more sensitive to lossy environments, since losses can more sharply decrease their higher throughput. This can result in experiments that overestimate the impact of losses on performance.

The estimated Hurst parameters and their confidence intervals shown in Table 6.3 are somewhat surprising. In the inbound direction, the estimated Hurst parameter of the original trace is most closely approximated by the lossless replays. The lossy full replay is slightly lower, and the lossy collapsed-epochs replay is far lower. The same is true in the opposite direction, at least for the lossless replays. It is difficult to interpret the meaning of these estimates in the context of the previous results. On the one hand, we found large spikes in the time series of byte throughput that suggest substantially higher burstiness in the lossless replays. Additionally, the wavelet spectra in Figure 6.22 did not find better approximations from the lossless replays. Notice for example that the lossless collapsed-epochs replay is clearly the farthest from the original. On the other hand, the tails of the marginal distributions clearly favored the lossless replays, showing lighter tails for the lossy replays. We could argue that the different metrics refer to different measures of burstiness, and conclude that adding artificial losses (using our open-loop method) makes the lossy replays less realistic in terms of Hurst parameter estimates. However, this conclusion seems too simplistic, since it is in contradiction with the Leipzig-II results. Adding losses made the estimated Hurst parameters far closer in that case. Assuming that the observed differences between the estimated Hurst parameters are significant, the reason for these divergent conclusions regarding the impact of losses must necessarily lie in some fundamental difference in the nature of the two network links. The estimated Hurst parameters say little about the difference, since all of the estimates are similarly high (above 0.92).

As discussed in Chapter 4, the Leipzig-II trace is a good example of university traffic dominated by downloading behavior (*i.e.*, inbound traffic is substantially higher than outbound traffic). In contrast, the UNC 1 PM trace is dominated by content downloaded from UNC servers (rather than downloads from UNC clients) due to the presence at UNC of a major Internet repository of software and content, `ibiblio.org`. This made traffic volume and number of connections far higher for UNC. Still, why would these differences make introducing losses beneficial in the Leipzig case and detrimental in the UNC case for the approximation of the original Hurst parameters? We can speculate that the rate-limiting mechanisms used by `ibiblio.org` create unusual loss patterns that are poorly approximated by our open-loop losses, but we do not have any supporting evidence.

The lessons from the analysis of the scaling in the packet throughput series is quite similar. The plots in Figure 6.24 show reasonably close approximations of the original by all of the replays in the inbound direction, and somewhat worse ones in the outbound direction. The spectrum of the lossless full replay provides the closest approximation to the spectrum of the original in both directions. The spectrum of the lossless collapsed-epochs replay is clearly not as close, showing a higher slope for medium to coarse time scales. As in the case of byte throughput, lossy replays show less energy than the original trace, especially for the fine scales in the outbound direction. Note also the systematic ditch around octave 14 for all four spectra from lossy replays. This suggests some unexpected periodicities at the 1-minute scale. A similar ditch can be found in the outbound direction of the original time series in octave 13, and this ditch is not reproduced by the replays.

Regarding the estimated Hurst parameters and their confidence intervals, Table 6.4 shows different results for the two directions. The estimates for the inbound direction confirm the lossless full replay as an excellent approximation, but here the lossless collapsed-epochs replay is also very close to the original. Both lossy replays are well below the estimated Hurst parameter of the original time series, and outside its confidence interval. The estimates for the outbound direction show again an excellent approximation by the lossless full replay, but here the lossless collapsed-epochs replay is far higher than the original and well within the non-stationarity region. Lossy replays are substantially better in the outbound direction, with the lossy full replay matching the original estimate.

6.3.5 Time Series of Active Connections

As in the case of Leipzig-II, the lossy full replay of UNC 1 PM achieved a perfect match of the original time series of active connections This is clear for both the entire range of the time series shown in the left plot of Figure 6.26 using 1-minute bins, and for the 20-minute region shown in the right plot using 1-second bins. This finer scale view shows several sharp spikes (minutes 24, 28, 30, 35 and 39) that the lossy full replay tracked accurately. The lossless replay has only a slightly lower number of active connections per second, showing similar spikes (but with a negative offset in the y-axis). Collapsed-epochs replays had a far smaller number of active connections. Also, they did not track the features of the original time series so well. Notice for example the absence of the minute 24 spike in the collapsed-epochs replays.

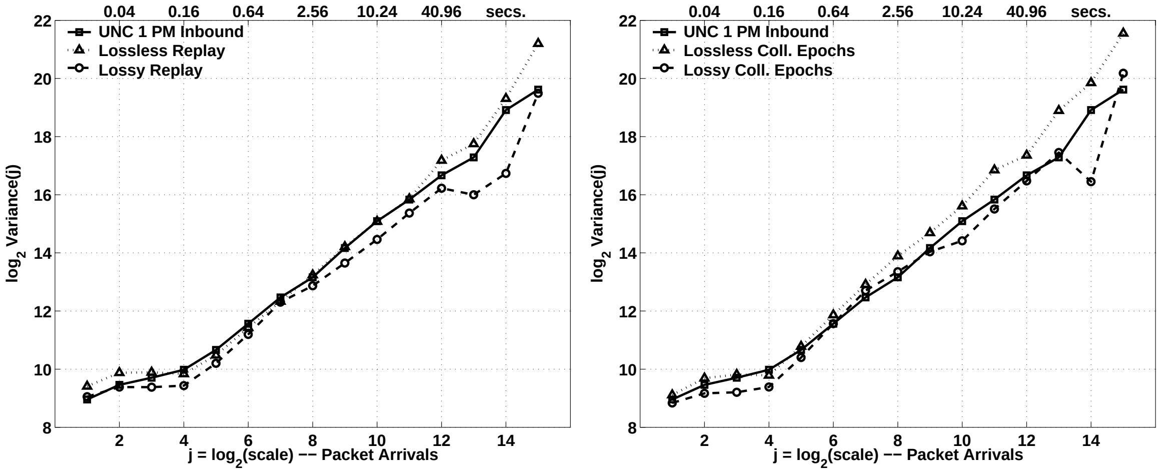


Figure 6.24: Wavelet spectra of the packet throughput time series for UNC 1 PM inbound and its four types of source-level trace replay.

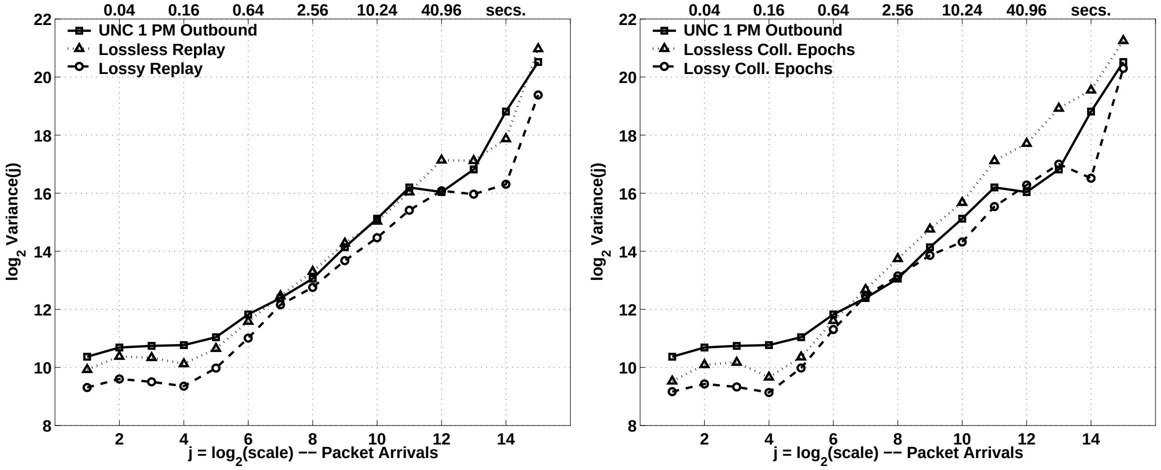


Figure 6.25: Wavelet spectra of the packet throughput time series for UNC 1 PM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 1 PM	*H*	C. I.	*H*	C. I.
Original	0.9564	[0.9158, 0.9970]	0.9339	[0.8933, 0.9746]
Lossless Replay	0.9776	[0.9370, 1.0182]	0.9512	[0.9106, 0.9918]
Lossy Replay	0.8719	[0.8313, 0.9125]	0.9512	[0.9106, 0.9919]
Lossless Coll. Epochs	0.9464	[0.9058, 0.9871]	1.0956	[1.0549, 1.1362]
Lossy Coll. Epochs	0.8509	[0.8103, 0.8916]	0.9200	[0.8793, 0.9606]

Table 6.4: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of UNC 1 PM and its four types of source-level trace replay.

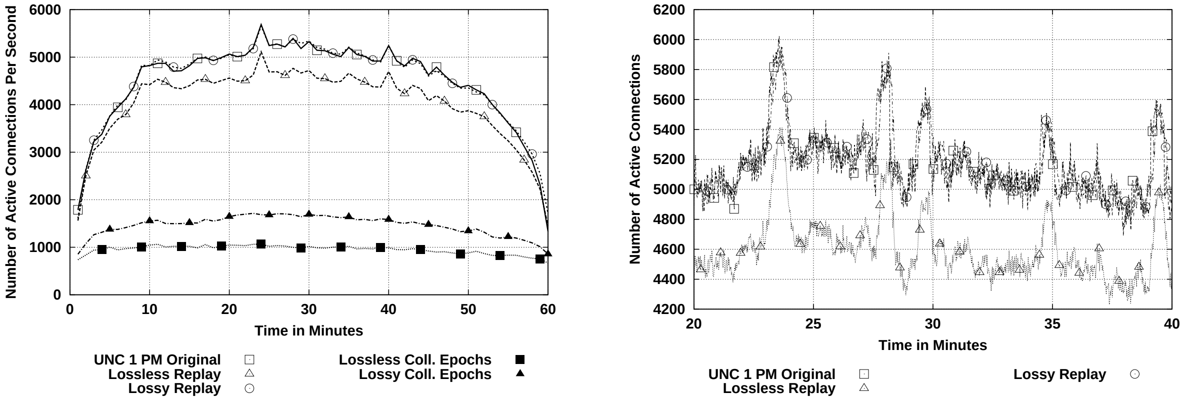


Figure 6.26: Active connection time series for UNC 1 PM and its four types of source-level trace replay.

6.4 Mid-Chapter Review

The present chapter is the longest one in this dissertation, and presents the results of 20 source-level trace replay experiments using 130 plots and 10 tables. The conclusions are not always straight-forward or consistent across traces, so it is difficult to form a coherent picture by simply going through the entire body of results. In this section, we summarize our results so far in order to make the rest of the chapter easier to follow. Our summary is in the form of a list of 18 observations, which report both on findings that were consistent for Leipzig-II and UNC 1 PM, and findings that were inconsistent.

6.4.1 Observations on Byte Throughput

From the analysis of the plots of the time series of byte throughput, their marginal distributions and wavelet spectra, we can make the following observations:

B.1 Both full and collapsed-epochs replays provide a reasonable approximation of the original 1-minute time series of byte throughput and the body of its 10-millisecond marginal. Replays do not track every spike in the original time series, but the similarity is remarkable. The replays achieve a very close approximation of the Leipzig-II time series, but are slightly below the UNC 1 PM time series. For both traces, the approximation of the bodies of the original marginal are somewhat better for the inbound direction than for the outbound one. This observation is not explained by traffic volume asymmetry, since the inbound direction was the dominant direction in terms of byte volume only in the case of Leipzig-II.

B.2 Lossless replays sometimes show substantially more spikes of 1-minute byte throughput above the original trace than lossy replays. This is clear for UNC 1 PM but not for Leipzig-II. At the finer scales studied by the marginal distributions, we find that the tails of the lossless replays are substantially heavier than those of the lossy replays. However, they are not consistently above the tails of the original distributions. In contrast, the results for every trace show that the bodies of the lossless replays are wider than the bodies of the lossy replays. This reveals higher burstiness in the lossless replays in the sense that they have a higher probability of bins with byte throughput far from the mean (*i.e.*, a larger number of 10-millisecond intervals with have rather low or rather high byte throughput).

B.3 Collapsed-epochs replays show somewhat more bursty 1-minute time series, and track the changes in the shape of the original time series less closely. The extra burstiness may not appear very substantial in the plots, but given the coarse scale, it may have a large impact on experiments sensitive to prolonged byte throughput spikes. We do not find a corresponding phenomenon for the marginal distributions, where collapsed-epochs replays are generally close to the full replays (except for the outbound direction of UNC 1 PM). Together with observation B.5, this shows that the extra burstiness of the collapsed-epochs replays manifests itself in the auto-correlation structure of the byte throughput process, rather than in the set of byte throughputs observed throughout the replays.

B.4 Full replays provide a close approximation of the scaling region (octaves 6 to 15) of the wavelet spectra of the original traces. This does not necessarily translate into similarly good approximations of the estimated Hurst parameters. Only the lossy replays are within confidence intervals for Leipzig-II, while only the lossless ones are within confidence intervals for UNC 1 PM.

B.5 Collapsed-epochs replays tend to show slightly more energy in the scaling region. This is true for the four spectra from lossless replays and for the two spectra from lossy replay of Leipzig-II. However, the energy of the original scaling region is well approximated by the lossy collapsed-epochs replay for the outbound direction of UNC 1 PM. This higher energy in the wavelet spectrum plot does not necessarily translate into higher estimates of the Hurst parameters.

B.6 Both full and collapsed-epochs replays do not consistently match the spectra of the finer scales (octaves 1 to 5). We find higher or slightly higher energy levels for the replays of Leipzig-II, similar levels for the replays of the inbound direction of UNC 1 PM and lower levels for the outbound direction of UNC 1 PM.

B.7 By construction, the most detailed replay is the lossy full replay, so we expect it to achieve the best approximation of the original trace. This was always true for 1-minute time series, the body of the marginal distribution and the scaling region of the wavelet spectrum. However, it was not consistently true for the tail of the marginal distribution, the energy of the wavelet spectrum at fine scales, and the estimated Hurst parameter.

6.4.2 Observations on Packet Throughput

We can make the following observations regarding packet throughput:

P.1 Full replays achieve a close approximation of the original 1-minute time series of packet throughput, remaining between 2% and 8% below the original for most of the time series. Collapsed-epochs replays result in a substantially worse approximation, being between 20% to 30% below the original for most of the time series. This difference is also present in the bodies of the 10-millisecond marginal distributions. In the best case for full replays, the median of the marginal distribution is equal to the original median for the inbound direction of the UNC 1 PM lossy replay. In the worst case, the median is 7% below the original for the inbound direction of the Leipzig-II lossy replay. Collapsed epochs replays show medians of the marginal distributions that are 20% (UNC 1 PM inbound) and 25% (Leipzig-II outbound) below the original median.

P.2 Incorporating losses into the replays increases packet throughput, reducing the distance to the original time series. While this effect is small for Leipzig-II, it is rather significant for UNC 1 PM inbound. In addition, lossless replays sometimes show more artificial spikes in the 1-minute time series plot than the lossy ones (*e.g.*, UNC 1 PM outbound). This phenomenon seems less prominent for packet throughput than for byte throughput (see observation B.2).

P.3 Unlike the byte throughput case, the tails of the packet throughput from the replays marginals are never significantly heavier than the original tails. Lossless replays provide the best approximations of the original tails, being excellent in some cases (Leipzig-II inbound and UNC 1 PM inbound). Lossy replays show lighter tails than lossless replays, revealing significantly worse approximations of the original tails. We can also observe that the tails of the collapsed-epochs replays are consistently lighter than those of the full replays. However, the impact of detailed modeling on the tails of the marginals is less prominent than the impact of incorporating losses.

P.4 Full replays and lossy collapsed-epochs replays provide good approximations of the original wavelet spectra, while the lossless collapsed-epochs replays show somewhat higher energy. In general, we can say that the best approximation is achieved by the lossless full replay. As in the case of byte throughput, Hurst parameter estimates offer a different picture. Only the estimates for the lossy full replay are within confidence intervals of the original estimates for Leipzig-II, while the estimates for both lossless and lossy full replays are within confidence intervals for UNC 1 PM.

P.5 Replays do not consistently reproduce the energy levels at the finest scales of the original time series of packet arrivals. We find minor differences for Leipzig-II and UNC 1 PM inbound, and substantially larger ones for UNC 1 PM outbound. Collapsed-epochs replays are significantly worse than full replays only for UNC 1 PM.

6.4.3 Observations on Active Connections

Regarding active connections, we can make the following observations that hold true for both Leipzig-II and UNC 1 PM:

C.1 The number of active connections in the original trace and in the full replays is very similar.

C.2 The lossy full replay provides the best approximation of the active connection time series, being within 1% of the original time series. There is no difference for UNC 1 PM.

C.3 The number of active connections in collapsed-epochs replays is several times smaller than the original (around 3 times smaller for Leipzig-II and UNC 1 PM).

C.4 Adding losses to the replays substantially increases the average number of connections. This increase is of the same magnitude for both full and collapsed-epochs replays.

C.5 Full replays track the features of the original time series very closely. The only difference between lossless and lossy replays is a slowly varying offset. This suggests a homogeneous impact of losses, which lengthens the lifetimes of a stable number of connections throughout the traces.

C.6 Unlike full replays, collapsed-epochs replays do not track the features of the original time series. However, the magnitude of this effect pales in comparison to the much smaller number of active connections.

6.5 Source-level Replay of UNC 1 AM

6.5.1 Time Series of Byte Throughput

The plots of the 1-minute time series of byte throughput for the original UNC 1 AM and its replays are shown in Figure 6.27 (inbound direction) and in Figure 6.28 (outbound direction). For the inbound,

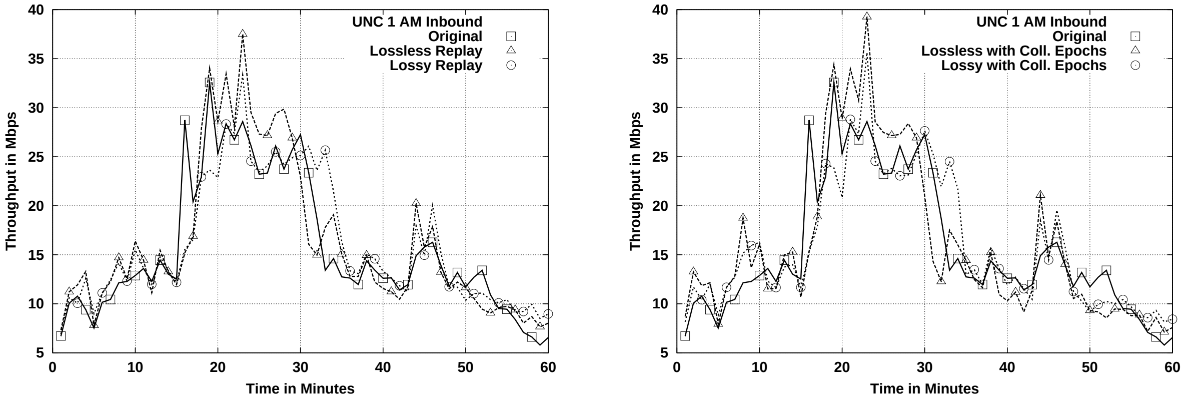


Figure 6.27: Byte throughput time series for UNC 1 AM inbound and its four types of source-level trace replay.

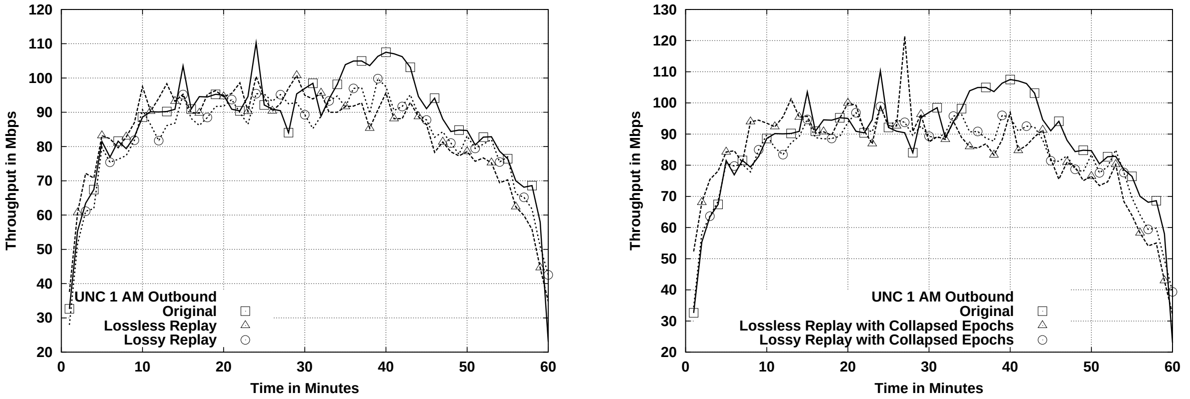


Figure 6.28: Byte throughput time series for UNC 1 AM outbound and its four types of source-level trace replay.

we observe a moderately bursty time series with a large increase in byte throughput between minutes 15 and 32. In good agreement with observation B.1, the replays track the shape of the original time series well. They also approximate some smaller spikes, such as the one in minute 45, and miss others, such as the one in minute 17. The result is similar for the outbound direction, although we again find a slightly lower overall throughput in the replays. There is also an area of higher throughput in the original trace between minutes 35 and 43 that is not properly reproduced by any of the replays. The full lossy replay provide the closest approximation, but there is still a clear difference with respect to the original time series.

The results also support the observation of higher burstiness from lossless replays, B.2, and from collapsed-epochs replays, B.3; especially for the inbound direction. The results are also consistent with observation B.7, since the full lossy replay appears closest to the original.

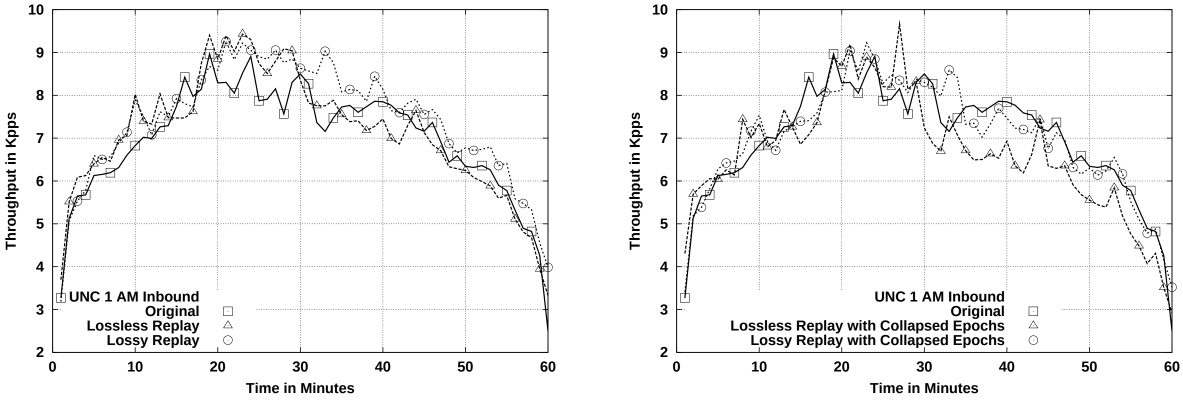


Figure 6.29: Packet throughput time series for UNC 1 AM inbound and its four types of source-level trace replay.

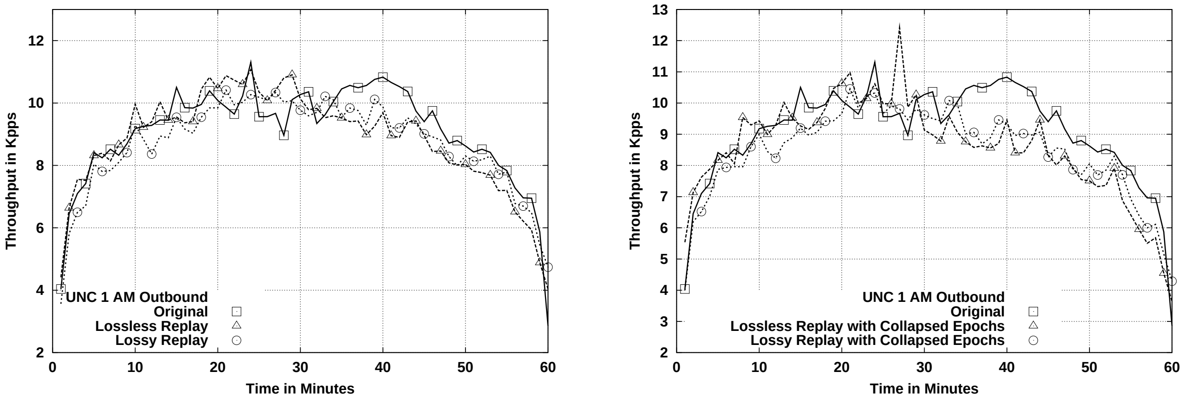


Figure 6.30: Packet throughput time series for UNC 1 AM outbound and its four types of source-level trace replay.

6.5.2 Time Series of Packet Throughput

The time series of packet throughput for UNC 1 AM inbound shown in Figure 6.29 are in sharp contrast to earlier results. As stated in observation P.1, the time series from the replays of the previous traces were generally below the time series of the original trace. However, the full replays of UNC 1 AM are often above the original packet throughput, especially in the case of the lossy full replay. The same is not true for the outbound direction, as shown in Figure 6.30, where the replays are again below the original for a large fraction of the time series. While the replays provide a reasonable approximation of the overall time series, the original packet throughput in the outbound direction is substantially lower between minutes 35 and 43. The difference is most apparent for the collapsed-epochs replays.

Regarding observation P.2, we can see that collapsing epochs substantially reduced packet throughput. Paradoxically, this makes the time series of the lossy collapsed-epochs match the original quite

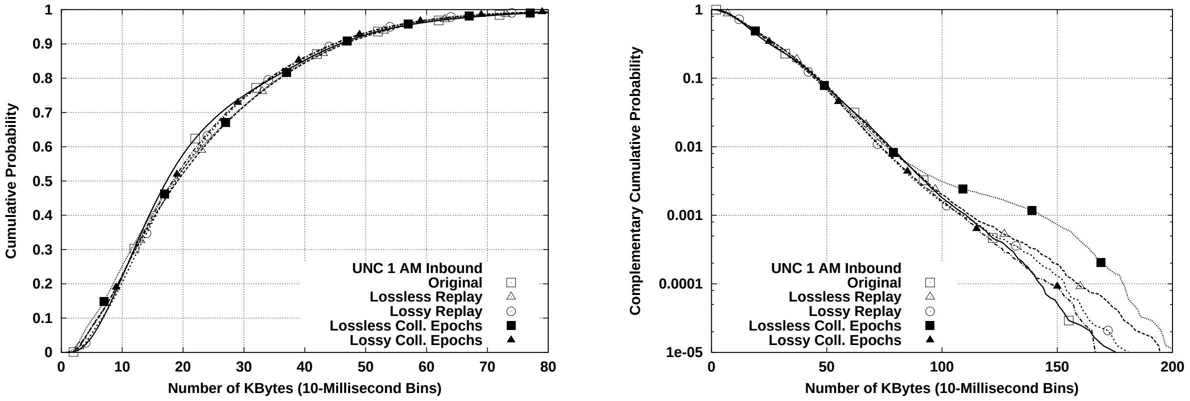


Figure 6.31: Byte throughput marginals for UNC 1 AM inbound and its four types of source-level trace replay.

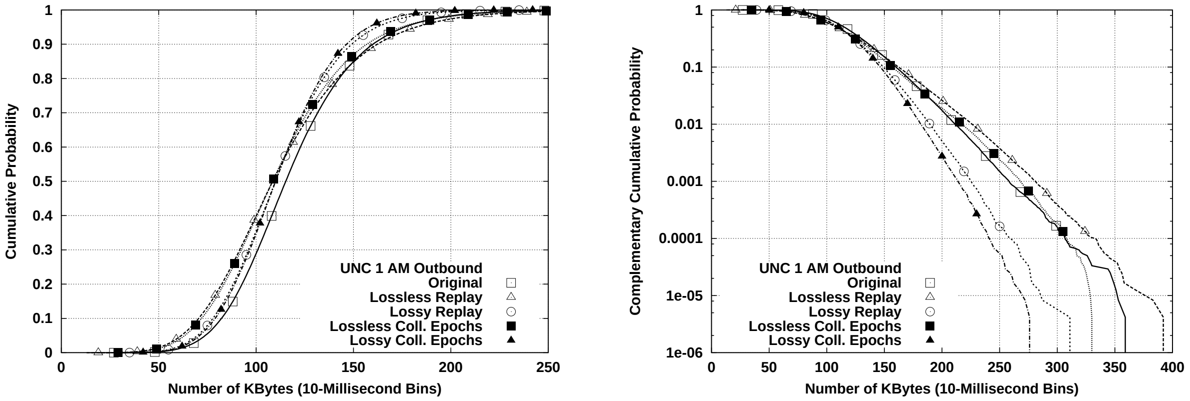


Figure 6.32: Byte throughput marginals for UNC 1 AM outbound and its four types of source-level trace replay.

well, although the same is not true for the lossless collapsed-epochs replay. Note also that it is difficult to argue for this trace that the lossy replays are significantly more bursty than the lossy ones at the 1-minute scale. We only observe one artificial spike in minute 27 for the lossless collapsed-epochs replay.

6.5.3 Marginal Distributions

Figures 6.31 and 6.32 study the marginal distributions of the original 10-millisecond time series UNC 1 AM and those from the source-level trace replays. The bodies of the distributions from the replays are almost identical to the original for the inbound direction, and quite close for the outbound direction, which further supports observation B.1. Observation B.2 is consistent with these results, although the bodies of the lossless replays are very close to those of the lossy ones in this case. We do however observe consistently heavier tails from lossless replays. Note the much heavier tail from the lossless collapsed-

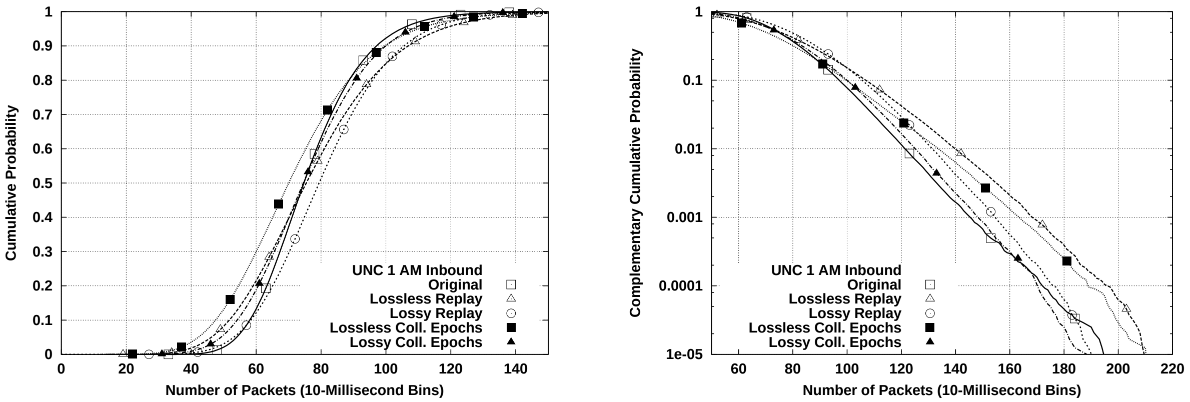


Figure 6.33: Packet throughput marginals for UNC 1 AM inbound and its four types of source-level trace replay.

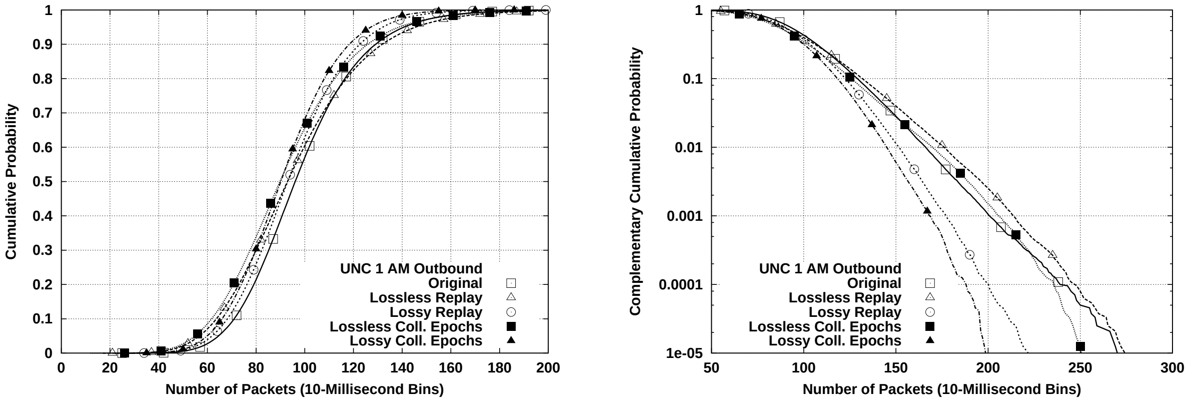


Figure 6.34: Packet throughput marginals for UNC 1 AM outbound and its four types of source-level trace replay.

epochs replay, which reveals an extra burstiness that was not visible in Figure 6.27. In agreement with observation B.3, we do not find consistently wider bodies or heavier tails from the collapsed-epochs replays. Finally, observation B.7 remains valid, with the lossy full replay being best for the tail of the inbound direction, but clearly not for the opposite direction.

The lesson from the plots of the packet throughput marginals shown in Figures 6.33 and 6.34 is similar to the one discussed for the time series in Section 6.5.2. In the inbound direction, the marginal from the lossy full replay is heavier than the original, while the lossless full replay and the lossy collapsed-epochs replays are rather close to the original. In the outbound direction, the results are consistent with the somewhat lower packet throughput for the replay stated in observation P.1.

The tails of the marginals are again surprising for UNC 1 AM, and do not follow observation P.3. The inbound plot shows lossless replays that are significantly heavier than the original, which exhibits the lightest tail. Lossy replays provide far better approximations. The outbound plot appears closer to the previous observation, with the lossless replays being the closest ones to the original. Note however that they are somewhat heavier, unlike in the Leipzig-II and UNC 1 PM cases.

6.5.4 Long-Range Dependence

While the wavelet spectra for the inbound direction shown in Figure 6.35 are in good agreement with observation B.4, we find substantially higher energy above the original in the spectrum of the lossless full replay for outbound direction. The estimated Hurst parameters shown in Table 6.5 are again difficult to assess, as mentioned in that observation. Lossless replays are the only ones within the confidence interval of the estimate for the original inbound direction, while only the lossless collapsed-epochs replay is outside the confidence interval for the outbound direction. Incidentally, the extremely high estimate for the lossless collapsed-epochs replay is remarkable. It is 0.23 above the lossless full replay, illustrating the major difference that detailed source-level modeling can make on traffic long-range dependence.

In the scaling region, collapsed-epochs replays do show higher energy than full replays, as observed in B.5. This higher energy does not translate into higher Hurst parameter estimates. Notice for example the lower estimates for the inbound direction. For both directions, the lossy collapsed-epochs replay provides a good approximation of the original spectrum, although not as good as the lossy full replay. The results for UNC 1 AM are therefore consistent with observation B.7. At the finest scales, we find that the lossy full replay approximates the energy levels of the inbound direction most closely, while it is the lossy collapsed-epochs replay the best match for the outbound direction. This inconsistency supports observation B.6.

Figures 6.37 and 6.38 reveal that the wavelet spectra from lossless replays do not approximate the original spectra well. For both directions, the full lossless replay shows significantly more energy, while the full collapsed-epochs replay shows higher slope in the scaling region. This poor fit for the lossless full replay contradicts observation P.4. Lossy replays appear closer to the original spectra in the scaling region, especially in the case of the lossy full replay. Replays do not consistently match the energy in the fine-scale region, as stated in observation P.5. Lossy replays are the closest ones in this region.

The estimated Hurst parameters shown in Table 6.6 do not follow observation P.4 very clearly. The estimates from the lossless collapsed-epochs replay are far larger than the original estimates. The estimates from the lossless full replays are far lower, but they are still above the upper ends of the confidence intervals. Finally, both lossy replays are within confidence intervals, although the actual

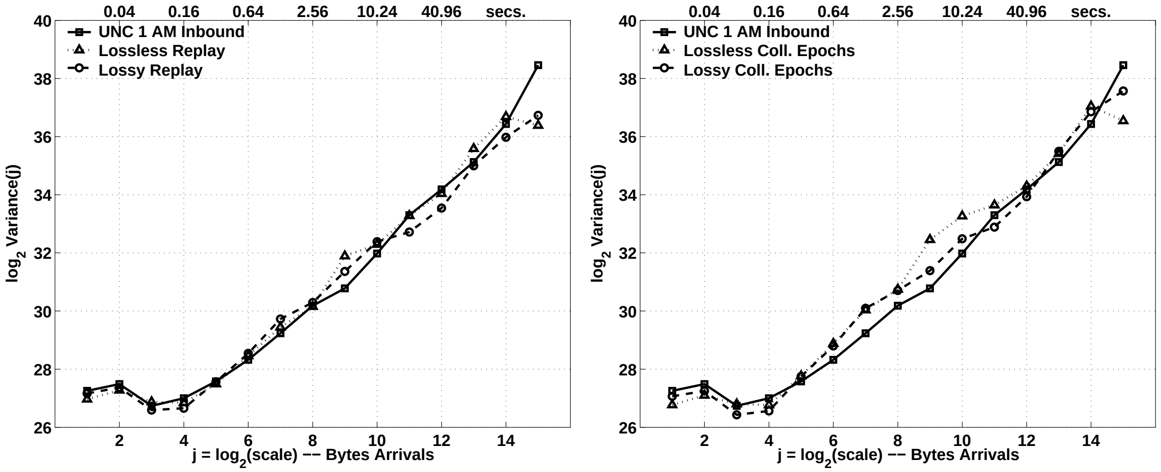


Figure 6.35: Wavelet spectra of the byte throughput time series for UNC 1 AM inbound and its four types of source-level trace replay.

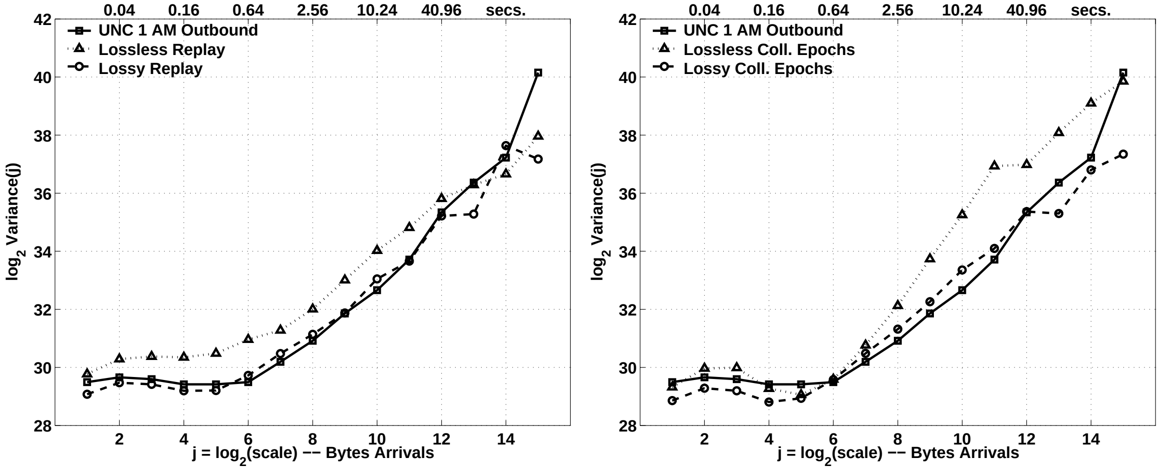


Figure 6.36: Wavelet spectra of the byte throughput time series for UNC 1 AM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 1 AM	*H*	C. I.	*H*	C. I.
Original	0.9885	[0.9479, 1.0292]	0.9990	[0.9584, 1.0397]
Lossless Replay	1.0275	[0.9868, 1.0681]	0.9705	[0.9299, 1.0111]
Lossy Replay	0.9465	[0.9058, 0.9871]	0.9546	[0.9140, 0.9953]
Lossless Coll. Epochs	1.0089	[0.9683, 1.0495]	1.2036	[1.1630, 1.2443]
Lossy Coll. Epochs	0.9136	[0.8730, 0.9542]	0.9720	[0.9313, 1.0126]

Table 6.5: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of UNC 1 AM and its four types of source-level trace replay.

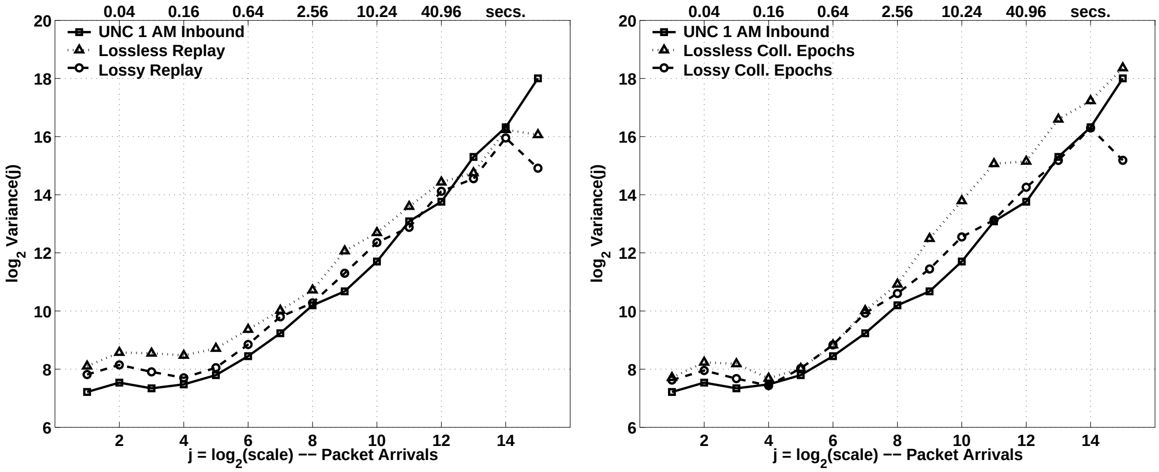


Figure 6.37: Wavelet spectra of the packet throughput time series for UNC 1 AM inbound and its four types of source-level trace replay.

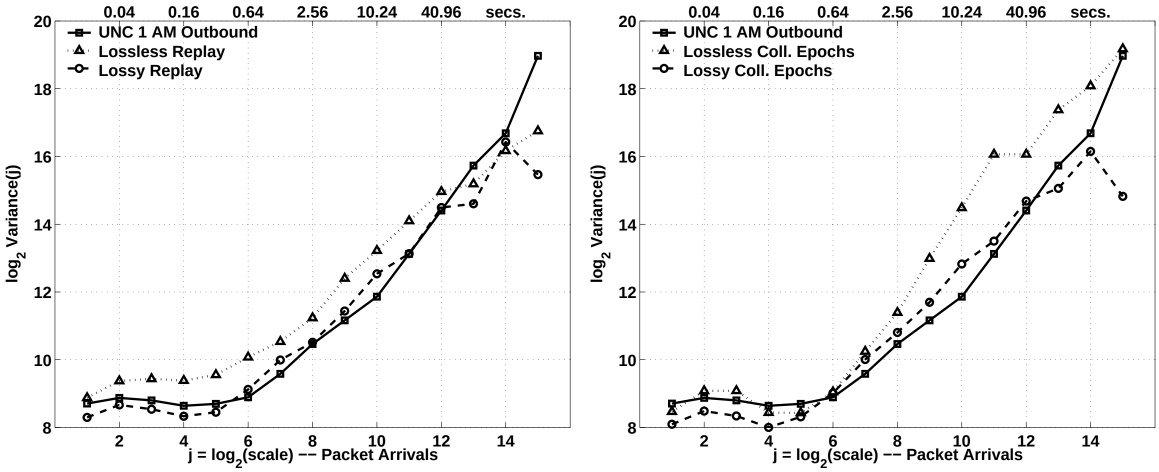


Figure 6.38: Wavelet spectra of the packet throughput time series for UNC 1 AM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 1 AM	H	C. I.	H	C. I.
Original	0.9316	[0.8871, 0.9760]	0.9309	[0.8864, 0.9753]
Lossless Replay	0.9860	[0.9416, 1.0305]	0.9830	[0.9385, 1.0274]
Lossy Replay	0.9749	[0.9304, 1.0193]	0.9759	[0.9315, 1.0204]
Lossless Coll. Epochs	1.1478	[1.1034, 1.1923]	1.2128	[1.1683, 1.2572]
Lossy Coll. Epochs	0.9504	[0.9059, 0.9948]	0.9757	[0.9313, 1.0202]

Table 6.6: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of UNC 1 AM and its four types of source-level trace replay.

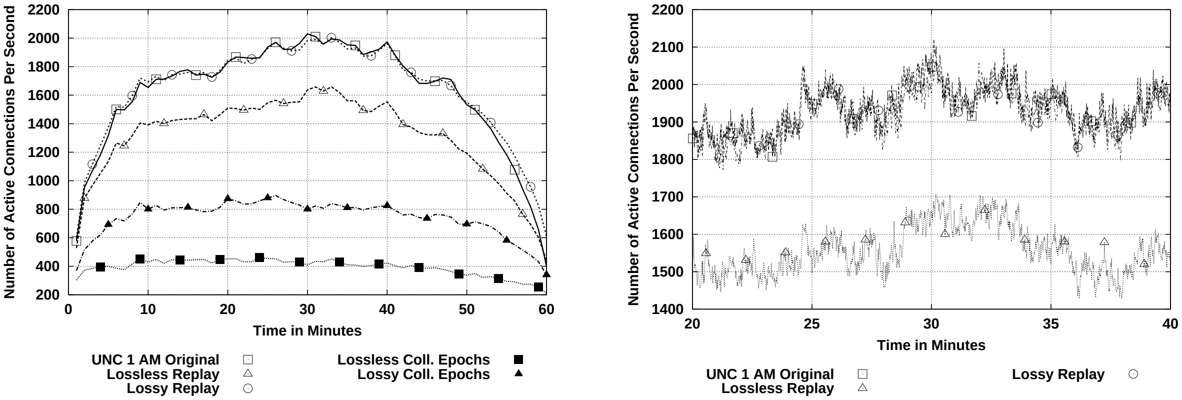


Figure 6.39: Active connection time series for UNC 1 AM and its four types of source-level trace replay.

estimates are higher.

6.5.5 Time Series of Active Connections

The time series of active connections shown in Figure 6.39 confirm the list of observations in Section 6.4. It is clear that observations C.1 and C.2 hold, being the lossy full replay a perfect match of the original time series. Observation C.3 is also true, although the relative gap between the number of connections in full and collapsed-epochs replays is smaller for this trace. The impact of losses is somewhat more significant for the collapsed-epochs replays, which is not completely in agreement with observation C.4. Observations C.5 and C.6 are consistent with the results for UNC 1 AM.

6.6 Source-level Replay of UNC 7:30 PM

6.6.1 Time Series of Byte Throughput

The time series of byte throughput for the inbound and outbound directions of UNC 7:30 PM are shown in Figures 6.40 and 6.41 respectively. Observation B.1 is clearly applicable to these results. For the inbound direction, note the very good approximation of the time series features between minutes 20 and 60, and the accurately reproduced spikes in minutes 46 and 53. In contrast, the replay seems out of phase for the initial spike in minute 1, and the large spike in minute 32. This could be explained by one or a few fast connections in the original trace that could not be replayed fast enough. In the outbound direction, we find replays with somewhat lower byte throughput, which was also observed in the other

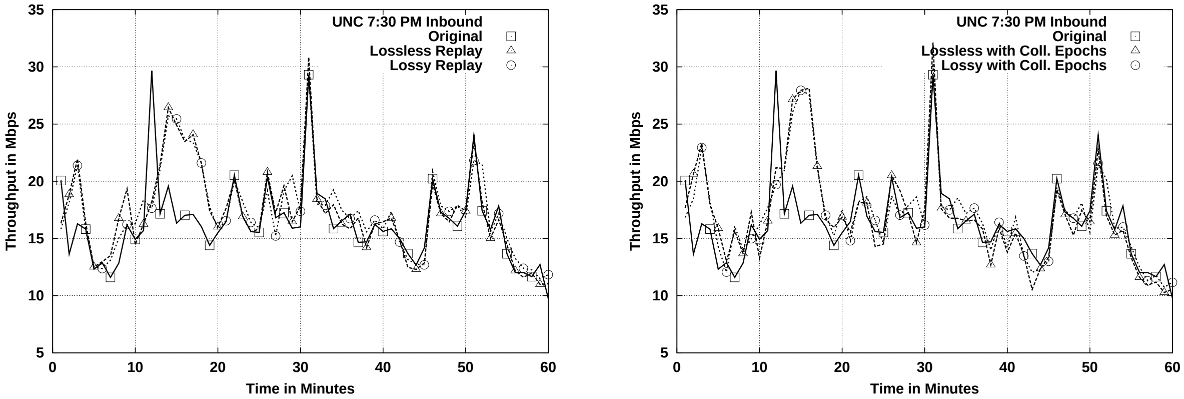


Figure 6.40: Byte throughput time series for UNC 7:30 PM inbound and its four types of source-level trace replay.

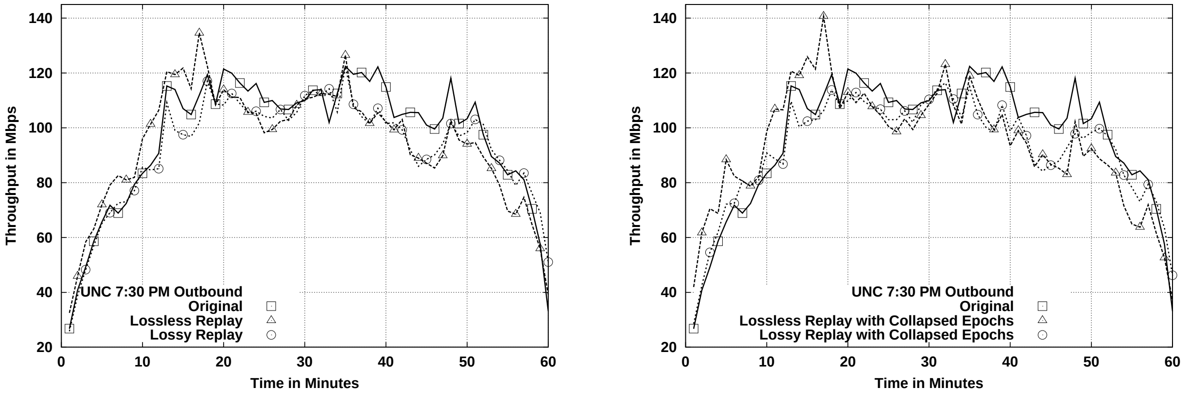


Figure 6.41: Byte throughput time series for UNC 7:30 PM outbound and its four types of source-level trace replay.

two UNC traces.

The possible extra burstiness in lossless replays mentioned in observation B.2 is not present in the inbound direction, and the last 40 minutes in the outbound direction. We do however observe substantially higher throughputs in the outbound direction for the first 20 minutes, especially in the case of the lossless collapsed-epochs replay. Regarding observation B.3, we do observe slightly more bursty time series from the collapsed-epochs replay in both directions, although the difference seems minor in this case. A few of the (smaller) features in the inbound direction are more closely approximated by the full replays, such as the spike in minute 22 and the ditch in minute 43.

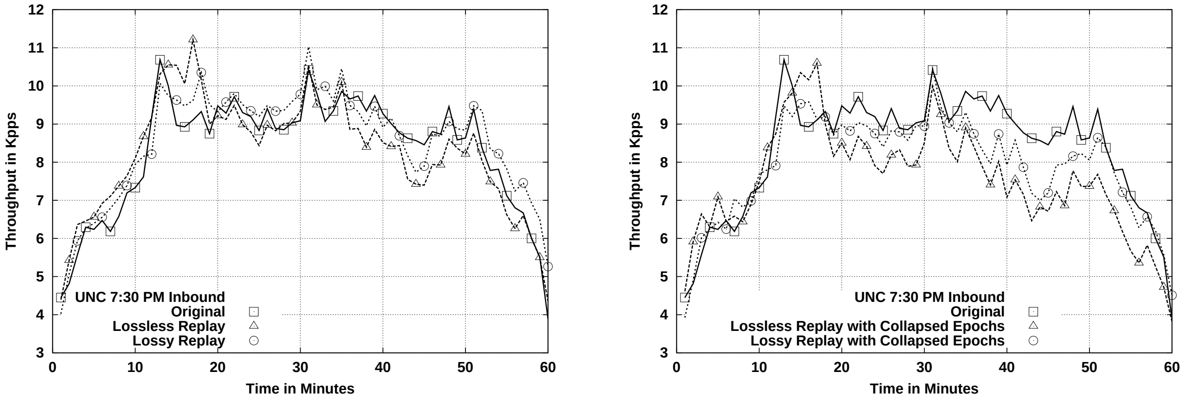


Figure 6.42: Packet throughput time series for UNC 7:30 PM inbound and its four types of source-level trace replay.

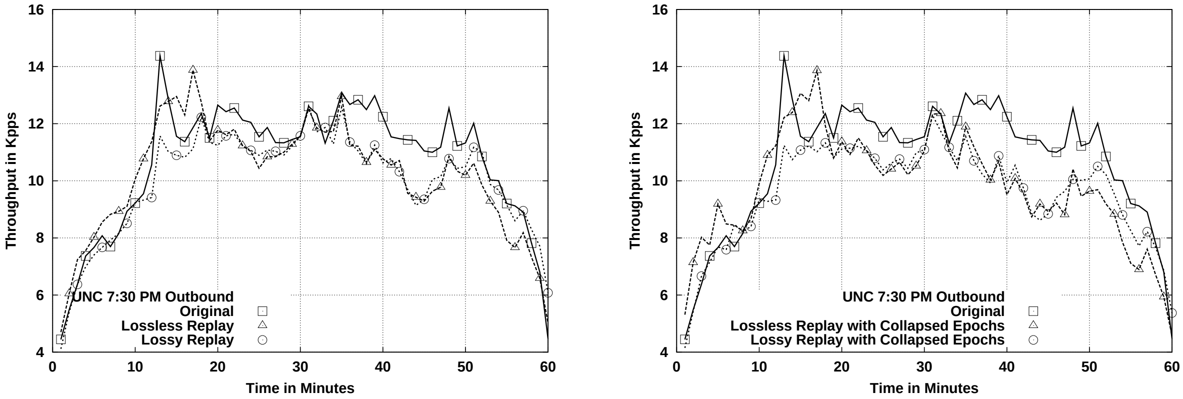


Figure 6.43: Packet throughput time series for UNC 7:30 PM outbound and its four types of source-level trace replay.

6.6.2 Time Series of Packet Throughput

The analysis of the packet throughput results shows again some interesting differences with respect to earlier results and observations P.1 and P.2, but only in the outbound direction. The results for the inbound direction presented in Figure 6.42 are in good agreement with observation B.1, since we observe substantially lower packet throughput for collapsed-epochs replays. The result is also consistent with observation B.2, showing higher packet throughput in the lossy replay. However, the result for the outbound direction is more surprising. Unlike previous cases, losses have a minimal impact on the replays, as shown in Figure 6.43. We could argue that the lossy replay provides a better fit between minutes 30 and 35 and after minute 52, but the rest of the time series for this replay is very similar to the one for the lossless replay. We can also say that the lossless replay makes a better attempt to match the spike in minute 14, although the replay spike seems shifted to minute 16.

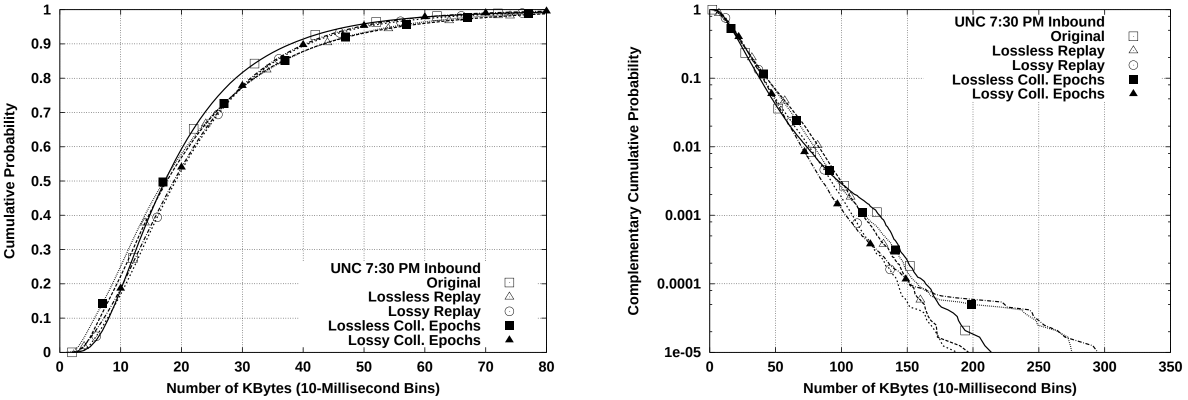


Figure 6.44: Byte throughput marginals for UNC 7:30 PM inbound and its four types of source-level trace replay.

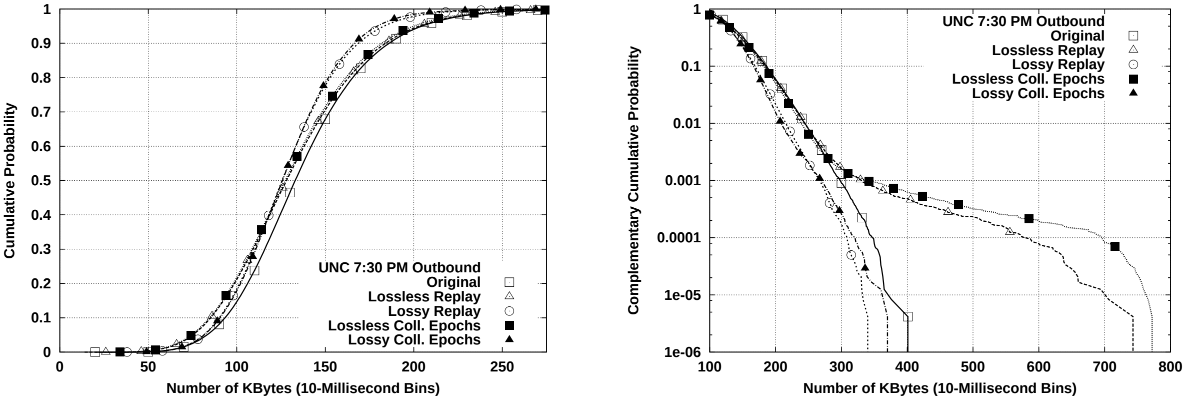


Figure 6.45: Byte throughput marginals for UNC 7:30 PM outbound and its four types of source-level trace replay.

6.6.3 Marginal Distributions

The marginal distributions of the 10-millisecond time series of byte throughput for the inbound direction are shown in Figure 6.44, while the ones for the outbound direction are shown in Figure 6.45. In agreement with observation B.1, the body of the marginals are closely approximated by the replays, especially in the case of the inbound direction. For the outbound, it is interesting to note a better approximation by the lossless replays for the upper part of the body and the first half of the tail.

As observation B.2 and B.3 pointed out, it is difficult to make general a statement about the approximation of the tails. For UNC 7:30 PM inbound, collapsed epoch replays match the original as closely as the full replays that match the original tail below 10^{-4}, but they are substantially heavier above that probability. Note that this heaviness did not manifest itself in the plots of 1-minute byte throughput. For UNC 7:30 PM outbound, we however have that the lossless replays are the ones showing an excellent

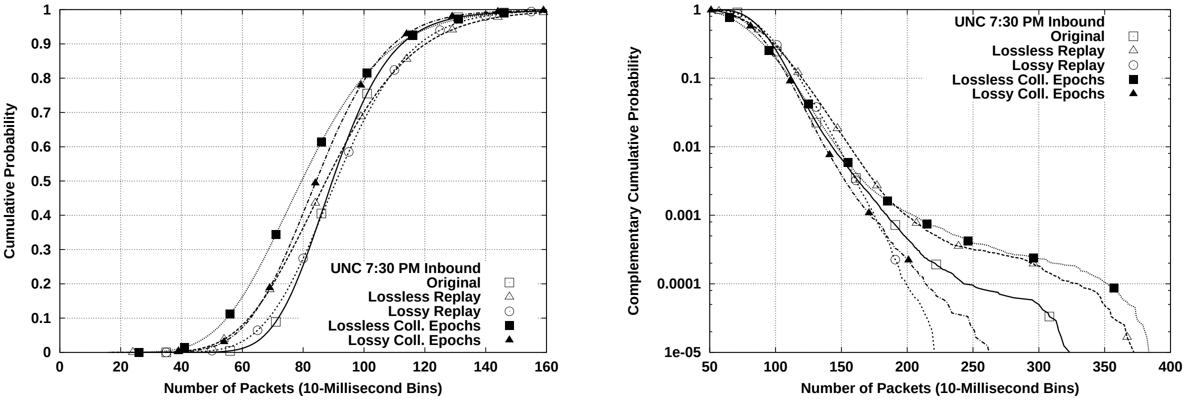


Figure 6.46: Packet throughput marginals for UNC 7:30 PM inbound and its four types of source-level trace replay.

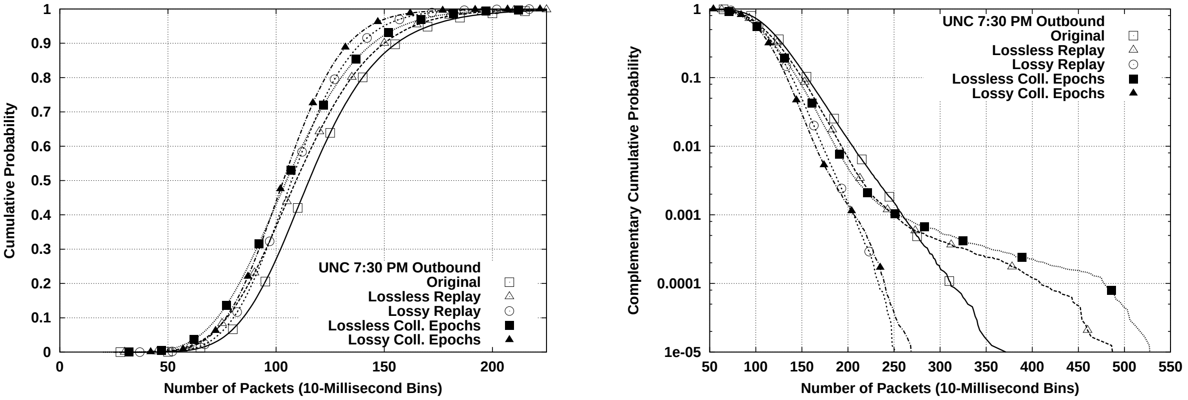


Figure 6.47: Packet throughput marginals for UNC 7:30 PM outbound and its four types of source-level trace replay.

match below 10^{-3}, but a far heavier tail above that probability. Overall, the only type of replay that did not show an overly heavy tail was the lossy full replay, which supports observation B.7.

The body of the marginal distributions for the 10-millisecond packet throughputs shown in Figure 6.46 and 6.47 do not clearly follow observation P.1. In the inbound direction, the distributions from the collapsed-epochs replays are clearly lighter than those from the full replays and the original trace for most of the distribution. However, they provide a better approximation above 100 packets. Interestingly, the distributions from the lossless replays exhibit similar shapes, but a clear offset, and the same is true for the lossy replays. For this direction, we find that the impact of detailed source-level modeling and per-connection losses is of the same order. The lesson is similar for the outbound direction, although all of the replay distributions are lighter than the original distribution in this case.

Regarding the tails of the marginal distributions, we observe similar conditions in both directions. Lossless replays exhibit substantially heavier tails than the original, while lossy replays exhibit substantially lighter tails. This is in sharp contrast to the results for the 1-minute time series studied in Section 6.6.2, where losses had a very small impact. We can argue that lossless replays are artificially more bursty only at fine-time scales for UNC 7:30 PM. The results clearly support observation P.3, showing that the tails are far more sensitive to losses than to detailed source-level modeling for this trace.

6.6.4 Long-Range Dependence

The spectra of the byte throughput in the full replays are close to the original spectrum, as shown in Figure 6.48. However, the spectra of the outbound byte throughput shown in Figure 6.49 reveals a lossless replay with substantially more energy in the scaling region (which starts at octave 6). As in the case of UNC 1 AM outbound, this finding does not support observation B.4 regarding lossless full replays. The lossy full replay provides however a closer approximation to the original spectrum. Estimated Hurst parameters in Table 6.7 show similar results. Estimates from the replays are within the confidence interval of the inbound estimate, but somewhat lower. However, they are above the upper end of the confidence interval of the outbound estimate. The estimate from the lossy collapsed-epochs replay is specially high in this case, probably driven by the spike in octave 11. It is again difficult to draw any strong conclusion other than the general finding of inconsistent results already stated in observation B.4.

Collapsed-epochs replays show substantially more energy in the lossless case, but the difference in not so substantial for the lossy replay, especially in the inbound direction. This is in agreement with observation B.5.

The lossy full replay provides the best approximation again, as pointed out in observation B.7, However, some regions, such as the one between octaves 9 to 12 in the inbound direction, are more closely reproduced by the lossy collapsed-epochs replay. Interestingly, the four replays track the fine-scale energy profile of the original spectrum for the inbound direction, but only the lossless ones do so for the outbound direction. Observation B.6 already reflected this type of inconsistency in the results.

The lessons from the wavelet spectra in Figures 6.50 and 6.51 are surprisingly similar to those from the analysis of the packet throughput spectra for UNC 1 AM, discussed in Section 6.5.4. The full lossless replay shows higher energy, and the full collapsed-epochs replay shows substantially higher slope in the scaling region. Lossy replays provide far better approximations of the original spectra. As we observed for UNC 1 AM, these findings are inconsistent with observation P.4. Regarding fine scale energy levels

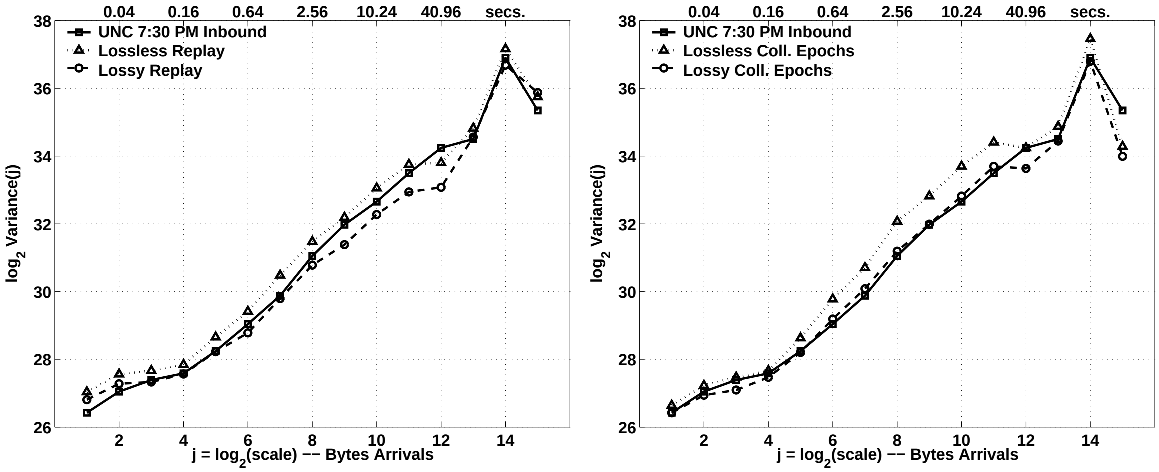


Figure 6.48: Wavelet spectra of the byte throughput time series for UNC 7:30 PM inbound and its four types of source-level trace replay.

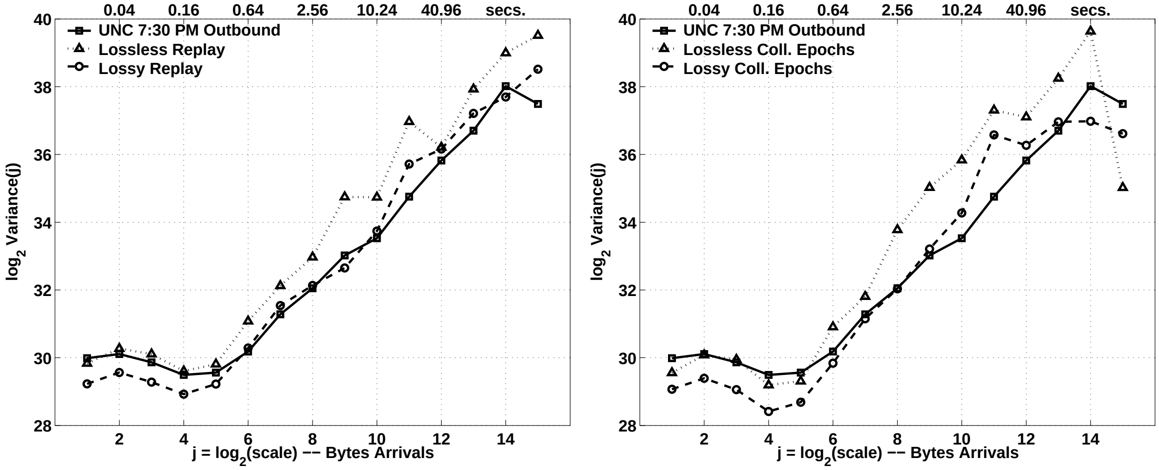


Figure 6.49: Wavelet spectra of the byte throughput time series for UNC 7:30 PM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 7:30 PM	*H*	C. I.	*H*	C. I.
Original	0.8927	[0.8520, 0.9333]	0.9424	[0.9018, 0.9830]
Lossless Replay	0.8490	[0.8083, 0.8896]	1.0191	[0.9784, 1.0597]
Lossy Replay	0.8449	[0.8043, 0.8856]	1.0044	[0.9637, 1.0450]
Lossless Coll. Epochs	0.8392	[0.7985, 0.8798]	0.9984	[0.9578, 1.0390]
Lossy Coll. Epochs	0.8655	[0.8249, 0.9062]	1.0971	[1.0564, 1.1377]

Table 6.7: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of UNC 7:30 PM and its four types of source-level trace replay.

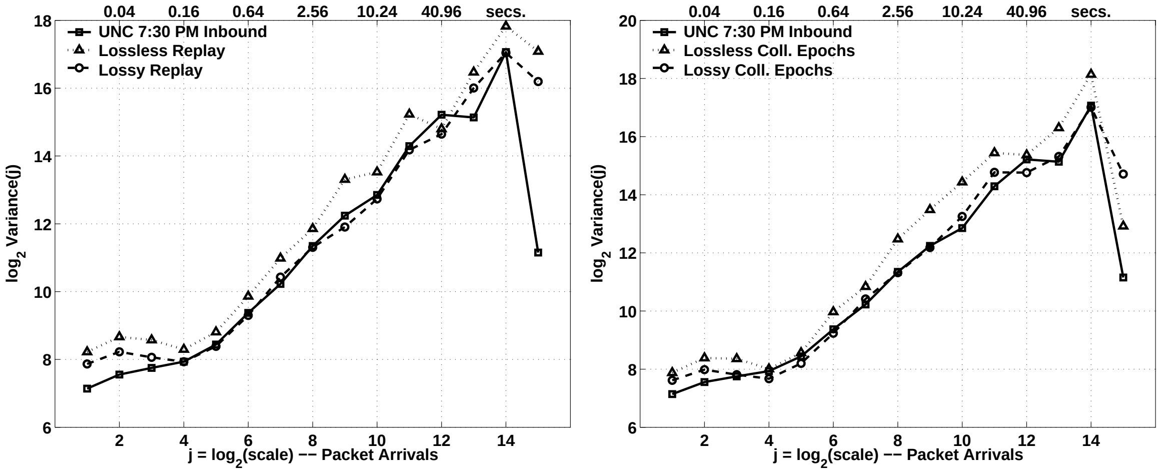


Figure 6.50: Wavelet spectra of the packet throughput time series for UNC 7:30 PM inbound and its four types of source-level trace replay.

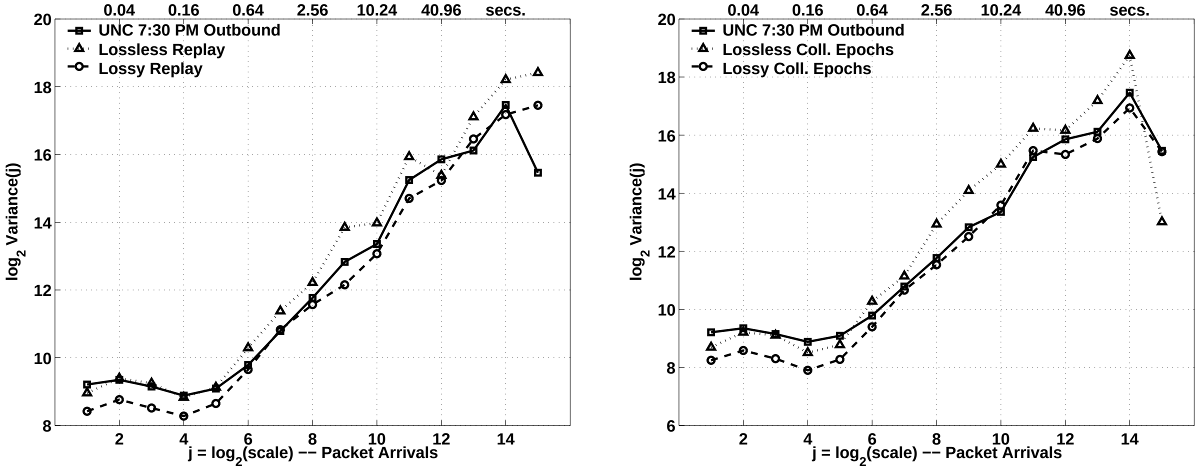


Figure 6.51: Wavelet spectra of the packet throughput time series for UNC 7:30 PM outbound and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
UNC 7:30 PM	*H*	C. I.	*H*	C. I.
Original	0.9560	[0.9116, 1.0005]	1.0061	[0.9617, 1.0506]
Lossless Replay	0.9655	[0.9210, 1.0099]	1.0043	[0.9599, 1.0488]
Lossy Replay	0.9186	[0.8742, 0.9631]	0.9524	[0.9080, 0.9969]
Lossless Coll. Epochs	0.9491	[0.9047, 0.9936]	0.9931	[0.9487, 1.0375]
Lossy Coll. Epochs	0.9967	[0.9523, 1.0411]	1.0508	[1.0064, 1.0953]

Table 6.8: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of UNC 7:30 PM and its four types of source-level trace replay.

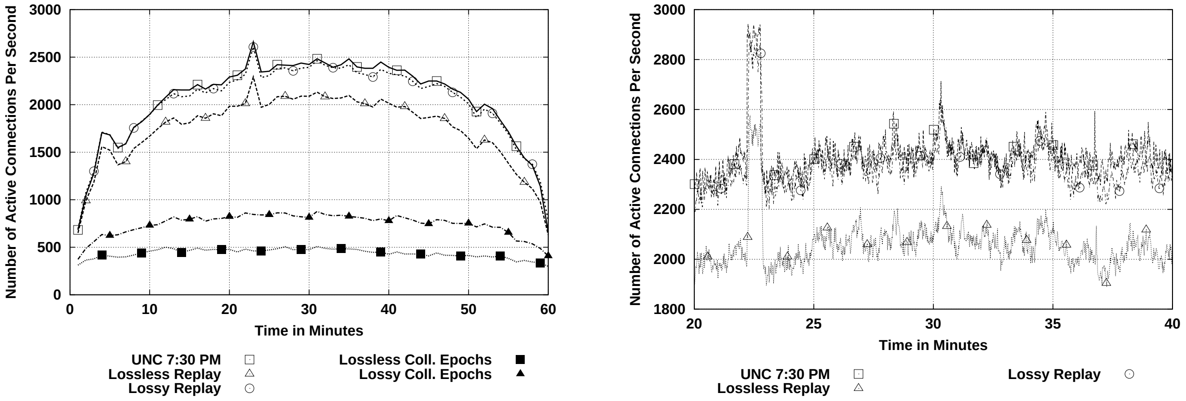


Figure 6.52: Active connection time series for UNC 7:30 PM and its four types of source-level trace replay.

in the outbound direction, lossless replays show higher energy than lossy ones, which are still above the original. In the outbound direction, lossless replays are very close to the original, while lossy ones are below it. Once again, the difficulties for matching fine scale energies mentioned in observation B.6 are present in this trace.

The estimates of the Hurst parameters are not so consistent with the results for UNC 1 AM, and are in better agreement with observation P.4. Here lossless replays approximate the original spectra closely, while lossy replays appear lower (full case) or higher (collapsed-epochs case) than the original.

6.6.5 Time Series of Active Connections

The time series of active connections shown in Figure 6.52 are in good agreement with earlier traces. Every observation listed in Section 6.4.3 is confirmed by the UNC 7:30 PM results. Unlike the two previous UNC traces, the lossy full replay is not a perfect fit of the original time series, but it still provides a very close approximation, well within the 1% bound mentioned in observation C.2. The large spike around minute 23 does not appear in the collapsed-epochs replay, providing another clear illustration of observation C.6. Note that whatever the cause of this spike, it is not due to a difference in the number of connections started, since they are identical in the four replays. The spike is necessarily explained by a set of connections with substantially longer lifespans in the full replays than in the collapsed-epochs replays.

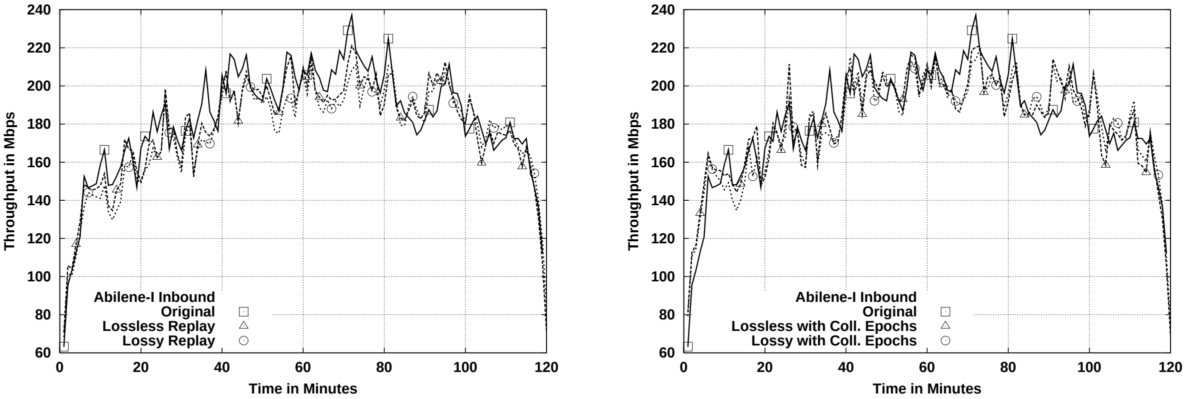


Figure 6.53: Byte throughput time series for Abilene-I Clev/Ipls and its four types of source-level trace replay.

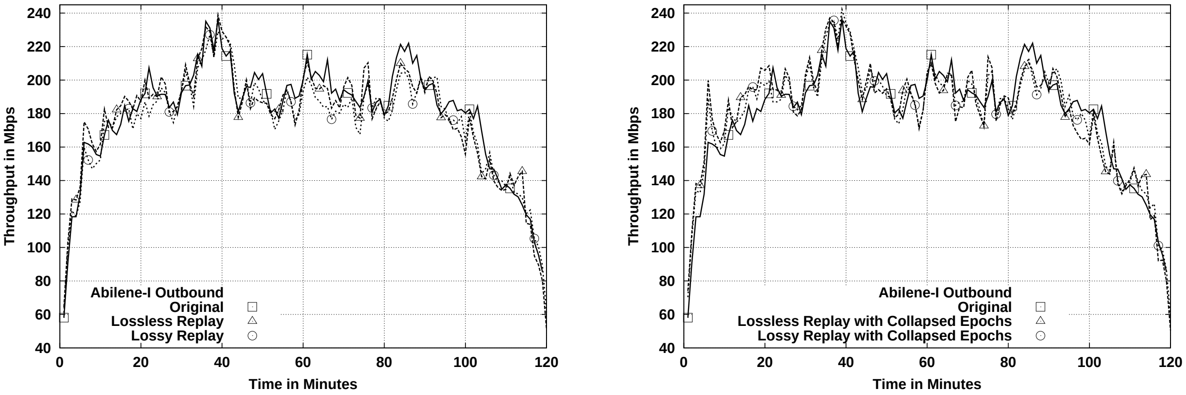


Figure 6.54: Byte throughput time series for Abilene-I Ipls/Clev and its four types of source-level trace replay.

6.7 Source-level Replay of Abilene-I

6.7.1 Time Series of Byte Throughput

The two directions of Abilene-I show the highest throughput of the five traces considered in this chapter. Combined byte throughput is often above 400 Mbps, creating the most challenging traffic generation scenario in terms of traffic volume. The excellent agreement between original and replay data shown in Figures 6.53 and 6.54 provide convincing evidence in favor of observation B.1. The replays closely track the general shape of the time series, even reproducing major changes such as the one between minutes 30 and 42. In general, we observe some spikes that appear in both original and replay time series, while others do not.

Lossless replays and collapsed-epochs replays do not seem to add any significant burstiness for this

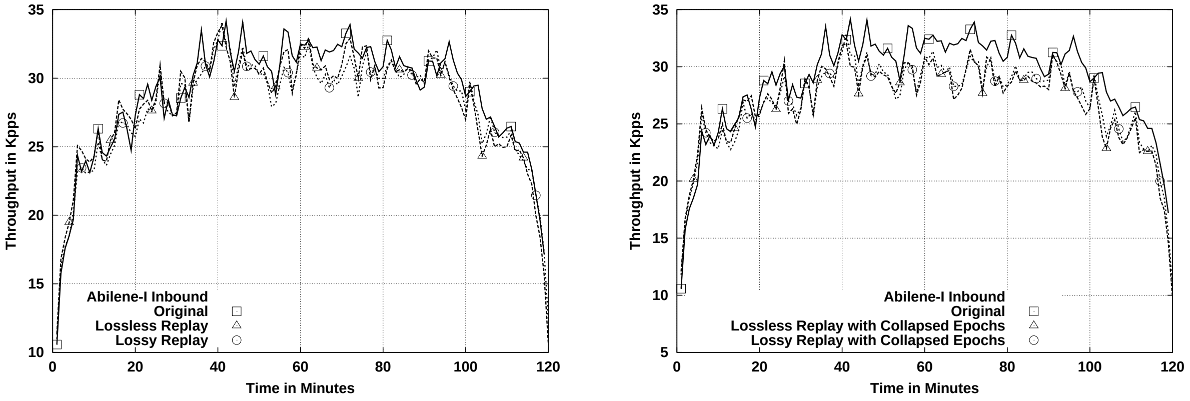


Figure 6.55: Packet throughput time series for Abilene-I Clev/Ipls and its four types of source-level trace replay.

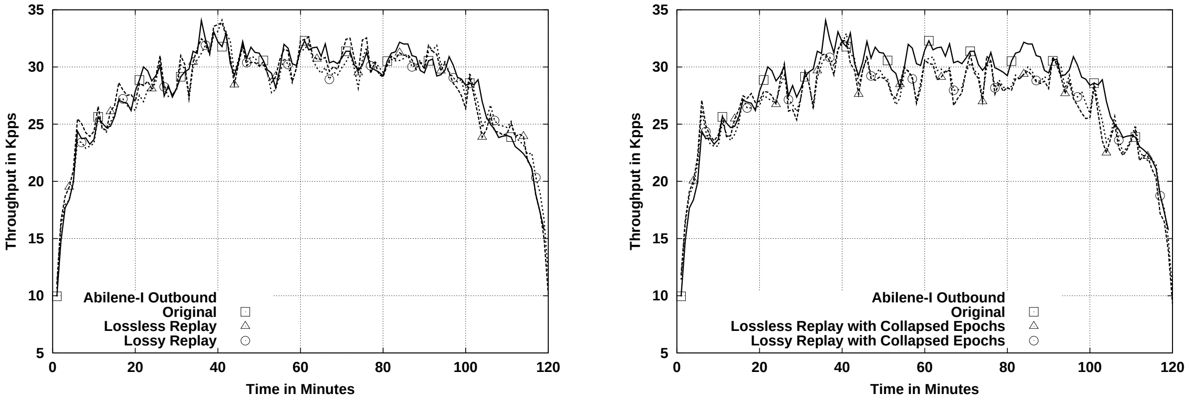


Figure 6.56: Packet throughput time series for Abilene-I Ipls/Clev and its four types of source-level trace replay.

trace, which agrees with the weak statements in observations B.2 and B.3. Note however that the high aggregate throughput could easily be hiding extra burstiness of the magnitude observed for previous traces. For example, careful examination uncovers higher throughput above the original in collapsed-epochs replay, for the spike in minute 7 and for the region between minutes 15 and 30.

6.7.2 Time Series of Packet Throughput

The time series of packet throughput in Figures 6.55 and 6.56 are consistent with observation P.1, showing an excellent match between original and full replays. Given that Abilene-I is the trace with the lowest loss level (see Section 4.1.3), this could suggest that the difficulties with the last two UNC traces were probably due to the complexity of their loss characteristics. Collapsed-epochs replays show a lower packet throughput, generally 2,000 to 3,000 packets below the original. In relative terms, the difference

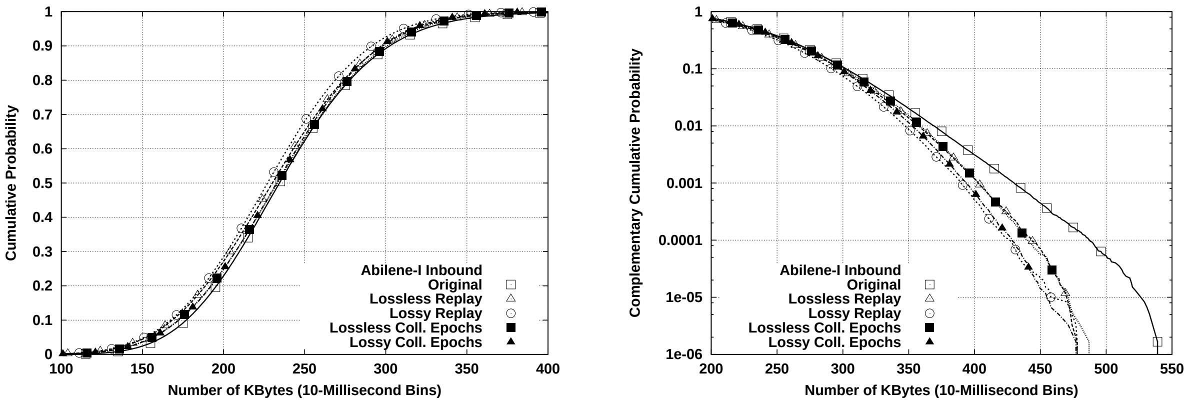


Figure 6.57: Byte throughput marginals for Abilene-I Clev/Ipls and its four types of source-level trace replay.

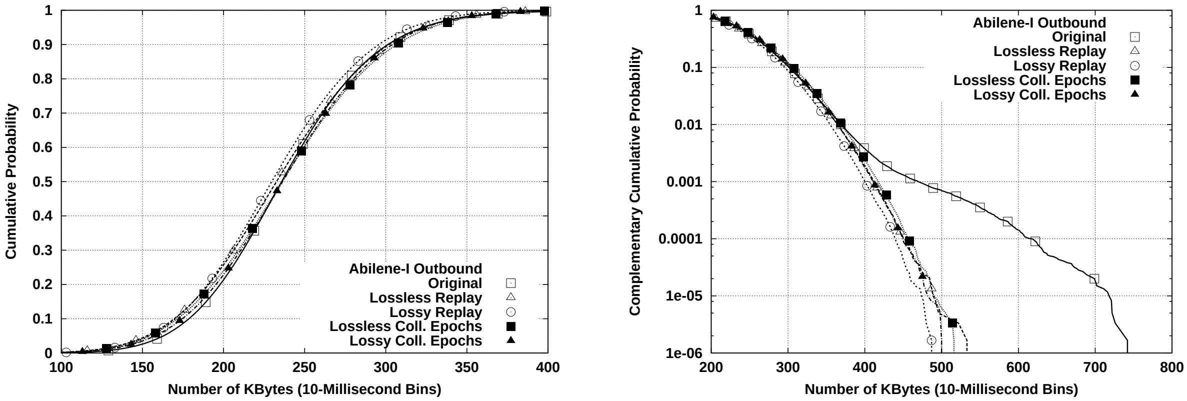


Figure 6.58: Byte throughput marginals for Abilene-I Ipls/Clev and its four types of source-level trace replay.

is between 8% and 10%, which is smaller than for previous traces. This could easily be explained by a larger percentage of bulk transfers in Abilene-I, where a single ADU carrying a single file constitutes the only payload of the TCP connection. This is for example the case in FTP-DATA connections.

6.7.3 Marginal Distributions

The marginal distributions from Abilene-I presented in Figure 6.57 and 6.58 show very similar bodies for original and replay traces. This further confirms observation B.1. Unlike previous traces, we find remarkably similar tails for all four replay traces that are consistently lighter than the original tail. The difference is specially striking in the outbound direction. One possible explanation for this intriguing result for the Abilene-I trace comes from the type of monitored link. Abilene-I is the only trace in this chapter collected in a link technology (OC-48, 2.5 Gbps) different from the one used in the replay

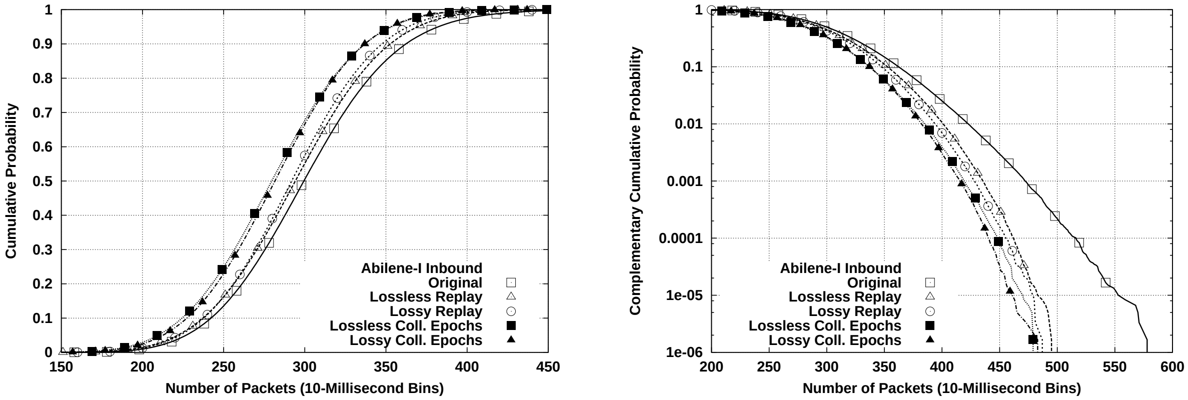


Figure 6.59: Packet throughput marginals for Abilene-I Clev/Ipls and its four types of source-level trace replay.

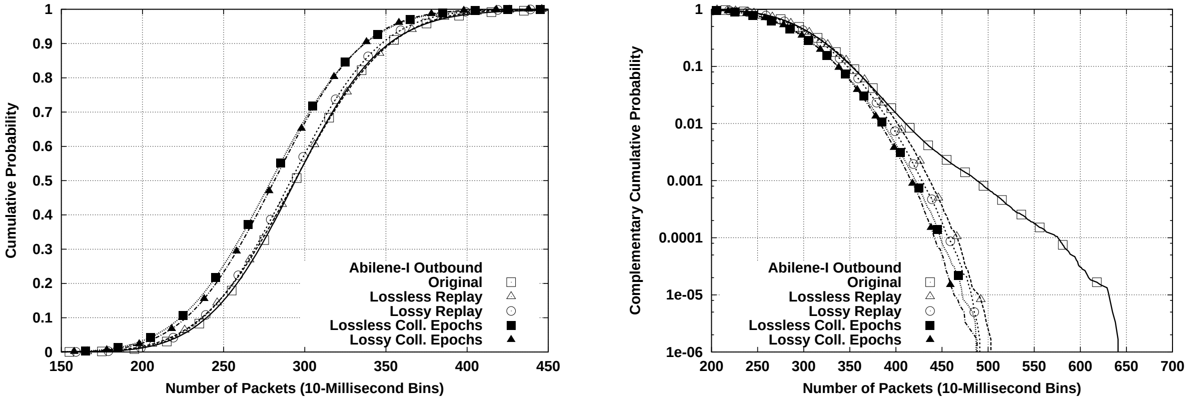


Figure 6.60: Packet throughput marginals for Abilene-I Ipls/Clev and its four types of source-level trace replay.

(Gigabit Ethernet, 1 Gbps). While the plots in Figures 6.53 and 6.54 showed no single minute with more than 500 Mbps, it is perfectly possible to have shorter (*e.g.*, 10 millisecond) intervals with far higher byte throughput. An alternative explanation is the presence of some possible limit in the forwarding capacity of our software routers, which is not far above 500 Mbps.

The bodies of the marginal distributions for packet throughput in Figures 6.59 and 6.60 are consistent with observation P.1. Collapsed-epochs replays show substantially lighter distributions, while full replays are closer to the original. The approximation in the outbound direction is remarkably good. As a consequence of the low loss in this trace, observation P.3 does not apply to Abilene-I. The impact of losses is smaller than the impact of source-level modeling. This effect is however dwarfed by the large difference between the tails of the replays and the original one. As discussed for byte throughput, differences in link technology between Abilene-I and the testbed could explain the far lighter tails in the replays.

6.7.4 Long-Range Dependence

The wavelet spectra for the inbound direction, shown in Figure 6.61, support observation B.4. However, the difference between original and full replays is substantial in the outbound direction, shown in Figure 6.62. Given the major change in slope after octave 11, it is difficult to draw any conclusions from this finding. Regarding observation B.5, we do observe worse approximations by the collapsed-epochs replays, which exhibit substantially deeper ditches around octave 5 (notice the lower smallest value in the y-axis of the outbound plot). In any case, the replays do not closely track the fine scale shape of the spectra, which is in agreement with observation B.6.

Hurst parameters, shown in Table 6.9, are remarkably high for this trace. All of them are above 1, suggesting significant non-stationarity, which is clearly preserved in the replays. The estimate for the lossy full replay is the closest one for both directions. Together with the wavelet spectra, this supports observation B.7.

As for byte throughput, the wavelet spectra of the packet throughput for Abilene-I shown in Figures 6.63 and 6.64 are not comparable to previous cases, and inconsistent with observation P.4. The difference between the replays and the original follows the same pattern in all of the cases, with a large ditch in octave 5. This ditch is much more pronounced for the collapsed-epochs replays. In any case, the result is a poor match between the original spectra and the replays, both at fine scales and at the the scaling region. Estimated Hurst parameters for the replays are lower for the inbound replays and higher for the outbound ones, and mostly outside confidence intervals. Incorporating losses had a minimal impact on the wavelet spectra of the Abilene-I replays, resulting only in a small decrease of the slope in the scaling region. This decrease translated into a slightly smaller Hurst parameters estimates for the lossy replays.

6.7.5 Time Series of Active Connections

Unlike the results for previous traces, Figure 6.65 shows a substantial difference between the lossy full replay and the original time series. This weakens observations C.1 and C.2 from Section 6.4.3, being the replay around 15% below the original. The rest of the observations clearly hold. The relative magnitude of the gap between the full replays and the collapsed-epochs ones is largest for Abilene-I. The reason is unclear, especially given the excellent approximations for the other traces. It is hard to imagine a larger fraction of bandwidth-constrained connections in this trace, and round-trip time estimation should be as accurate as for the other traces. We are more inclined to think that the mix of applications in

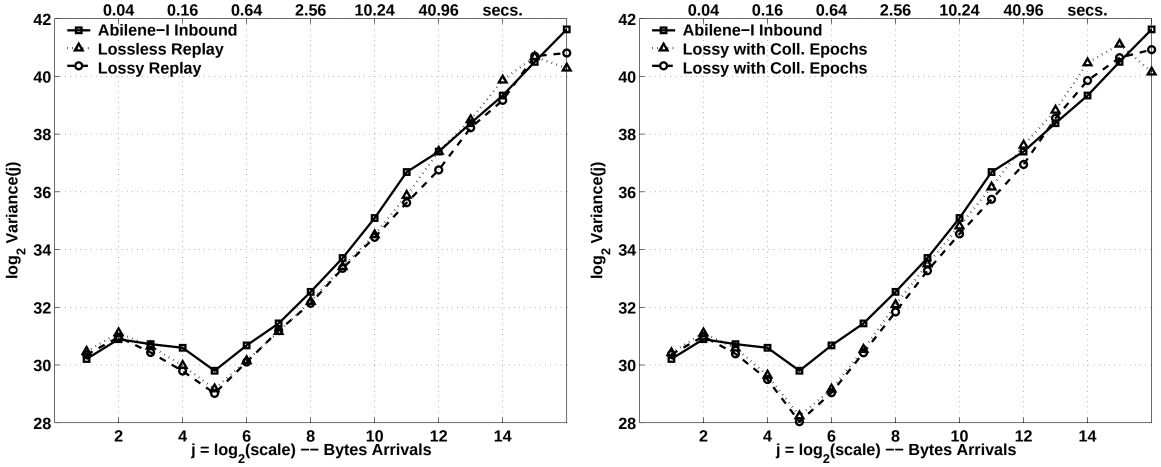


Figure 6.61: Wavelet spectra of the byte throughput time series for Abilene-I Clev/Ipls and its four types of source-level trace replay.

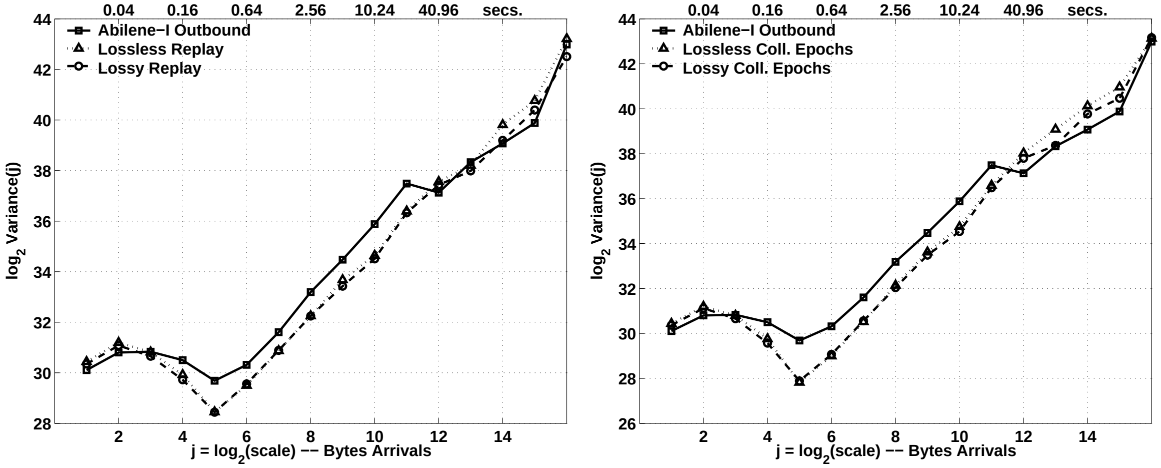


Figure 6.62: Wavelet spectra of the byte throughput time series for Abilene-I Ipls/Clev and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
Abilene-I	*H*	C. I.	*H*	C. I.
Original	1.0597	[1.0320, 1.0874]	1.0604	[1.0327, 1.0881]
Lossless Replay	1.1170	[1.0893, 1.1447]	1.1356	[1.1079, 1.1633]
Lossy Replay	1.0814	[1.0537, 1.1091]	1.1079	[1.0802, 1.1356]
Lossless Coll. Epochs	1.1824	[1.1573, 1.2075]	1.2111	[1.1860, 1.2362]
Lossy Coll. Epochs	1.1580	[1.1329, 1.1831]	1.1874	[1.1623, 1.2125]

Table 6.9: Estimated Hurst parameters and their confidence intervals for the byte throughput time series of Abilene-I and its four types of source-level trace replay.

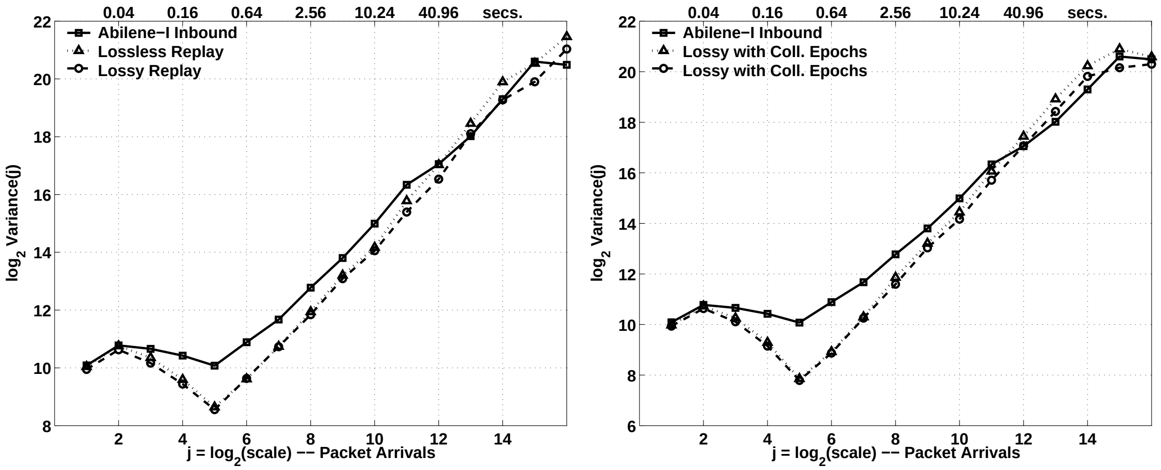


Figure 6.63: Wavelet spectra of the packet throughput time series for Abilene-I Clev/Ipls and its four types of source-level trace replay.

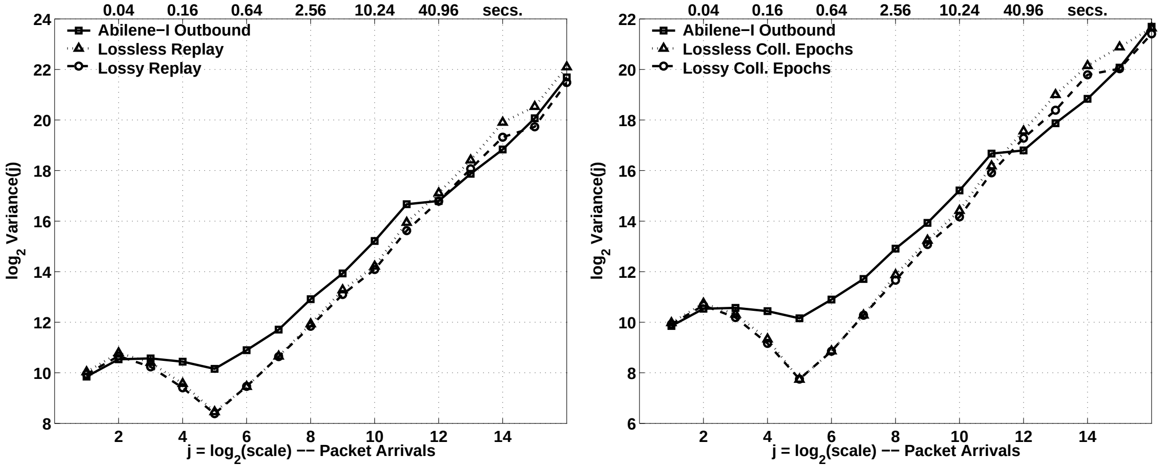


Figure 6.64: Wavelet spectra of the packet throughput time series for Abilene-I Ipls/Clev and its four types of source-level trace replay.

Trace	**Inbound**		**Outbound**	
Abilene-I	H	C. I.	H	C. I.
Original	1.1326	[1.1075, 1.1577]	1.0996	[1.0745, 1.1247]
Lossless Replay	1.1191	[1.0941, 1.1442]	1.1443	[1.1192, 1.1694]
Lossy Replay	1.0849	[1.0598, 1.1100]	1.1232	[1.0981, 1.1483]
Lossless Coll. Epochs	1.1841	[1.1563, 1.2118]	1.1923	[1.1646, 1.2200]
Lossy Coll. Epochs	1.1757	[1.1480, 1.2034]	1.1850	[1.1573, 1.2127]

Table 6.10: Estimated Hurst parameters and their confidence intervals for the packet throughput time series of Abilene-I and its four types of source-level trace replay.

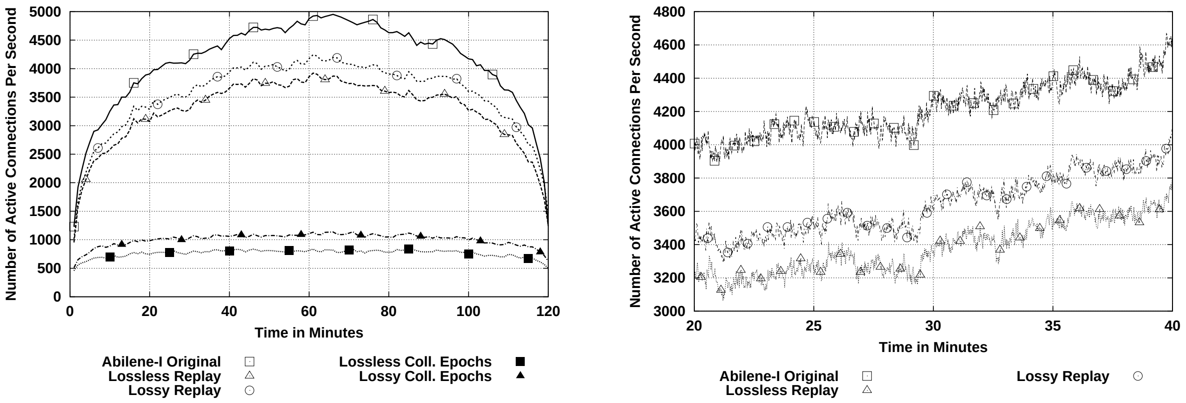


Figure 6.65: Active connection time series for Abilene-I and its four types of source-level trace replay.

Abilene-I includes a substantial number of (probably long) connections whose driving application is not well-described by our source-level model.

6.8 Summary

The results in this chapter demonstrated that source-level trace replay can closely approximate the characteristics of real traffic traces. We have also shown that full source-level replays are closer or far closer to original traces than collapsed-epochs replays for several metrics. In particular, the largest difference is observed for the time series of packet throughput, the body of the packet throughput marginal and the time series of active connections. Byte throughput is similar for full and collapsed-epochs replays. The latter exhibits somewhat more bursty time series, but the bodies of the marginals do not change significantly.

The rest of the metrics cannot be clearly interpreted, since losses have a much more significant impact on them than the use of full or collapsed-epochs replays. Lossy full replays are clearly better than lossy collapsed-epochs replays in terms of wavelet spectra, estimated Hurst parameters and tails of the marginals for some traces, but this is not consistent for the five traces. Our analysis clearly demonstrated the need to carefully consider the impact of losses on evaluating the quality of synthetic traffic. Without our direct comparison of lossless and lossy replays, the results for certain metrics could have mislead our conclusions regarding source-level modeling. In contrast, other metrics are less affected by the loss model. This is the case for the time series of packet throughput, the body of the packet throughput marginal and the time series of active connections, where full replays are clearly better approximations than collapsed-epochs replays.

CHAPTER 7

Trace Resampling and Load Scaling

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.

— John A. Locke (1632–1704)

Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted.

— Albert Einstein (1879–1955)

The previous chapters presented a complete methodology for reproducing the traffic observed on a network link in a closed-loop manner, and proposed a number of metrics for studying the realism of the generated traffic. In this chapter, we study ways to introduce statistical variability in synthetic traffic in a meaningful and controlled manner. In addition, we address the need for changing offered load in network experiments. The methods that we introduce in this chapter add significant flexibility to our traffic generation approach, enabling researchers to perform a wider range of experiments.

In the approach presented so far, traffic is generated according to a trace $\mathcal{T}_c = \{(T_i, C_i)\}$. Each augmented connection vector C_i is replayed starting at time T_i. This implies that two different replays of $\mathcal{T}_c$ using the same hardware and the same physical network result in very similar synthetic traffic. In both cases, the synthetic traffic has the same number of TCP connections, replaying the same source-level behaviors under the same network-level parameters, and starting exactly at the same times. Only tiny variations would be introduced on the end-systems by changes in clock synchronization, operating system scheduling and interrupt handling, and at switches and routers by the stochastic nature of packet multiplexing. This reproducibility was exactly what was needed to evaluate how well synthetic traffic approximated the real traffic from which it derived.

However, in the practice of experimental networking, experimenters often want to introduce more variability in their experiments. One way of accomplishing this is to use more than one trace in a replay, exposing the studied network protocol or mechanism to different types of workloads. This is highly desirable, but it has its drawbacks. First, the experimenter may want to perform a number of experiments that is larger than the number of available traces. Second, traces from different sites, and even traces from the same site but collected at different times of the day, may be so different that it becomes difficult to extrapolate from the results of the experiments.

A different, and complementary, approach is to conduct several experiments using traffic that "looks like" some specific trace $\mathcal{T}_c$, without exactly replaying $\mathcal{T}_c$ over and over. The first challenge in devising a method for accomplishing this task is to define what "looks like" mean. This involves creating a model (either formal or informal) of the traffic which is general enough to contain $\mathcal{T}_c$ but specific enough to always *resemble* the original trace. Running different experiments then requires to instantiate this model several times to create new derived traces $\mathcal{T}_c'$, $\mathcal{T}_c''$, ... and to generate traffic with these new traces using their source-level trace replay. In this chapter, this instantiation consists of resampling the set of connection vectors in $\mathcal{T}_c$ and assigning them new start times. Statistical variability in the derived traces comes from the resampling of the original connection vectors, and from the process of connection start times. We preserve the statistical properties of the original set of connection vectors by resampling *entire* connection vectors, *i.e.*, we do not manipulate the sizes and order of the ADUs and inter-ADU quiet times inside connection vectors. Our belief is that a trace created by modifying the source-level behavior of the connection vectors or their network-level parameters "does not look like" the original trace. For example, doubling the size of the ADUs in $\mathcal{T}_c$ is an easy way of creating a new trace and increasing the offered load. However, the resulting connection vectors have little to do with the connections observed in the link from which $\mathcal{T}_c$ was collected. Our choice to maintain connection vectors intact is reasonable, and consistent with the spirit of our overall methodology, which goes to great lengths to accurately characterize the source-level characteristics of each connection. Other researchers may have a different mental model of the legitimate level of statistical variability which could be introduced in $\mathcal{T}_c', \mathcal{T}_c'', \ldots$ We propose a specific solution and demonstrate that it is reasonable using quantitative data. A discussion of the different philosophies is outside the scope of this work.

The two sections in this chapter describe two techniques for introducing variability in the source-level replay of a trace. Section 7.1 describes *Poisson Resampling*. This technique assumes that connections are independent of each other, which is a reasonable choice for highly aggregated traffic. Poisson Resampling involves randomly resampling the connection vectors in $\mathcal{T}_c$ in an independent manner to create a new $\mathcal{T}_c'$. New start times are given to each resampled connection vector in a such a way that connection inter-arrivals are exponentially distributed. As we will show, empirical data support the choice of exponential inter-arrivals.

Section 7.2 describes *Block Resampling*. This technique involves resampling blocks of connection vectors, preserving arrival dependencies among the connections inside the same block. Each block is the set of connections observed in an interval of fixed duration (*e.g.*, 1 minute) in the original trace. We will demonstrate that this technique, unlike Poisson Resampling, preserves the long-range dependence in the connection arrival process found in real traces. This cannot be achieved by sampling independently from an exponential (or any other) distribution. Note that we will reserve the term *resampling* for randomly selecting connection vectors, and the term *sampling* for randomly drawing values from a parametric distribution, such as the exponential distribution.

The second topic of this chapter is how to manipulate a trace $\mathcal{T}_c$ to modify the traffic load (*i.e.*, average byte throughput) that this trace offers during its source-level replay. This is a common need in experimental networking research, where the performance of a network mechanism or protocol is often affected by the amount of traffic to which it is exposed. For example, active queue management mechanisms have very different performance depending on the level of utilization of the input link, so researchers generally perform experiments with offered loads that range from 50% to 120% of the link bandwidth. Rather than trying to find or collect traces with the exact range of loads needed (which is generally difficult), we propose to produce a collection of resampled traces with the intended range of offered loads.

Average load l is defined as the total number of bytes injected in the network s divided by the total duration of the experiment d. Changing the average load in an experiment of constant duration therefore implies creating a *scaled trace* $\mathcal{T}_c'$ with a higher or a lower total number of bytes. Once again, the assumption is that it is possible to create a scaled trace $\mathcal{T}_c'$ which "looks like" the original $\mathcal{T}_c$ but with a larger or smaller number of bytes. This requires a model of traffic that is general enough to encompass $\mathcal{T}_c$ and traces derived from $\mathcal{T}_c$ with different offered loads. As should be obvious, the problems of introducing statistical variability and changing average load are related, and can naturally be treated together, as we will do in this chapter. The two techniques mentioned above, Poisson Resampling and Block Resampling, provide the foundation for deriving scaled traces. In both cases, the resampling of $\mathcal{T}_c$ to create a scaled $\mathcal{T}_c'$ can be modified to achieve a target average load. This means that our scaling method is *predictable*, which is an advance over earlier traffic generation methods, *e.g.*, [CJOS00, LAJS03, SB04]. These earlier methods required a separate experimental study, a *calibration*, to construct a function coupling the parameters of the traffic generator and the achieved load. For example, web traffic generators usually require a calibration to discover the relationship between average load and the number of user equivalents. The scaling methods presented in this chapter eliminate the need for this extra study. Their starting point is the observation that the average load offered by the source-level replay of $\mathcal{T}_c$ is a deterministic function of the total number of bytes in the ADUs of $\mathcal{T}_c$. We will show that these observation holds true using numerical simulations and testbed experiments. In contrast, the same analysis will demonstrate that the average load offered by the replay of $\mathcal{T}_c$ is not strongly correlated with its number of connections. In the case of Poisson Resampling, our method to construct a new trace $\mathcal{T}_c'$ with a specific target offered load involves resampling $\mathcal{T}_c$ until the desired total number of bytes (coming from ADUs) is reached. In the case of Block Resampling, constructing a new trace $\mathcal{T}_c'$ with a specific target offered load involves subsampling blocks ("thinning") to decrease load, or combining two or more blocks ("thickening") to increase load.

7.1 Poisson Resampling

7.1.1 Basic Poisson Resampling

The first technique we consider for introducing variability in the traffic generation process is *Poisson Resampling.* The starting point of every method presented in this chapter is a connection vector trace $\mathcal{T}_c = \{(T_i, C_i) \,|\, i = 1, 2, \ldots, n\}$ where C_i is an augmented connection vector (an a-b-t connection vector plus some network-level parameters), and T_i is its relative start time. The basic version of our Poisson Resampling technique consists of deriving a new trace $\mathcal{T}_c' = \{(T_j', C_j') \,|\, i = 1, 2, \ldots, n'\}$ by randomly choosing connection vectors from $\mathcal{T}_c$ without replacement, and assigning them start times according to an exponential distribution. We define the duration d of $\mathcal{T}_c$ as $T_n - T_1$, the length of the interval in which connections are started[1]. Given a target duration d' for $\mathcal{T}_c'$, the Poisson Resampling algorithm iteratively adds a new (T_j', C_j') to $\mathcal{T}_c'$ until $T_j' > d'$. Each C_j' is equal to some randomly selected C_i, and

$$T_j' = T_{j-1}' + \delta_j,$$

where δ_j is sampled independently from an exponential distribution. The mean μ' of this exponential distribution provides a way to control the *density* of connections in the derived trace. For example, if we intend to have the same density of connections in $\mathcal{T}_c'$ as in $\mathcal{T}_c$, we can compute the mean inter-arrival

[1] This duration is always slightly below the true duration of the original packet header trace, since at least the packets of the last connection started are observed after its start time.

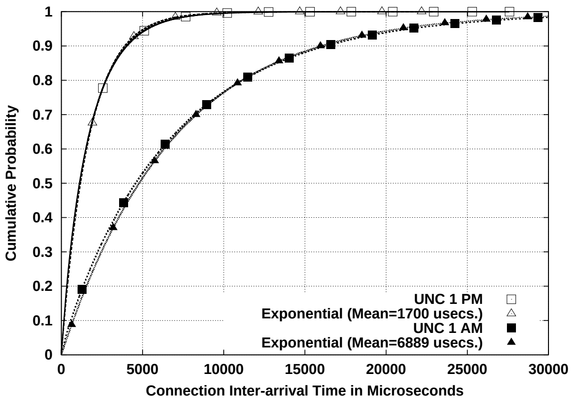


Figure 7.1: Bodies of the distributions of connection inter-arrivals for UNC 1 PM and 1 AM, and their exponential fits.

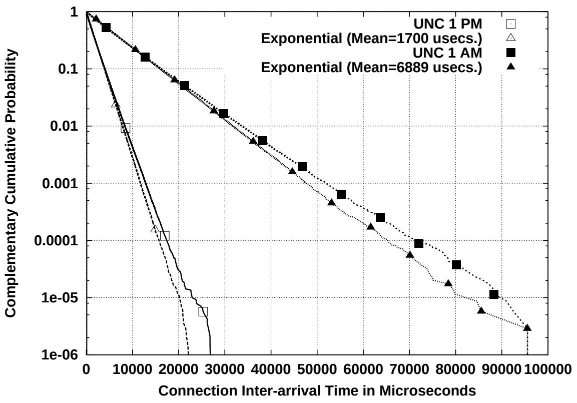


Figure 7.2: Tails of the distributions of connection inter-arrivals for UNC 1 PM and 1 AM, and their exponential fits.

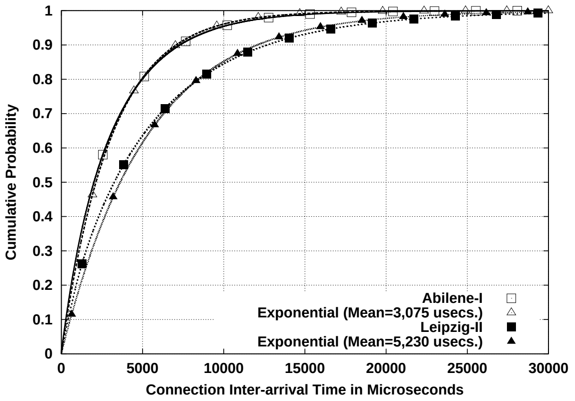


Figure 7.3: Bodies of the distributions of connection inter-arrivals for Abilene-I and Leipzig-II, and their exponential fits.

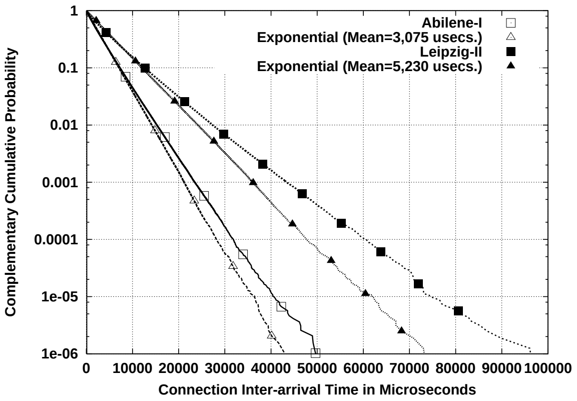


Figure 7.4: Tails of the distributions of connection inter-arrivals for Abilene-I and Leipzig-II, and their exponential fits.

time $\mu = d/n$ of the connection vectors in $\mathcal{T}_c$, and use it as the mean μ' of the experimental distribution used to construct $\mathcal{T}_c'$. Given the light tail of the exponential distribution, the resulting number n' of connection vectors in $\mathcal{T}_c'$ is always very close to d'/μ'. If $d = d'$, the number of connection vectors in $\mathcal{T}_c'$ is also very close to n.

The resampling technique described above has the advantage of its simplicity. Furthermore, it is statistically appealing, since the exponential distribution naturally arises from the combination of independent events. The use of connection inter-arrivals sampled independently from an exponential distribution is intuitively consistent with the view of traffic as a superposition of a large number of independent connections that transmit data through the same network link. Our empirical data, presented in Figures 7.1 to 7.4, confirm the applicability of this inter-arrival model. Figure 7.1 shows two pairs of CDFs comparing real connection arrivals and their exponential fits. The first pair (white symbols) corresponds to the distribution of connection inter-arrivals for UNC 1 PM (squares), and an exponential distribution[2] with the same mean (triangles). The second pair (black symbols) shows the distribution of connection inter-arrivals for UNC 1 AM and an exponential distribution with the same mean. Both fits are excellent, so exponentially distributed connection inter-arrivals are clearly a good starting point for a trace resampling technique. The tails of the empirical distributions, shown in Figure 7.2, are also consistent with the fitted exponentials. Their slope is slightly lower, which could motivate a fit with a more general distribution like Weibull. However, a small improvement in the fit would require an increase in the complexity of the model, since the one-parameter exponential model would have to be replaced by the two-parameter Weibull model. This additional effort would produce only a limited gain given that the exponential fit is excellent for 99.9% of the distribution.

Figures 7.3 and 7.4 consider another two traces, Abilene-I and Leipzig-II. The bodies are again very closely approximated, but the tails are heavier for the original data. Note that this effect is more pronounced as the traces get longer. The duration of the UNC traces is one hour, the duration of Abilene-I is 2 hours, and the duration of Leipzig-II is 2 hours and 45 minutes. This could suggest that the worse fit is due to non-stationarity in the connection arrival process, which becomes more likely for longer traces. Further analysis is needed to confirm this hypothesis or find an alternative explanation. We must note that these results are in sharp contrast with those in Feldmann [Fel00], where the empirical inter-arrival distributions were significantly different from the bodies[3] of fitted exponential distributions. The reason for this difference is unclear at this point[4].

The main problem with the basic Poisson Resampling technique is the lack of control over the load offered by the replay of $\mathcal{T}_c'$. As we will demonstrate, the number of connections in $\mathcal{T}_c'$ is only loosely correlated to the offered load. As a consequence, it becomes difficult to predict the load that a Poisson resampled trace generates, even for resamplings of the same duration and mean inter-arrival rate. This would force researchers to create many resampled traces until they hit upon a resampling with the intended offered load. We studied the wide range of offered loads that result from basic Poisson Resampling by performing a large of number of resamplings of the same connection vector trace $\mathcal{T}_c$.

As discussed in the introduction of this chapter, average load l created by $\mathcal{T}_h$ is equal to the total number of bytes s in $\mathcal{T}_h$ divided by its duration d. Given that TCP headers and retransmitted segments

[2] The shown exponential distribution comes from randomly sampling the theoretical distribution $n-1$ times.

[3] The tails were not studied in that paper.

[4] Besides problems with the fitting or the data acquisition in the paper, we conjecture that this could be due to the slightly different type of data we considered in our study. Our connections were fully captured and included only those connections that actually carried data. Those in [Fel00] included degenerate cases in which no data was transferred.

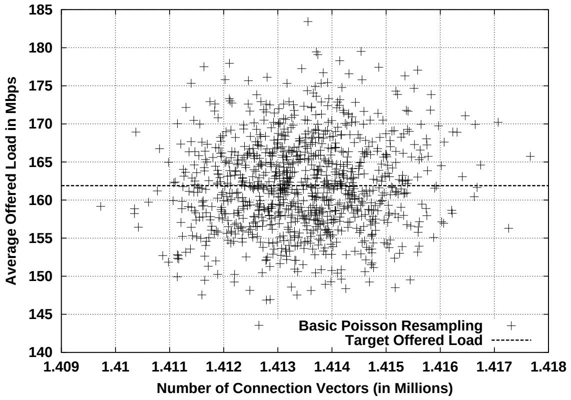


Figure 7.5: Average offered load *vs.* number of connections for 1,000 Poisson resamplings of UNC 1 PM.

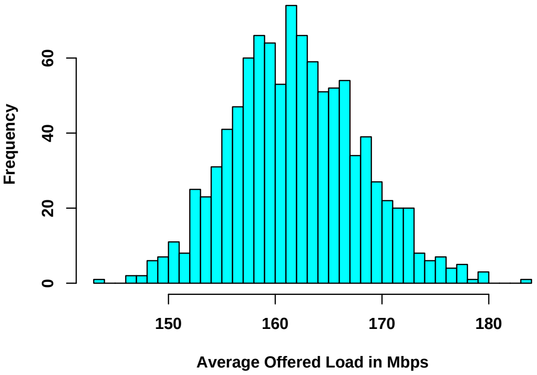


Figure 7.6: Histogram of the average offered loads in 1,000 Poisson resamplings of UNC 1 PM.

usually represent only a small fraction of s, the total size of the ADUs in $\mathcal{T}_c$ divided by d is also a close approximation of l. We use this approximation to examine the average loads offered by a large number of Poisson resamplings, considering the offered load of a resampling $\mathcal{T}_c'$ equal to the total size s' of its ADUs divided by its duration d'. It is also important to note that the traces we consider are bidirectional, and they do not necessarily create the same average load in both directions. The analysis in the rest of this section will focus only on one direction of the trace, the *target* direction, whose average load is given the total size of the ADUs flowing in that direction divided by the duration of the trace. More formally, the total size of the ADUs in the target direction is equal to

$$s' = \sum_{i \in C_{init}} \sum_{j=1}^{n_a^i} a_j^i + \sum_{i \in C_{acc}} \sum_{j=1}^{n_b^i} b_j^i, \tag{7.1}$$

where C_{init} is the set of connection vectors in $\mathcal{T}_c'$ initiated in the target direction, C_{acc} is the rest of the connection in $\mathcal{T}_c'$ (the connections accepted rather than initiated in the target direction), n_a^i and n_b^i are the numbers of a-type and b-type ADUs of i-th connection vector respectively, and a_j^i and b_j^i are the sizes of the j-th a-type and b-type ADU of the i-th connection vector respectively. Computing offered load as s'/d' is only a convenient (and reasonable) approximation of the load generated by replaying $\mathcal{T}_c'$. First, s' is an underestimation, since it does not take into account the total size of packet headers (only ADUs), and the retransmissions in the replay. Second, the duration of the replay of the connection vectors in $\mathcal{T}_c'$ will be somewhat above d'. d' only considers the period in which connections are started, but some of them will terminated after the last connection is started. An obvious example is the last connection. As we will demonstrate using experiments, the inaccuracy of s'/d' is very small, so it provides a good foundation for understanding load scaling. This calculation is obviously much more convenient than

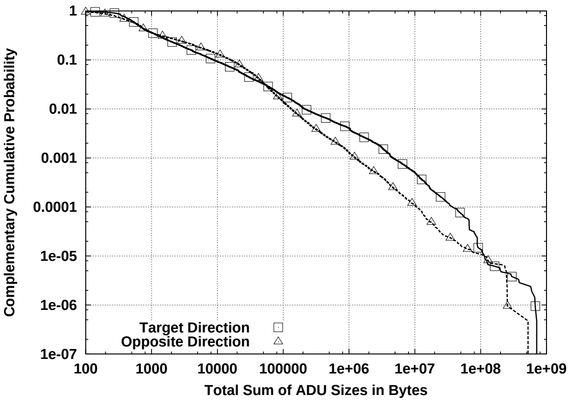


Figure 7.7: Tails of the distributions of connection sizes for UNC 1 PM.

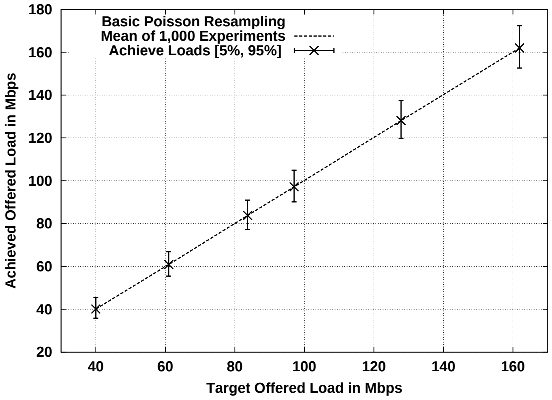


Figure 7.8: Analysis of the accuracy of connection-driven Poisson Resampling from 6,000 resamplings of UNC 1 PM (1,000 for each target offered load).

replaying thousands of resamplings in the testbed network.

Figure 7.5 shows a scatterplot of the results of 1,000 resamplings of UNC 1 PM. The duration of the resamplings and their mean rate of connection inter-arrival were equal to the ones in UNC 1 PM. For each resampling, the total number of connections is shown on the x-axis, while the resulting offered load s'/d' is shown on the y-axis. This plot demonstrates that basic Poisson Resampling results in traces with very small variability in the number of connection vectors, between 1,409,727 and 1,417,664 (the standard deviation σ was equal to 1,191.71). On the contrary, the range of offered loads is very wide, between 143.55 and 183.44 Mbps ($\sigma = 6.01$ Mbps), centered around the offered load of $\mathcal{T}_c$, 161.89 Mbps. The distribution of offered loads and its spread is further illustrated by the histogram in Figure 7.6.

The wide range of offered loads that can result from Poisson Resampling is due to the heavy-tailed nature of the distribution of the total number of bytes contributed by each connection vector. The tail of this distribution for the UNC 1 PM trace is shown in Figure 7.7. The values in the plot correspond to the target direction, *i.e.*, $\sum_{j=1}^{n_a^i} a_j$ for each connection in C_{init} and $\sum_{j=1}^{n_b^i} b_j$ for each connection in C_{acc}. The tails show non-negligible probabilities for very large sizes, and a linear decay of the probability over six orders of magnitude in the log-log CCDF. As a consequence, the contribution to the offered load made by each connection is highly variable and thus the number of connections in a trace is a poor predictor of its offered load. This makes basic Poisson Resampling inadequate for controlling load. Its only parameter is the mean inter-arrival rate of connections. This rate controls the same number of connections in the resampling, but not the total size of these connections, which varies greatly due to the heavy-tailed connection sizes. Figure 7.8 further illustrates this point using six sets of 1,000 trace resamplings, each set with a different target offered load. The plot shows a cross marking the mean of the load achieved by each of the sets of 1,000 experiments. The variability in the offered loads is illustrated using error bars for each set of experiments. The lower and upper endpoints of the error bars correspond to the 5th and 95th percentiles of the loads offered by each set of trace resamplings. Each set of trace resamplings had a fixed mean inter-arrival rate μ'. Under the assumption that the mean offered load l is proportional to the number of connections n, we would expect the load to be inversely proportional to the mean arrival rate μ, since $\mu = d/n$. Therefore, if the resamplings have the same duration d, we would expect to achieve an offered load of

$$l' = \frac{l\mu'}{\mu} \tag{7.2}$$

in each resampling. This expected value is shown in the x-axis as "target offered load". The mean of the loads offered by each set of resamplings is indeed equal to the expected (target) value. However, the error bars show a wide range of offered loads for the same μ', which is undesirable for a load scaling technique. For example, the width of the range of loads offered by the resamplings with the highest target load was 20 Mbps. Differences of this magnitude between resamplings can have a dramatic effect on the experiments, completely obscuring differences among competing mechanisms. The difficulty of precisely controlling the load using only the number of connections as a parameter motivated our refinement of Poisson Resampling to achieve far more predictable offered loads.

7.1.2 Byte-Driven Poisson Resampling

As the previous section demonstrated, the number of connection vectors in a trace is a poor predictor of the mean offered load achieved during the replay of a resampled trace $\mathcal{T}_c'$. Therefore, controlling the number of connections in a resampling does not provide a good way of achieving a target offered load, and an alternative method is needed. In the idealized model of offered load in the previous section, the offered load l was said to be exactly equal to s/d. If so, we need to control the total number of bytes in the resampled trace $\mathcal{T}_c'$ to precisely match a target offered load. In *Byte-driven Poisson Resampling*, the mean inter-arrival rate of $\mathcal{T}_c'$ is not computed a priori using Equation 7.2. Instead, the target load l' is used to compute a target size $s' = l'd'$ for the payloads in the scaled direction.

Byte-driven Poisson Resampling has two steps:

1. We construct a set of connection vectors (without arrival times) by iteratively resampling the

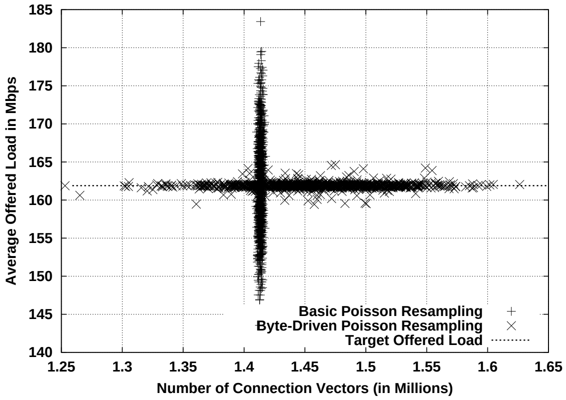


Figure 7.9: Comparison of average offered load *vs.* number of connections for 1,000 connection-driven Poisson resamplings and 1,000 byte-driven Poisson resamplings of UNC 1 PM.

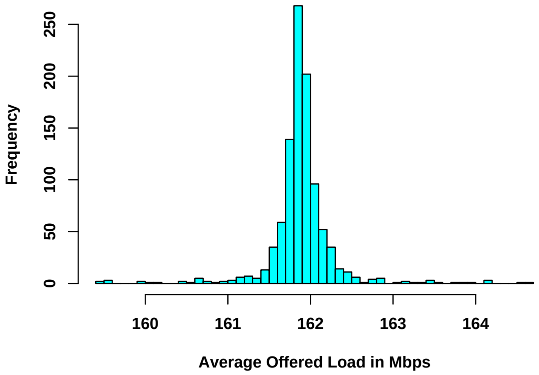


Figure 7.10: Histogram of the average offered loads in 1,000 byte-driven Poisson resamplings of UNC 1 PM.

connection vectors in $\mathcal{T}_c$ until the total payload size of the chosen connection vectors, computed using Equation 7.1, reaches s'.

2. We assign start times to the connection vectors in the resampling using the technique described in the previous section. The mean of the exponential distribution from which inter-arrival times are sampled is d'/n', where d' is the desired duration of $\mathcal{T}_c'$, and n' is the number of connection vectors in the resampling.

Using this technique, and under the assumption that $l = s/d$, the load offered by the resulting $\mathcal{T}_c'$ should be very close to the target load[5]. Figure 7.9 demonstrates that this is the case by comparing the offered loads of 1,000 simulated trace resamplings constructed using the technique in section 7.1.1 and another 1,000 resamplings using the byte-driven technique. The target load of the byte-driven resamplings was 161.89 Mbps, which was the average load in the original UNC 1 PM trace. The range of achieved offered loads is far narrower for the second technique, thanks to the variable number of connection vectors that are assigned to each $\mathcal{T}_c'$. The histogram in Figure 7.10 shows that the vast majority of the resamplings are very close to the target load ($\sigma = 0.41$ Mbps).

Figure 7.11 summarizes the results of 4 sets of 1,000 byte-drive Poisson resamplings. The plot uses the same type of visualization found in Figure 7.8. The error bars, barely visible in this case, illustrate the accurate load scaling that can be achieved with Byte-driven Poisson Resampling.

[5] Note that the duration of $\mathcal{T}_c'$ comes from random samples of an exponential distribution, so it can be slightly lower or higher that the intended d'. Given the light tail of the exponential distribution and the large number of samples, this deviation is necessarily very small.

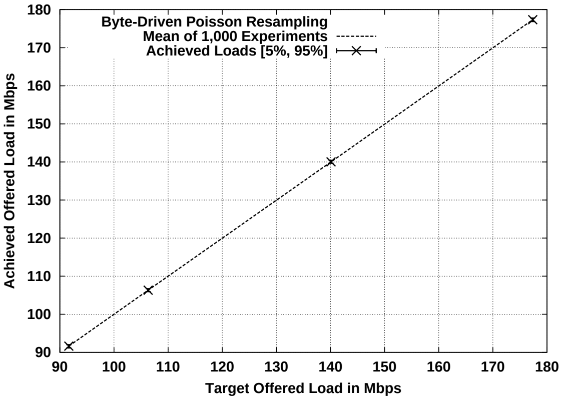


Figure 7.11: Analysis of the accuracy of byte-driven Poisson Resampling from 4,000 resamplings of UNC 1 PM (1,000 for each target offered load).

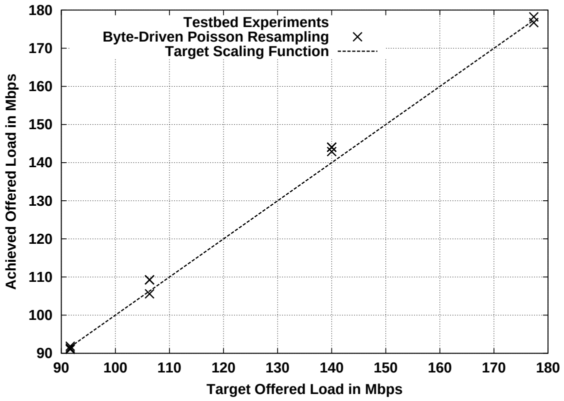


Figure 7.12: Analysis of the accuracy of byte-driven Poisson Resampling using source-level traces replay: replays of three separate resamplings of UNC 1 PM for each target offered load, illustrating the scaling down of load from the original 177.36 Mbps.

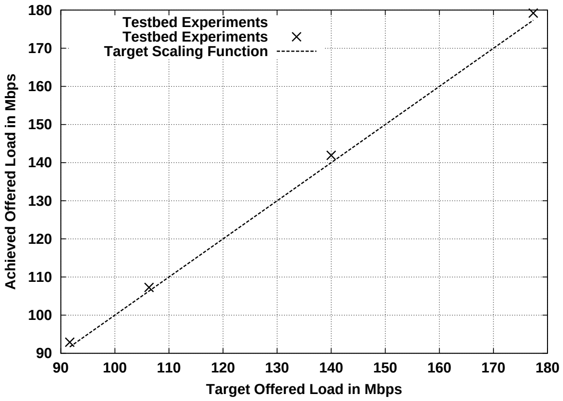


Figure 7.13: Analysis of the accuracy of byte-driven Poisson Resampling using testbed experiments: replay of one resampling of UNC 1 AM for each target offered load, illustrating the scaling up of load from the original 91.65 Mbps.

The previous analysis demonstrated that accurate load scaling requires control of the total number of bytes in $\mathcal{T}'_c$ rather than of the total number of connections. This demonstration was based on the computation of the offered load using equation 7.1. It is important to verify that the actual load generated during a testbed replay of $\mathcal{T}'_c$ is similar to the computed load. To show that this is indeed the case, we replayed a number of resampled traces with four different target loads. Each resampled trace was then constructed using byte-driven Poisson Resampling, with a duration of 1 hour. To eliminate startup and shutdown effects, we only considered the middle 40 minutes for computing the achieved load. Figure 7.12 summarizes the results of the experiments for the resamplings of the UNC 1 PM trace. Each point corresponds to a separate replay experiment, showing the target load on the x-axis and the achieved load on the y-axis. We ran three experiments for each target load, and the results show a good approximation of the intended scaling line. Several experiments achieved loads a few Megabytes above the target. In general, we expected the experiments to have slightly higher achieved loads, since the scaling method focuses on the offered payload, ignoring packetization overhead (*i.e.*, extra load from bytes in the packet headers). A more precise tuning of the offered load would take packetization into account, perhaps using a fixed constant to decrease the target payload, or by studying the total size (including headers) of each connection, as replayed in the same testbed environment. In any case, and given the above good results, it seems reasonable to ignore these further refinements.

The results in Figure 7.12 provide examples of *scaling down* the load of a trace, since the original load of UNC 1 PM was 177.36 Mbps, and 9 of the 12 experiments had target offered loads below this value. Scaling down the load of a trace using byte-driven resampling simply requires to choose a target load l' below the original load l, which in turns means that the s' of the resampling will be below the original s. These results confirm the close approximation of the target loads in the testbed experiments, where offered load is measured from real TCP segments (rather than computed using Equation 7.1). The plot shows for example that the three resamplings with target load 177.36 Mbps achieved loads of 176.72, 178.23 and 182.45 respectively. The impact of the TCP headers, retransmission and the slightly underestimated duration mentioned in the previous section is therefore very small. The results in Figure 7.13 provide an example of *scaling up* the load of a trace, since they correspond to byte-driven Poisson resamplings of UNC 1 AM, which had an original load of 91.65 Mbps. The resulting loads also approximate the intended targets very closely.

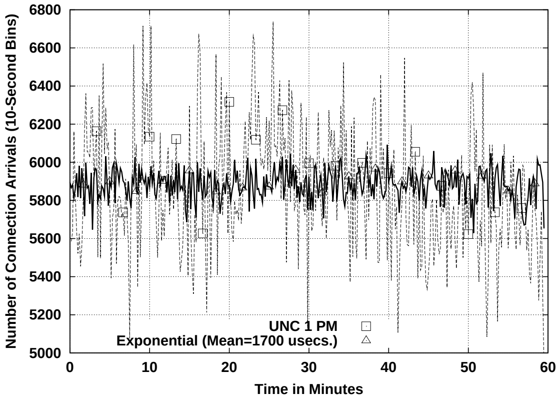


Figure 7.14: Connection arrival time series for UNC 1 PM (dashed line) and a Poisson arrival process with the same mean (solid line).

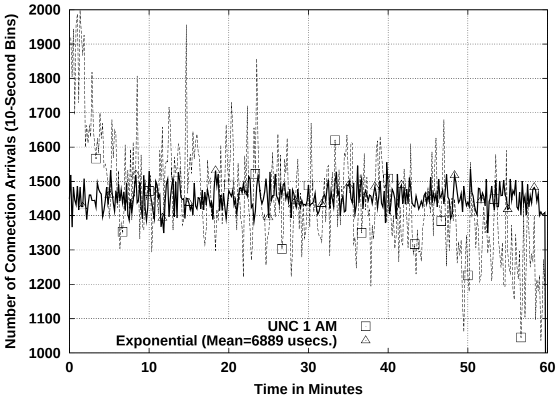


Figure 7.15: Connection arrival time series for UNC 1 AM and a Poisson arrivals process with the same mean.

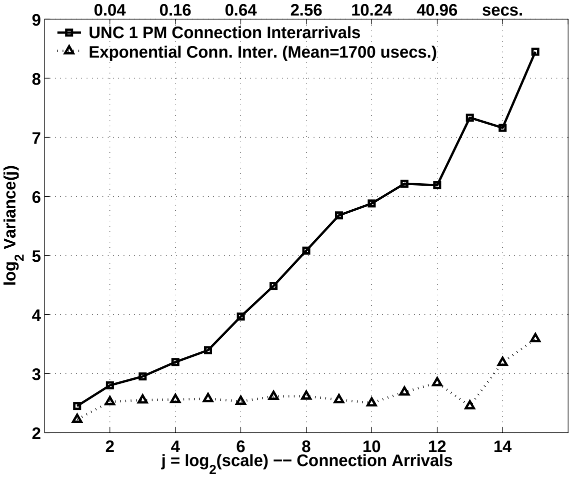


Figure 7.16: Wavelet spectra of the connection arrival time series for UNC 1 PM and a Poisson arrival process with the same mean.

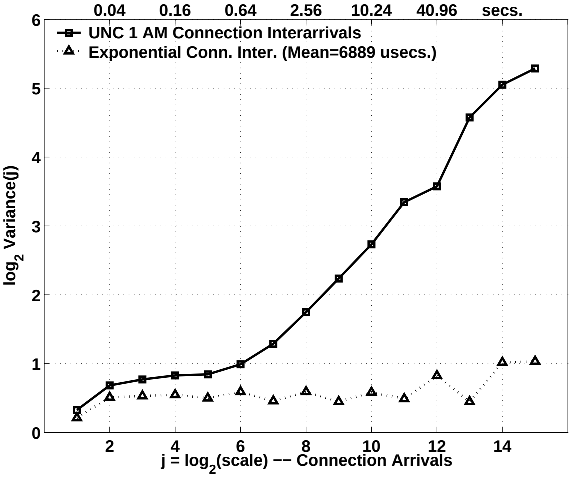


Figure 7.17: Wavelet spectra of the connection arrival time series for UNC 1 AM and a Poisson arrival process with the same mean.

Trace	Estimated Parameters
UNC 1 PM Conn. Arrivals	H=0.685041 C.I.=[0.646250, 0.723831]
Poisson $\mu = 1,698$ μsecs.	H=0.506069 C.I.=[0.467279, 0.544860]
UNC 1 AM Conn. Arrival	H=0.756533 C.I.=[0.717743, 0.795324]
Poisson $\mu = 6,889$ μsecs.	H=0.502217 C.I.=[0.463427, 0.541008]

Table 7.1: Estimated Hurst parameters and their confidence intervals for the connection arrival time series of UNC 1 PM and UNC 1 AM, and their Poisson arrival fits.

7.2 Block Resampling

The basic assumption of Poisson Resampling is that connection inter-arrivals are independent and identically distributed according to an exponential distribution, which results in a Poisson arrival process. While the choice of exponential inter-arrivals is reasonable given the measurement data presented in Figures 7.1 to 7.4, the arrival process may not necessarily be independent. On the one hand, we can argue that common application protocols make use of more than one connection, creating correlations among some start times. For example, web browsers often open several connections for downloading a web page. On the other hand, we focus on traces of highly aggregated traffic, where a large number of hosts start hundreds or even thousands of connections every second. The high aggregation could diminish or even eliminate completely any correlation structure in the connection arrival processes.

The analysis of our traces reveals non-negligible correlations in the connection arrival process. Figures 7.14 and 7.16 examine the arrival process for the UNC 1 PM trace. Using a time series of 10-millisecond bin counts, Figure 7.14 compares the burstiness of the original arrival process (dashed line) and that of a Poisson arrival process with the same mean inter-arrival time (solid line). The original arrival process was far more variable. Its standard deviation was equal to 346.07, while the one for the Poisson process was equal to 79.21. In order to further study the connection arrival process across a range of time-scales, we rely on wavelet analysis. Figure 7.16 shows the wavelet spectra of the original connection arrivals and a Poisson process with the same mean inter-arrival time. The Poisson process exhibits the expected flat spectrum of short-range dependent processes [HVA02]. On the contrary, the spectrum for the original connection arrivals follows a line with a substantial positive slope, which is characteristic of long-range dependent processes. The results of the wavelet-based estimation [AV98] of the Hurst parameters of these processes are given in table 7.1. The Poisson process has a Hurst parameter very close to the expected 0.5, while the original arrival process has a Hurst parameter of 0.685. This is consistent with moderate long-range dependence. For comparison, typical estimates of the Hurst parameter for packet and byte arrival processes are between 0.75 and 0.95, *i.e.*, typical packet and byte arrival processes exhibit significantly stronger long-range dependence than this connection arrival process.

We performed a similar analysis for the UNC 1 AM trace, and the results are shown in Figures 7.15 and 7.17. As in the previous case, the time series plot shows a connection arrival process that is significantly more bursty than that of a Poisson process with the same mean. Note however than in this case there is some degree of non-stationarity. We observe a significantly larger number of connections started in the first 5 minutes, and a significantly lower number started in the last 10 minutes. In this case we compute the mean inter-arrival rate required to construct the Poisson arrivals using the middle 40 minutes of the trace. We therefore handle the effect of trace boundaries by ignoring the first and the last few minutes of the arrival process. The wavelet spectra for these middle 40 minutes and a Poisson process with the same mean arrival rate are shown in Figure 7.17. As in the UNC 1 PM case, the original connection arrival process exhibits clear long-range dependence. The estimated Hurst parameter in Table 7.1 reveals a somewhat stronger long-range dependence for the UNC 1 AM trace (0.757 *vs.* 0.685).

In summary, the connection arrival processes we have examined are consistent with significant long-range dependence. Therefore, it is desirable to develop the resampling and load scaling methods that can reproduce this structure, to cover experiments where the manner in which connections arrive is relevant for the network phenomenon studied using synthetic traffic. One example of this type of scenario is the evaluation of a router mechanism where the arrival of new connections creates new state in the router. For such a mechanism, a more bursty arrival process creates a more stringent workload, just like burstier traffic was shown by [BC98] to be more demanding on web server performance.

Poisson Resampling cannot reproduce this observed long-range dependence in the connection arrival process since its inter-arrivals times come from independently sampling an exponential distribution. For this reason, we propose a second resampling technique that can reproduce the long range dependence in the connection arrival process. The starting point is the intuition that dependencies between connection start times are far more likely to occur within relatively small periods. For example, web browsing results in new connections started according to the sequence of web page downloads and the way the browser opens new connections to the servers in which these pages are found. This results in brief bursts of connections whose start times are correlated. We use this intuition to develop a resampling method wherein the resampled objects are not individual connections, but groups of connections started during the same period, which we call *blocks*. The key idea of our *Block Resampling* method is that sampling blocks of connections rather than individual connections preserves the relative offsets of connection start times within blocks, and therefore the dependency structure[6] Our method is derived from the Moving Block Bootstrap method [ET93].

Block Resampling proceeds in the following manner: Given a trace $\mathcal{T}_c$, we divide it in blocks of duration β, so that the first block B_1 groups together connections started in the interval $[0, \beta)$, the second block B_2 groups together connections started in the interval $[\beta, 2\beta)$, and so on. The block resampled trace $\mathcal{T}_c'$ is obtained by concatenating randomly sampled blocks, and adjusting the start time

[6] We thank Peter Hall for suggesting the use of block bootstrapping in the context of the a-b-t model. The theoretical aspect of this idea are explored in [HNHC02], while this chapter focuses on its use to preserve the long-range dependence in connection arrivals and develops thinning and thickening methods to scale offered load in block-resampled traces.

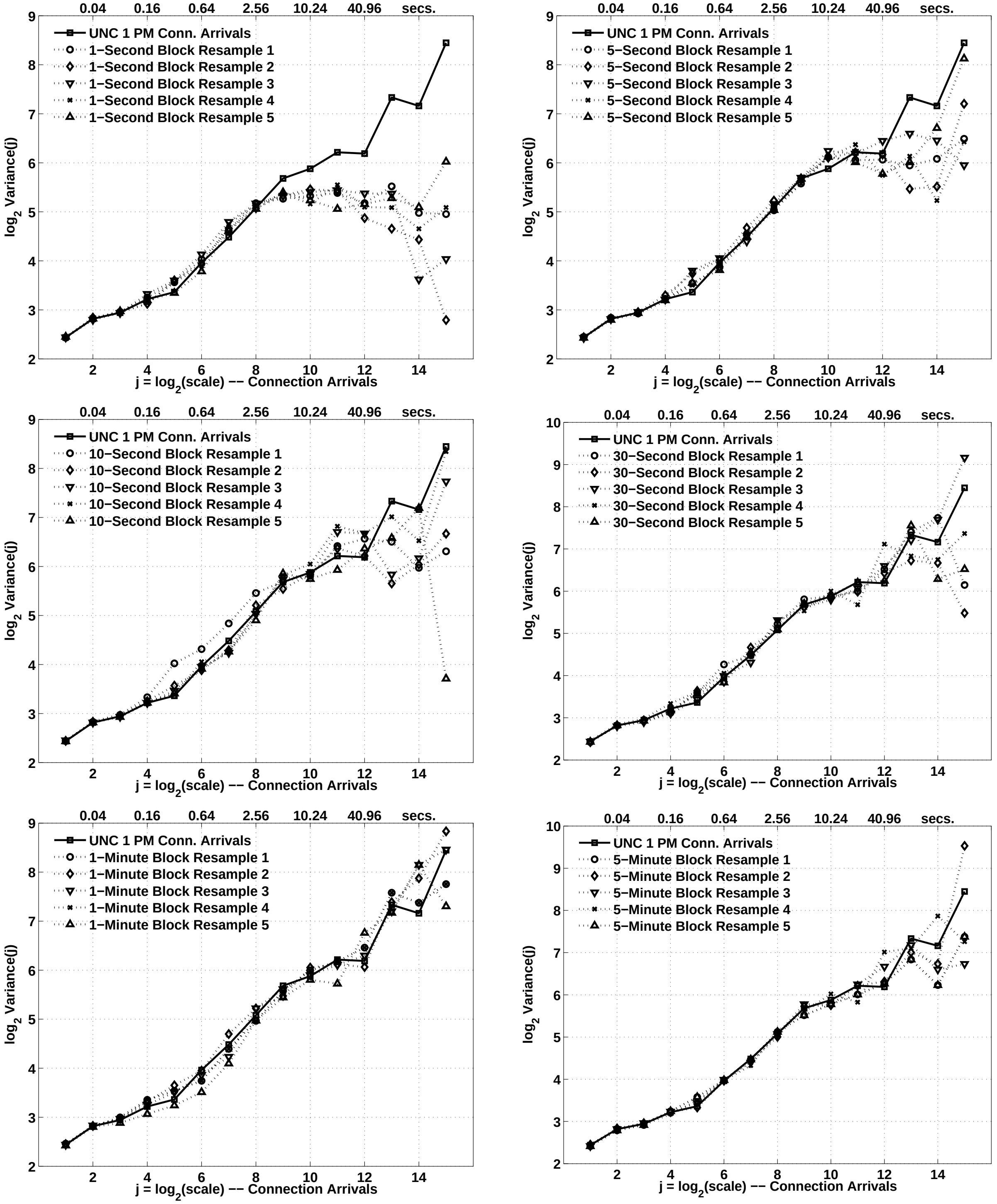


Figure 7.18: Block resamplings of UNC 1 PM: impact of different block lengths on the wavelet spectrum of the connection arrival time series.

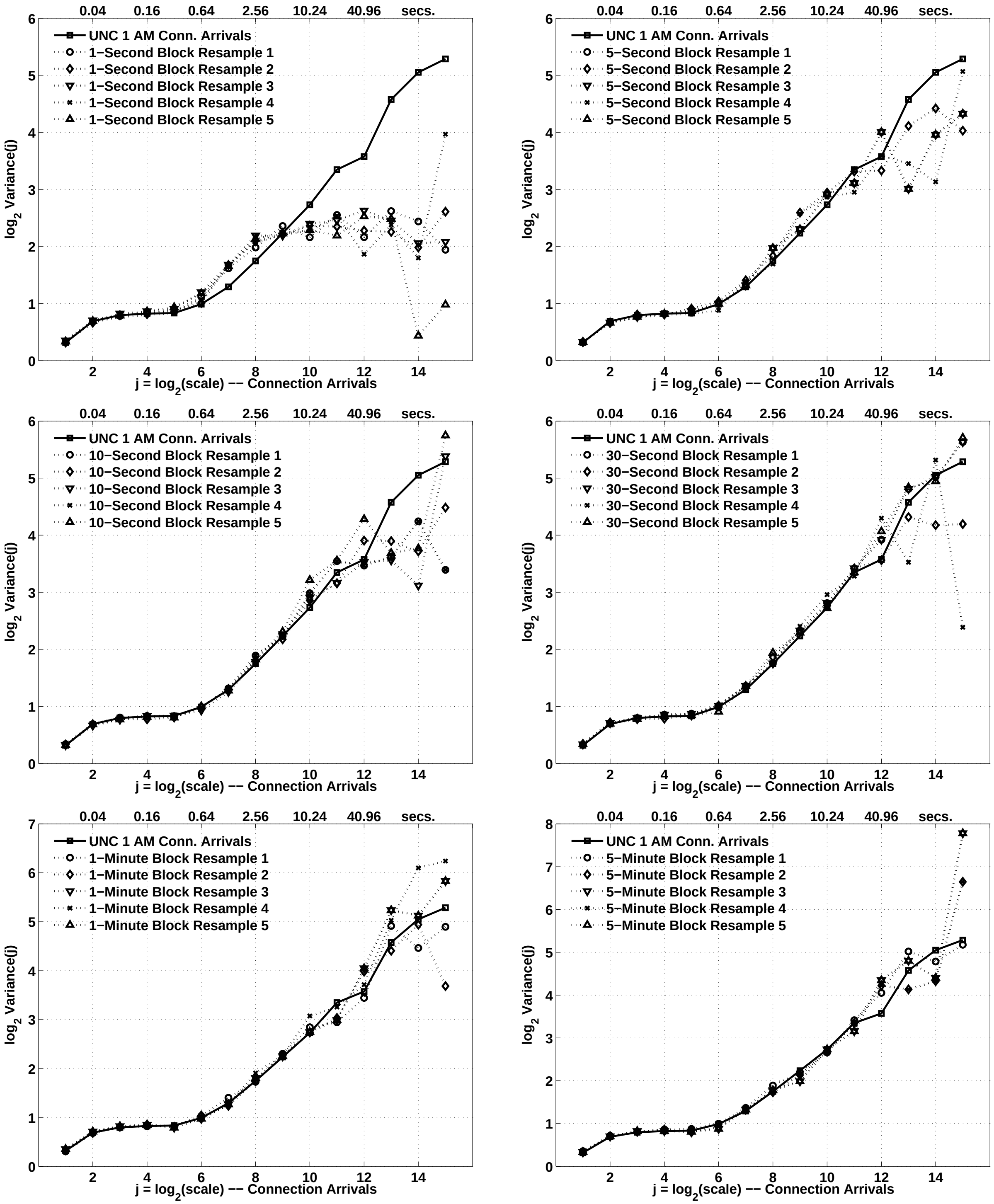


Figure 7.19: Block resamplings of UNC 1 AM: impact of different block lengths on the wavelet spectrum of the connection arrival time series.

of connections in each block by the time offset of the new location of this block. For example, if the random resampling puts block B_2 as the first block of $\mathcal{T}_c'$, the start times of the i-th connection vector in this block is set to $T_i - \beta$. Similarly, if B_2 is placed in the fourth location of $\mathcal{T}_c'$, the start times of the i-th connection in this block are set to $T_i + 2\beta$. More formally, when the j-th block B_j in the original trace becomes the k-th block B_k in the block resampling, the start time T_i of the i-th connection vector in B_j is set to

$$T_i' = T_i + (k - j)\beta.$$

Block Resampling chooses blocks for $\mathcal{T}_c'$ with replacement, making it possible to create new traces that are longer than the original $\mathcal{T}_c$ from which the blocks are obtained.

As pointed out by Efron and Tibshirani [ET93], choosing the block duration β can be a difficult problem. In our case, we found a clear trade-off between block duration and how well long-range dependence was preserved in the resampled trace. The shorter the block duration, the larger the number of distinct trace resamplings that can be performed from the same trace $\mathcal{T}_c$. This number is equal to $(d/\beta)!$ for resampled traces with the same duration d of the original trace. However, if the duration of the blocks is too small, the process of connection arrivals in the resampled trace exhibits a dependency structure that does not resemble the one in the original trace.

Figure 7.18 explores the impact of block duration on the long-range dependence of the connection arrival process in the resampled trace. The top left plot shows the wavelet spectra of the connection arrivals for UNC 1 PM and for 5 block resamplings where the block duration was 1 second. There is a clear and consistent flat region after octave 8, which shows that blocks of 1 second are too short to preserve the long-range dependence of the original connection arrival process. As the block duration is increased in subsequent plots, we observe an increasingly better match between the arrivals in the block resamplings and the arrivals in the original trace. Blocks with a duration of 30 seconds or 1 minute provide the best trade off between blocks that are large enough to ensure realistic long-range dependence in the connection arrival process, and blocks that are short enough to provide a large number of distinct resamplings. The same sensitivity analysis was performed for the UNC 1 AM trace and the results are shown in Figure 7.19. Block durations of 30 seconds or 1 minute are also shown to perform well.

As discussed earlier in this chapter, an important goal of trace resampling is the ability to preserve the target load of the original trace and to scale it up and down according to the needs of the experimenter. The analysis of a large set of Poisson resamplings revealed that offered load and number of connections are only loosely correlated. This motivated the use of a byte-driven version of Poisson Resampling

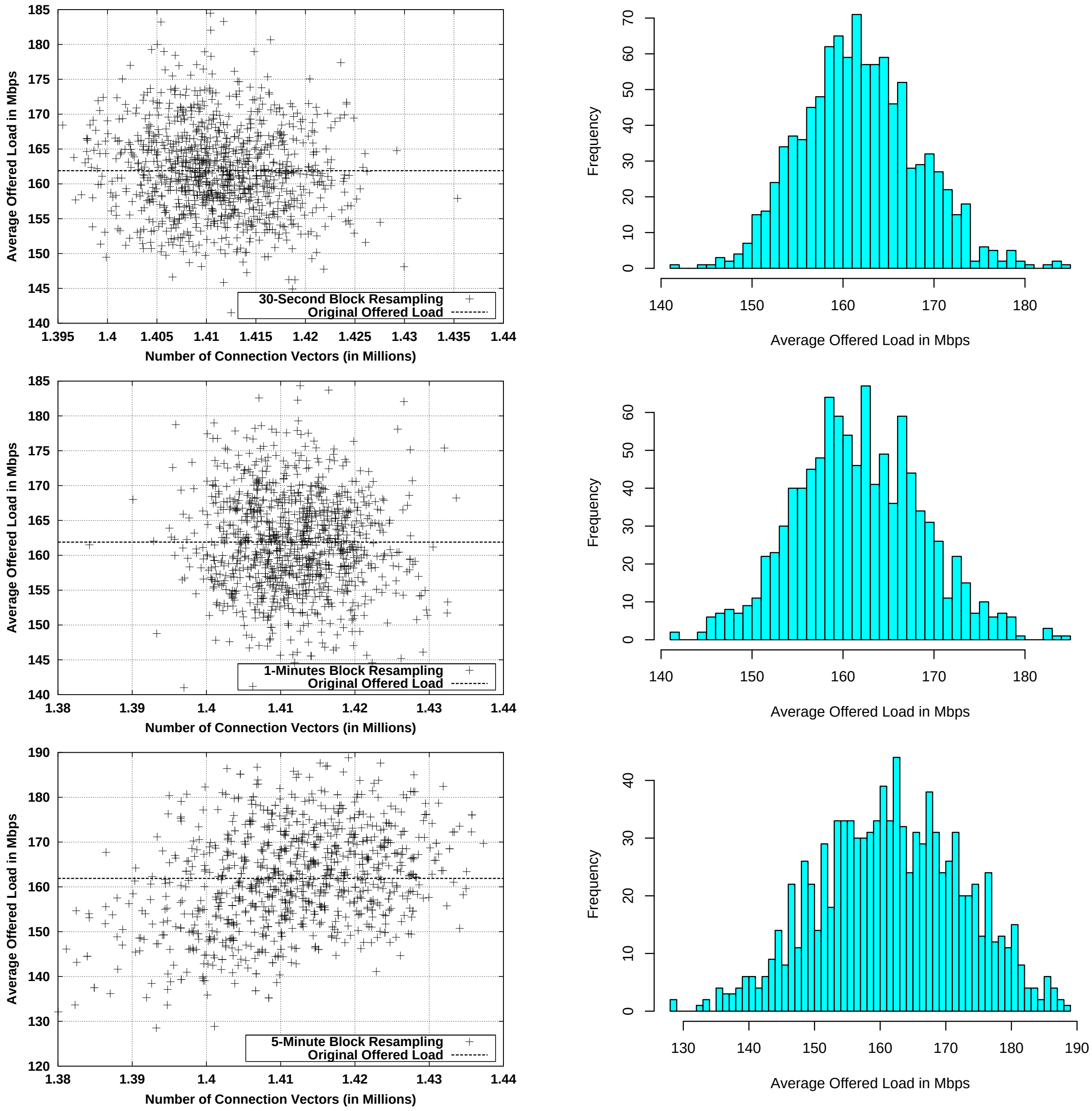


Figure 7.20: Block resamplings of UNC 1 PM: average offered load *vs.* number of connection vectors (left) and corresponding histograms of average offered loads (right) in 3,000 resamplings.

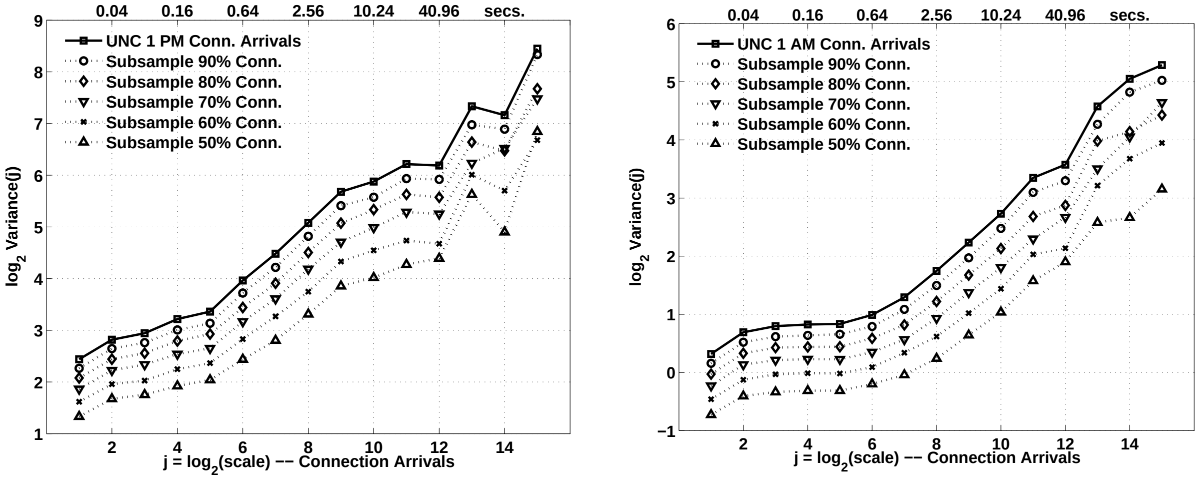


Figure 7.21: Wavelet spectra of several random subsamplings of the connection vectors in UNC 1 PM (left) and 1 AM (right)

which could achieve a very precise scaling of the load offered by the resampled trace. In the case of Block Resampling, the question is whether the averaging effect of grouping connections into blocks significantly diminishes the variability observed for the basic version of Poisson Resampling. We study this question by examining the offered load found in a large collection of block resampled traces. If the blocks had roughly uniform offered load, we would expect to generate similar offered load with each resampled trace. The results in Figure 7.20 do not confirm this expectation. The top row presents the analysis of 1,000 trace resamplings constructed by resampling UNC 1 PM using 30-second blocks. The average offered load was derived from the total payload computed using Equation 7.1. As shown in the scatterplot, the number of connections stayed within a narrow range, but the offered loads were far more variable. The histogram on the right further characterizes the distribution of offered loads in these trace resamplings. The use of blocks does not appear to have any benefit in terms of a more predictable load. This is not surprising given the known burstiness of the packet and byte arrival processes at many time-scales. If blocks were effective at smoothing out these processes, we would find little long-range dependence. This situation does not change for longer block durations, as shown in the middle and lower rows of Figure 7.20 for blocks of 1 and 5 minutes respectively. It is interesting to note the wider y-axis and range of the histogram for the 5-minutes blocks, which suggest even higher variability for this longer block duration.

The Block Resampling method described so far makes it possible to construct a resampled $\mathcal{T}_c'$ of arbitrary duration but it cannot be used to adjust its offered load. In order to perform this task, we can rely on *thinning*, when the offered load of $\mathcal{T}_c$ is above our intended offered load, and on *thickening*, when

Trace	Estimated Parameters
UNC 1 PM Conn. Arrivals	H=0.727540 C.I.=[0.701687, 0.753393]
Subsample 90% Conn.	H=0.724175 C.I.=[0.698322, 0.750028]
Subsample 80% Conn.	H=0.724046 C.I.=[0.698193, 0.749899]
Subsample 70% Conn.	H=0.718502 C.I.=[0.692649, 0.744354]
Subsample 60% Conn.	H=0.701378 C.I.=[0.675525, 0.727230]
Subsample 50% Conn.	H=0.701020 C.I.=[0.675167, 0.726872]
UNC 1 AM Conn. Arrivals	H=0.746591 C.I.=[0.720738, 0.772444]
Subsample 90% Conn.	H=0.738659 C.I.=[0.712806, 0.764512]
Subsample 80% Conn.	H=0.725030 C.I.=[0.699177, 0.750882]
Subsample 70% Conn.	H=0.715679 C.I.=[0.689827, 0.741532]
Subsample 60% Conn.	H=0.696723 C.I.=[0.670870, 0.722576]
Subsample 50% Conn.	H=0.691139 C.I.=[0.665287, 0.716992]

Table 7.2: Estimated Hurst parameters and their confidence intervals for five subsamplings obtained from the connection arrival time series of UNC 1 PM and UNC 1 AM

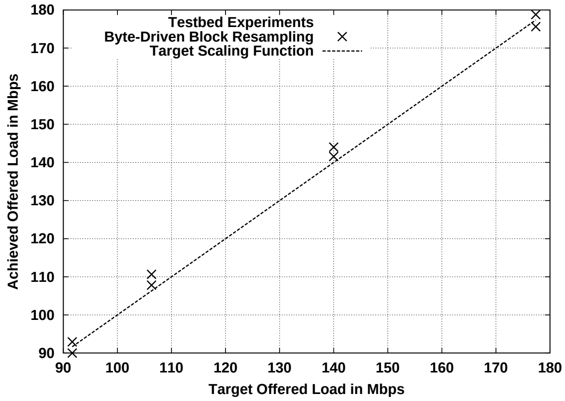


Figure 7.22: Analysis of the accuracy of byte-driven Block Resampling using source-level trace replay: replays of two separate resamplings of UNC 1 PM for each target offered load, illustrating the scaling down of load from the original 177.36 Mbps.

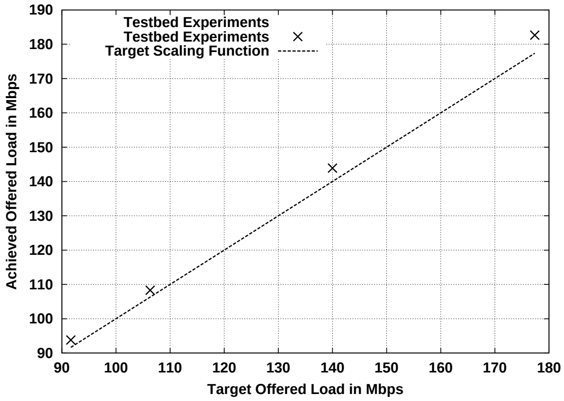


Figure 7.23: Analysis of the accuracy of byte-driven Block Resampling using source-level trace replay: replay of one resampling of UNC 1 AM for each target offered load, illustrating the scaling up of load from the original 91.65 Mbps.

the offered load of $\mathcal{T}_c$ is below our intended offered load. Block thinning involves randomly removing connections from $\mathcal{T}_c'$. Theoretical work by Hohn and Veitch [HV03]has shown that the thinning of a long-range dependent process does not change its long-range dependence structure. Our own experimentation confirms this result. Figure 7.21 shows the wavelet spectra of thinned versions of the connection arrivals in the UNC 1 PM trace (left) and the UNC 1 AM trace (right). The overall energy level decreases as the fraction of connections removed from each block increases. However, the spectra maintain their shapes, which demonstrates that the degree of the long-range dependence remains unchanged. The estimated Hurst parameters for these two traces is presented in Table 7.2. The values reveal only a moderate decrease in the Hurst parameter even when half of the connections are dropped.

Block thickening consists of combining more than one block in each of the disjoint intervals of $\mathcal{T}_c'$, *i.e.*, to "fusion" one or more blocks from $\mathcal{T}_c$ to form a single block in $\mathcal{T}_c'$. This makes the offered load a multiple of the original load. For example, to double the load, the connection vectors of two randomly chosen blocks will be placed in the first interval, those from another pair of randomly chosen blocks will be placed in the second interval, and so on. The new start times of the connection vectors in the resampled trace are computed using Equation 7.2, but being careful to use the right j for each connection vector.

To achieve offered loads that are not a multiple of the original load, we can combine basic thickening and thinning using a two-step process. The first step is to create a preliminary version of $\mathcal{T}_c'$ by combining as many blocks as possible without exceeding the target load. The second step is to "complete" this trace by combining it with another block-resampled trace that has been thinned in such a manner that the combined load of the two resampled traces matches the intended load. For example, in order to create a $\mathcal{T}_c'$ with 2.5 times the load of $\mathcal{T}_c$, a first thickened trace $\mathcal{T}_c^{tk}$ is created by combining two blocks in each position. This trace is then combined with second trace $\mathcal{T}_c^{tn}$ that has been thinned to half of the offered load of $\mathcal{T}_c$. From our analysis in Figure 7.20, we can see that $\mathcal{T}_c^{tk}$ is not necessarily equal to twice the offered load of $\mathcal{T}_c$. For this reason $\mathcal{T}_c^{tn}$ is actually thinned to exactly the offered load needed to complement $\mathcal{T}_c^{tk}$, and not just to half of the original offered load This careful thinning makes the scaling match the intended load in a highly precise manner. We can therefore achieve any intended load with the Block Resampling method, so it is as flexible as Poisson Resampling. In accordance with our earlier analysis, accurate thinning cannot rely on any correlation between the number of connections and the offered load, so it must be driven by Equation 7.1, just like byte-driven Poisson Resampling. Therefore, our final resampling technique is Byte-driven Block Resampling.

Figures 7.22 and 7.23 show the result of several testbed experiments where Byte-driven Block Resampling is used to create new traces. The results demonstrate that traces resampled using this method achieve a very good approximation of the target offered loads. As in the case of Byte-driven Poisson Resampling, the achieved loads are slightly higher than target ones due to the packetization overhead, which is not taken into account in the resampling.

One interesting question is whether the effort to preserve the scaling of the connection arrival process has any effect on the generated traffic aggregate. To understand this question, we can compare the process of packet (or byte) arrivals from block resamplings and from Poisson Resampling, since the former fully preserves connection arrival long-range dependence and the latter fully eliminates it. Figure

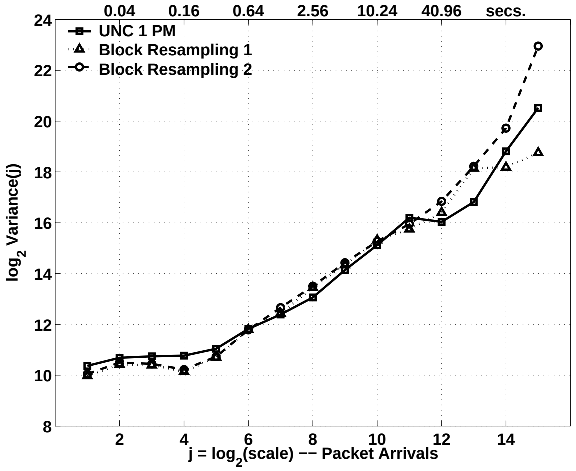


Figure 7.24: Wavelet spectra of the packet arrival time series for UNC 1 PM and the source-level trace replays of two block resamplings of this trace.

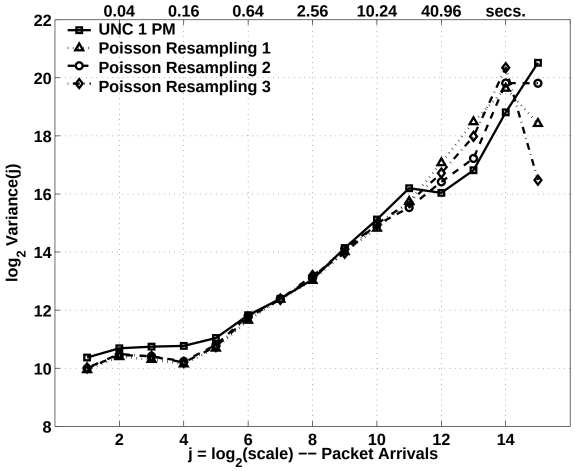


Figure 7.25: Wavelet spectra of the packet arrival time series for UNC 1 PM and the source-level trace replays of three Poisson resamplings of this trace.

7.24 shows the wavelet spectra of the packet arrivals in UNC 1 PM and those in two testbed experiments where byte-driven block resamplings of UNC 1 PM were replayed. Figure 7.25 shows the same wavelet spectrum of the packet arrivals in UNC 1 PM, and also the spectra from three testbed experiments where byte-driven Poisson resamplings of UNC 1 PM were replayed. Both resampling methods achieve equally good approximations of the packet scaling found in the original trace. In other words, according to this type of analysis, the simpler Poisson Resampling method performs as well as the more elaborate Block Resampling method. This is a confirmation, using a closed-loop traffic generation approach, of the results by Hohn *et al.* in [HVA02], which were obtained using (open-loop) semi-experiments. This is not to say that long-range dependence in the arrival of connections (*e.g.*, arrival of flow state or cache misses to a router) can be safely ignored, since other metrics and experimental results may be more sensitive to this characteristic of the synthetic traffic.

7.3 Summary

Our basic traffic generation method, source-level trace replay, results in highly realistic synthetic traffic. This method is however inflexible, in the sense that the same connection vectors are started at the same relative times in every replay. In this chapter, we proposed two methods for resampling an original trace of connection vectors, to create a new trace with similar statistical characteristics. This similarity is defined in terms of source-level behavior and network-level parameters, so the resampling methods also modify connection vector start times. Our first resampling method is Poisson Resampling, which chooses connections vectors at random and assigns them exponentially distributed inter-arrival times. Our measurement results demonstrated that this choice of the inter-arrival distribution is appropriate, in the sense that the marginal distribution of the connection inter-arrival in every trace we examined is remarkably consistent with the exponential distribution. Our second resampling method is Block Resampling, which chooses blocks of connection vectors at random. Unlike Poisson Resampling, Block Resampling preserves the dependency structure of the original connection arrival process. This makes it possible to reproduce the moderate long-range dependence that we observe in the connection arrivals of our traces.

Besides presenting two resampling methods, we also studied how to control the offered load by a resampled trace. Firstly, we demonstrated that the number of connections and the average offered load are not strongly correlated. This means that controlling the number of connections in the resamplings does not provide a good way of creating resampled traces with a specific target offered load. This is a common requirement when a set of experiments covers a range of offered loads in an empirical study. In order to address this difficulty, we propose to drive the resampling by a target total size of the ADUs in the resampling rather than by a target number of connections. We used this approach to develop byte-driven versions of Poisson Resampling and Block Resampling, which are shown to result in highly predictable offered loads.

CHAPTER 8

Conclusions and Future Work

***real**: (2b3) existing as a physical entity and having properties that deviate from an ideal, law, or standard.*

— Merrian–Webster English Dictionary

There are sadistic scientists who hurry to hunt down errors instead of establishing the truth.

— Marie Curie (1867–1934)

This dissertation proposed and evaluated a new approach for generating realistic traffic in networking experiments. Our construction relied on several components to form a coherent solution to this problem:

1. The a-b-t model of source-level behavior, which provides a *generic but detailed* way of describing source-level behavior that is applicable to any Internet application.

2. An *efficient measurement* method for accurately translating the packet header trace of any arbitrary TCP connection into its a-b-t connection vector, even in the presence of packet reordering and retransmission.

3. The source-level trace replay method for generating traffic in a closed-loop manner, which provides a way of introducing fully *reproducible* synthetic traffic in networking experiments.

4. The ability to directly compare original traffic and its source-level replay, after incorporating network parameters also derived from packet header analysis. Such a comparison enables us to *assess the realism* of the synthetic traffic.

5. A method for resampling a-b-t connection vectors that supports both the introduction of *controlled variability* in the generated traffic and the *predictable scaling* of the offered load.

The rest of this chapter discusses these components[1], highlighting some concrete contributions and open

[1] Also known as the five pillars of Abtism. [*sic*]

questions, which could be the subject of future work. Our focus is on the larger scheme of things, so we refer the reader to the summaries of each chapter for additional findings and possible refinement of our methodology.

8.1 Empirical Modeling of Traffic Mixes

The main problem solved by our approach is generating closed-loop traffic consistent with the behavior of the entire set of applications in modern traffic mixes. Unlike earlier approaches, which described individual applications in terms of the specific semantics of each application, we proposed to describe the source behavior driving each connection in a generic manner using the a-b-t model. This is consistent with the view of traffic from TCP, which does not concern itself with application semantics, but only with sending and receiving Application Data Units (ADUs) as demanded by the applications. The a-b-t model provides an intuitive but detailed way of describing source behavior. It also satisfies a crucial property: given a packet header trace collected from an arbitrary Internet link, we can algorithmically infer the source-level behavior driving each connection, and cast it into the notation of the a-b-t model.

Section 3.3 described our inference algorithm, whose asymptotic cost is is $O(sW)$, where s is the number of segments in a connection and W is the maximum size of the TCP receiver window (in segments). The foundation of the analysis is the *logical data order* that can be established between segments of the same connection. This order corresponds to the application-layer order of the data carried in each segment. From this order, we can accurately identify individual ADUs *without* any timing analysis. Furthermore, the handling of retransmission and reordering becomes very generic, eliminating the need to handle the many possible cases one by one. Our validation using traffic from synthetic applications with known source behavior demonstrated the robustness of our analysis to segment loss and reordering, and to the way in which endpoints use sockets (*i.e.*, using different sizes and timings of I/O operations).

Overall, our algorithmic approach enables us to model traffic in an automated manner in a question of hours. This addresses a major difficulty with earlier efforts targeted at individual applications, which required months to be completed and were hardly ever updated. One future direction is develop an online implementation of the algorithm, which would enable us to model traffic mixes in real time. The $O(sW)$ cost of our analysis makes this online processing feasible. Efficient memory management is the main challenge, since each connection would require separate state during its lifetime. It seems possible to restrict this per-connection state to the current ADU in each direction, which is much more efficient than keeping track of entire connection vectors. Real-time modeling has several benefits. First, the set of a-b-t connection vectors is between tens and hundreds of times smaller than packet header traces from which it derives. This would enable researchers to study traffic at the source-level for much longer periods that it is possible nowadays. Second, real-time modeling can remain active indefinitely, which makes it possible to observe unusual but important phenomena, such as flash crowds, BGP failures, *etc.* To satisfy storage constraints, uninteresting traffic can be periodically thrown away.

In our study, we identified a fundamental dichotomy between applications that exchange ADUs in a sequential manner and those that do it concurrently. Sequential communication follows an alternating sequence of ADUs sent in opposite directions, where ADUs from one endpoint usually play the role of requests and ADUs from the opposite endpoint play the role of responses. One important property of this pattern is that each ADU exchange must necessarily take one round-trip time. As a consequence, the duration of sequential connections often has little to do with the amount of transferred data, being dominated by the number of request/response pairs. For this reason, sequential connections usually show far lower throughputs than one would expect from their total number of bytes. SMTP provides a good example of this phenomenon, since most SMTP connections carry little data but take rather long to complete. As illustrated in Figure 3.3, this is mostly due the substantial number of control ADUs required by this protocol.

Concurrent communication supports the sending of ADUs from both endpoints at the same time. This is the natural model both for applications without requests and responses, and for applications that are able to pipeline their requests and responses. Pipelining eliminates the need to spend one full round-trip time to complete each request/response exchange, which can substantially increase throughput. The analysis of our collection of traces revealed that the number of connections that exhibit concurrent data exchanges is small (0.9-3.6%), but that they account for a far larger fraction of the total bytes in the traces (12.1-31.9%). This is consistent with the observation that concurrency can increase overall throughput, so application protocol designers are more compelled to use concurrency in applications that exchange large amounts of data. BitTorrent is a prominent example of data concurrency, where we can observe simultaneously natural concurrency (both endpoints send and received requests and file pieces), and pipelining (multiple requests and file pieces can be outstanding at any point in time). Figure 3.9 showed one example of this behavior.

Our measurement algorithm can determine whether a connection exhibits sequential or concurrent data exchanging by examining only the sequence and acknowledgment numbers in the segments of a connection, without analyzing of segment arrival times. The basis of our technique is again the logical data order among TCP segments, which is a total order for sequential connections, and a partial order for concurrent ones. The inequalities presented in Section 3.3.2 formalized this idea, providing a method for identifying data exchange concurrency without false positives.

8.2 Refining and Extending our Modeling

Our methodology strongly relies on *non-parametric modeling*. Parametric models are far more compact and can often provide deeper insights than non-parametric ones. However, their use has little to do with the quality of synthetic traffic. A non-parametric model can result in traffic as realistic or more than a parametric model, without the risk of oversimplification. In any case, our a-b-t connection vectors offer a good foundation for building a parametric model of Internet traffic mixes. Our analysis of the relationship between ADU sizes and numbers of epochs in Section 3.5.1 uncovered substantial complexity and a striking lack of consistency among the different links considered in our study. Techniques like Hidden Markov Modeling could perhaps provide the right approach.

Our own related work explored the possibility of attacking this complex modeling problem by decomposing traffic mixes in to a set of fundamental pattern of communication [HCNSJ05]. The idea was to use statistical clustering to find applications that behave in a similar manner, *i.e.*, that follow the same "communication pattern", and to separately model each of the identified *traffic clusters*. For example, interactive applications such as telnet and SSH are very different from file-sharing applications such as Kazaa or Gnutella, so it seems much easier to develop separate models for "interactive applications" and "file-sharing applications" than a single model to encompass both of them. In our exploratory study, we followed a two step process to find traffic clusters. First, we computed a vector of features for each connection, which included statistics such as the median size of the ADUs in the connection, a measure of the directionality of the data exchanges, and the correlation between the sizes of a-type and b-type ADUs. Feature vectors provide a way to compare connections, even if their a-b-t connection vectors have very different forms, and use a distance metric to quantify the similarity between the source behaviors in two connections. Second, we used a hierarchical clustering algorithm to construct a taxonomy of traffic classes based on the similarity among connections. The results of our analysis demonstrated that some clear and intuitive traffic clusters emerged when this procedure was applied to sets of connection vectors derived from real traces. We believe this type of approach can simplify the modeling of traffic mixes. Furthermore, it can also provide a more flexible way of resampling traces, where the fraction of connection vectors from each of the traffic clusters can be changed at will (*e.g.*, increasing of decreasing the fraction of file-sharing-like traffic).

There are other open questions in the modeling of Internet traffic mixes, and their solution is complicated by the need to devise better measurement methods. We can cite the following examples:

- Our modeling of concurrent connections employs two separate connection vectors, one for each direction, eliminating any dependencies among ADUs flowing in opposite direction. This dependencies are certainly present in some cases, at least when concurrency is used to implement pipelining. A refined version of the a-b-t model where the causality between ADUs is specified using an acyclic graph could capture this type of structure. The analysis of sequence and acknowledgment numbers can provide a starting point for understanding ADU dependencies. However, such an approach would result in a substantial number of spurious dependencies that were not really part of the application behavior.

- The a-b-t model has no mechanism to specify dependencies between ADUs in different connections. While more complex forms of the model are possible, there is again great difficulty in determining when these dependencies exists. By analyzing ADU arrival times for the same endpoint, we could hypothesize a dependency. We could further strengthen such an analysis by requiring several instances of the same dependency pattern, *i.e.*, only accepting a timing dependency when several pairs of connections with "similar" ADU sizes and number of epochs are observed.

- A important problem that has received very limited attention in the source-level modeling literature is the possibility of changes in user behavior as a function of network conditions. Such possibility would break the assumption of network independence in source-level models. Our work in this area [PHCM$^+$06] revealed phenomenal difficulties in measuring such dependencies. Even a simple question such as whether users with higher access bandwidths tended to download larger objects was statistically problematic. Our results showed that this trend does not appear to be present in the UNC trace. While substantial differences exists in the access bandwidth of different UNC endpoints (*e.g.*, between wireless and wired end hosts), the number of endpoints with severely limited bandwidth is very small (*e.g.*, few endpoints were behind a modem).

These three problems are unlikely to have straightforward solutions. We also believe that their impact on the quality of synthetic traffic is small, or even insignificant, in empirical studies focusing on large traffic aggregates.

A final question is how to combine source-level modeling and unwanted traffic modeling. Our analysis in Section 4.2.1 showed the need to carefully separate connections with regular data exchanges, for which the a-b-t model is applicable, and other types of connections (*i.e.*, failed connection establishments attempts, port and network scans, *etc.*). While our filtering for regular connections removed only a tiny fraction of the bytes in the traces, the number of individual connections was very large, which may be detrimental for certain studies. In addition, we did not consider how to generate malicious traffic. Our literature review discussed some relevant efforts on this topic. However, they tend to be open-loop. Since malicious traffic can have dramatic effect on the network conditions, understanding its impact on source behavior seems critical. We know of no study that considered this question.

8.3 Assessing Realism in Synthetic Traffic

The result of our packet header processing is a collection of a-b-t connection vectors, which can then be replayed in software simulators and testbed experiments to drive network stacks. Such a replay generates synthetic traffic that fully preserves the feedback loop between the TCP endpoints and the state of the network, which is essential in experiments where network congestion can occur. By construction, this type of traffic generation is fully reproducible, providing a solid foundation for networking experiments where two or more network mechanisms must be compared under similar conditions.

Our experimental work demonstrated the high quality of the generated traffic, by directly comparing traces from real Internet links and their source-level trace replay. This comparison is both a rigorous way of validating the a-b-t model and its measurement methods, and a challenging exercise where each connection vector must be replayed in a TCP connection whose original network conditions are preserved in the experiments. If these network conditions were not preserved, it would be very difficult to determine whether differences between an original trace and its source-level trace replay are due to shortcomings of the a-b-t model or to a lack of realistic network parameters. For this reason, we devote substantial effort to the accurate measurement, purely from packet header traces, of three important network parameters: round-trip times, maximum receiver window sizes, and loss rates. These three parameters have a major impact on the throughput that a TCP connection can achieve. In addition, the testbed experiments in our evaluation of the approach carefully reproduce these parameters, using an extended version of *dummynet* to efficiently simulate per-connection round-trip times and loss rates.

It is important to note that the inclusion of open-loop loss rates in some of our experiments is only a means to achieve a more fair validation of the a-b-t model. A substantial loss rate has a dramatic effect on the characteristics of a connection, so comparing such a connection in the original trace and in a replay without a simulated loss rate tells us very little about the accuracy of the source-level characterization. In general, we always conduct source-level trace replay experiments both with and without simulated loss rate, and compare their results. This type of analysis allowed us to conclude that source-level behavior had a more substantial impact on our traces than losses, but that neither of them can be ignored when trying to understand the characteristics of network traffic. One interesting finding from our experimental work is that simplistic source-level models substantially exacerbate the impact of losses, which may substantially change the conclusions from certain empirical studies.

Our results demonstrated that source-level trace replay can closely approximate the characteristics of real traffic traces. By comparing synthetic traffic with and without detailed source-level structure, we showed that more complete source-level modeling makes synthetic traffic closer or far closer to real Internet traffic. In particular, the largest difference was observed for the time series of packet throughput, the body of the packet throughput marginal and the time series of active connections. Other metrics did not show consistent improvement when detailed source-level modeling is used. However, in these cases, it is often difficult to determine whether the difference between real and synthetic traffic comes from the shortcomings of the source-level model or from the lack of certain network-level parameters. This is the main difficulty with our approach: while providing the most stringent way of evaluating synthetic traffic, it also requires to deconstruct the factors that shape traffic very carefully. While some factors are well understood and can be measured accurately, others are not. In this regard, our work complements current efforts to further understand traffic, provides a way to verifying new theories using an elaborate experimental approach.

One important future direction for our work is to expand the set of metrics used to evaluate the quality of synthetic traffic. At a low level, the distribution of packet sizes provides a good avenue to understand the effect of source behavior on packetization. At a higher level, the distribution of connection goodputs is a particularly good (and demanding) metric to study how closely the modeling (of sources and network parameters) reproduces TCP performance. We could study goodput either by looking at the distribution of connection goodputs directly, or by comparing each replayed connection with its original version and computing relative errors of some sort. Another important high-level metric is response time, which can be easily defined as the duration of epoch for sequential connections. Many studies rely on response time to examine the performance of network mechanism, so it is desirable to validate its experimental reproduction. However, there are several difficulties with this metric. It requires to identify request and response pairs, which are not necessarily the pair formed by ADUs a_i and b_i. The server side initiates the connection in some protocols, while other protocols do not have clearly-defined roles as client and server for their endpoint. It is very difficult to distinguish among these situations purely from packet header analysis. Also, there is no simple definition of response time for concurrent connections. As an alternative, we can use connection duration as a metric, which is always well-defined, but it has far lower resolution.

8.4 Incorporating Additional Network-Level Parameter

While our methods to measure and simulate network parameters appear sufficiently accurate in our experimental evaluation, there are several directions in which this part of the work can be refined. Path round-trip times are not fixed for each connection, but follow a distribution of delays. It seems possible to refine our measurement to incorporate this fact, at least to some extent, into our approach, although the lack of samples for most connections greatly complicates this problem. It is also unclear whether this refinement would have any significant impact on the generated traffic. Improving the measurement and simulation of losses could have a more substantial effect. Figure 4.18 already revealed some level of inaccuracy, and our experimentation revealed the need to take into account pure acknowledgment losses and not just data segment losses. More importantly, the assumption of independent losses and their simulation using random dropping seems unrealistic, which explains some of the differences between original and synthetic traffic.

There are other network parameters that could be taken into account. In general, we believe that only two of them would have a significant impact on the quality of synthetic traffic: maximum segment sizes and path capacity. Maximum segment sizes are straightforward to measure, and their incorporation into the experiments would improve the realism of packetization in the generated traffic. Its implementation in a network testbed experiments requires some careful handling of resources, since maximum segment sizes are often a machine-wide constant. The impact of this refinement is not expected to be dramatic, given that most connections are known to use the same maximum segments size (the one derived from Ethernet's MTU, which we employed in our experiments).

Path capacity presents a much more difficult measurement problem, both when defined as bottleneck capacity and as available bandwidth. Recent work by Huang and Dovrolis [JD04] provides a useful foundation. While it is only applicable with confidence to connections with large amounts of data, “bulk connections”, this is precisely the type of connection whose throughput could be dominated by capacity limits. Throughput in connections with small amounts of data is mostly a function of round-trip time. As discussed in Section 3.3, most connections are in this case. However, bulk connections are responsible for a large fraction of the bytes, so their accurate replay is important. We also believe that combining our ADU analysis with the Huang and Dovrolis approach can provide less noisy samples, improving the accuracy of the method. In the case of capacity, the implementation in the experiment is not difficult by making use of *dummynet* ’s per-connection capacity.

Besides these concrete specific network parameters, we believe that a better understanding of the impact of traffic shapers and end host bandwidth quotas can help to explain some of the differences between source-level trace replay experiments and original traffic. This seems specially relevant for UNC, where the impact of losses appeared rather different from the ones in other sites. We hypothesized that the presence of a major data and software repository known to use bandwidth constraints was behind our finding. Another important factor in traffic characteristics is the growing impact of wireless networks. Our large-scale measurement effort in this area [HCP05], showed an insignificant increase of end-to-end losses in this environment (thanks to link-layer retransmission) but substantial increases in the magnitude and variability of round-trip times.

8.5 Flexible Traffic Generation

The final problem that we considered in this work was how to introduce controlled variability in network experiments, *i.e.*, how to derive from a trace of connection vectors a new trace that still “resembles” the original one. Our solution involves resampling entire connection vectors, fully preserving observed source-level behavior, and assigning them new start times. We gave two methods for this assignment: sampling from an exponential distributions, which results in Poisson connection arrivals, and sampling blocks of connections, which preserves the long-range dependence in the connection arrival process that we encountered in our traces. The first method, Poisson Resampling, is analytically appealing, and supported by empirical data, since the marginal distribution of connection inter-arrival is consistent with an exponential distributions. Block Resampling provides a non-parametric alternative, which is more realistic with regards to the dependency structure of the connection arrival process. This structure did not show any effect on packet and byte arrivals, but it seems important for mechanisms that require per-connection state.

We also showed that our resampling methods can be carefully directed to produce a new trace of connection vectors whose offered traffic load matches an arbitrary target very closely. Such trace scaling is a common requirement in suites of experiments that must expose a network mechanism to a range of traffic loads. The key to our solution is to count the total amount of data in the resamplings, which was shown to be strongly correlated to offered load. On the contrary, our results clearly showed that the number of connections is only weakly correlated to offered load, and cannot be used for accurate scaling of resamplings. While this result is an intuitive consequence of the heavy-tailness in the amount of data carried by connections, the issue has been poorly understood in earlier models, where the parameters that can be controlled to tune offered load were associated with the number of connections. This is for example the case for the number of user equivalents in web traffic models. The traffic load offered by this type of "connection-driven" models can never match a target offered load as accurately as our "byte-driven" resamplings of connection vector traces.

Our work on trace resampling can be extended in several directions. First, there is some need to refine our handling of the packetization overhead, which would result in even more accurate load scaling. Second, our methods only manipulate one trace at a time. Being able to combine multiple traces would provide an even more flexible framework. While it seems straightforward to extend our methods to support this operation, demonstrating the validity of the results appears difficult. It represents a departure from measured traffic into a hypothetical traffic that may or may not be realistic, and it can introduce non-stationarities. Third, developing a broader model of network traffic, either parametric or non-parametric, could provide a better way to guide the resampling process. In this direction, a better understanding of the main patterns of source-level behavior would provide more flexible way of creating hypothetical scenarios. Our work on traffic clusters described above is a step in this direction, since combining clusters support the exploration of a wide range of traffic generation scenarios. The possibility of succinctly describing the range of patterns in a cluster, *e.g.*, file-sharing applications with symmetric bulk transfers and concurrency, is specially useful for exploring future scenarios where applications that only represent a small fraction of the traffic become increasingly important.

BIBLIOGRAPHY

[AA01] Masaki Aida and Tetsuya Abe. Pseudo-address generation algorithm of packet destinations for Internet performance simulation. In *Proceedings of IEEE Infocom*, pages 1425–1433. IEEE, 2001.

[AKM04] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proceedings of ACM SIGCOMM*, August 2004.

[AKSJ03] Jay Aikat, Jasleen Kaur, F. Donelson Smith, and Kevin Jeffay. Variability in TCP round-trip times. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, pages 279–284. ACM Press, 2003.

[APS99] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, 1999.

[AV98] P. Abry and D. Veitch. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, 44(1):2–15, 1998.

[AW95] Martin F. Arlitt and Carey L. Williamson. A synthetic workload model for Internet Mosaic traffic. In *Summer Computer Simulation Conference*, pages 24– 26, July 1995.

[Bar00] S. Barber. RFC 2980: Common NNTP Extensions, October 2000. Status: INFORMATIONAL.

[BBBC99] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella. Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, 1999.

[BC98] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, pages 151–160, 1998.

[BC99] Paul Barford and Mark Crovella. A performance evaluation of hyper text transfer protocols. In *Proceedings of ACM SIGMETRICS*, pages 188–197. ACM Press, 1999.

[BD99] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server under realistic loads. *World Wide Web*, 2(1-2):69–83, 1999.

[BEF$^+$00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[BHCKS04] A. Budhiraja, F. Hernández-Campos, V. G. Kulkarni, and F. D. Smith. Stochastic differential equation for tcp window size: Analysis and experimental validation. *Probab. Eng. Inf. Sci.*, 18(1):111–140, 2004.

[BHK$^+$91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–212, New York, NY, USA, 1991. ACM Press.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol — HTTP/1.0, May 1996. Status: INFORMATIONAL.

[Bra65] P.T. Brady. A technique for investigating on-off patterns of speech. *The Bell System Technical Journal*, pages 1–22, January 1965.

[Bra89] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989.

[CB96] Mark E. Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. In *Proceedings of ACM SIGMETRICS*, pages 160–169, New York, NY, USA, 1996. ACM Press.

[CBC95] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of WWW client-based traces. Technical report, Boston University, 1995.

[CCG$^+$04] J. Cao, W. S. Cleveland, Y. Gao, K. Jeffay, F. D. Smith, and M. Weigle. Stochastic Models for Generating Synthetic HTTP Source Traffic. In *Proceedings of IEEE Infocom*, 2004.

[CDJM91] Ramón Cáceres, Peter B. Danzig, Sugih Jamin, and Danny J. Mitzel. Characteristics of wide-area TCP/IP conversations. In *Proceedings of ACM SIGCOMM*, pages 101–112. ACM Press, 1991.

[CHC$^+$04a] Yu-Chung Cheng, Urs Hoelzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *USENIX Annual Technical Conference*, June 2004.

[CHC$^+$04b] Yu-Chung Cheng, Urs Hölzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for TCP tracing and replaying. In *USENIX Annual Technical Conference*, pages 87–98, 2004.

[CJOS00] Mikkel Christiansen, Kevin Jeffay, David Ott, and F. Donelson Smith. Tuning RED for web traffic. In *Proceedings of ACM SIGCOMM*, pages 139–150, 2000.

[CL97] Mark Crovella and Lester Lipsky. Long-lasting transient conditions in simulations with heavy-tailed workloads. In *Proceedings of the Winter Simulation Conference*, pages 1005–1012, 1997.

[CM99] P. Chaudhuri and J.S. Marron. SiZer for exploration of structures in curves. *Journal of the American Statistical Association*, 94:807–823, 1999.

[Coh03] B. Cohen. Incentives build robustness in BitTorrent. In *Conference Workshop on Economics of Peer-to-Peer Systems*, May 2003.

[Con04] Internet 2 Consortium. Internet2 netflow weekly reports. `http://netflow.internet2.edu/weekly`, 2004.

[Cor06] Cisco Corporation. Netflow. `http://www.cisco.com/netflow`, 2006.

[DJ91] Peter B. Danzig and Sugih Jamin. tcplib: A library of TCP/IP traffic characteristics. *USC Networking and Distributed Systems Laboratory TR CS-SYS-91-01*, October, 1991.

[Dow01a] Allen B. Downey. Evidence for long-tailed distributions in the internet. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, pages 229–241, New York, NY, USA, 2001. ACM Press.

[Dow01b] Allen B. Downey. The structural cause of file size distributions. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, page 361, Washington, DC, USA, 2001. IEEE Computer Society.

[ENW96] Ashok Erramilli, Onuttom Narayan, and Walter Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Transactions on Networking*, 4(2):209–223, 1996.

[ET93] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap.* Chapman & Hall, 1993.

[EV03] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[Fel88] D. Feldmeier. Improving gateway performance with a routing-table cache. In *Proceedings of IEEE Infocom*, pages 27–31, March 1988.

[Fel00] Anja Feldmann. Characteristics of TCP connection arrivals. In Kihong Park and Walter Willinger, editors, *Self-Similar Network Traffic and Performance Evaluation*. Wiley-Interscience, 2000.

[FGHW99] 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.

[FGM^{+}97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. RFC 2068: Hypertext Transfer Protocol — HTTP/1.1, January 1997. Status: PROPOSED STANDARD.

[FJ93a] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[FJ93b] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FK03] Sally Floyd and Eddie Kohler. Internet research needs better models. *ACM Computer Communication Review*, 33(1):29–34, 2003.

[FP01] Sally Ford and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, 2001.

[FTD03] Chuck Fraleigh, Fouad Tobagi, and Christophe Diot. Provisioning ip backbone networks to support latency sensitive traffic. In *Proceedings of IEEE Infocom*. IEEE, 2003.

[GcC02] Kartik Gopalan and Tzi cker Chiueh. Improving route lookup performance using network processor cache. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–10, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[GDS^{+}03] P. Krishna Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 314–329, 2003.

[GM01] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, page 180, Washington, DC, USA, 2001. IEEE Computer Society.

[GMP97] Ian Graham, Jed Martens, and Murray Pearson. The dag: an atm measurement board. In *4th Electronics New Zealand Conference*, 1997.

[GW94] Mark W. Garrett and Walter Willinger. Analysis, modeling and generation of self-similar vbr video traffic. In *Proceedings of ACM SIGCOMM*, pages 269–280, New York, NY, USA, 1994. ACM Press.

[HCJS^{+}01] F. Hernández-Campos, K. Jeffay, F.D. Smith, J. S. Marron, and A. Nobel. Methodology for developing empirical models of TCP-based applications, June 2001. unpublished manuscript.

[HCJS03] F. Hernández-Campos, K. Jeffay, and F. D. Smith. Tracking the evolution of web traffic: 1995-2003. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2003.

[HCMSS04] F. Hernández-Campos, J. S. Marron, Gennady Samorodnitsky, and F. D. Smith. Variable heavy tails in internet traffic. *Perform. Eval.*, 58(2+3):261–261, 2004.

[HCNSJ05] Félix Herández-Campos, Andrew B. Nobel, F. Donelson Smith, and Kevin Jeffay. Understanding patterns of tcp connection usage with statistical clustering. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 35–44, 2005.

[HCP05] Félix Hernández-Campos and Maria Papadopouli. Assessing the real impact of 802.11 WLANS: A large-scale comparison of wired and wireless traffic. In *14th IEEE Workshop on Local and Metropolitan Area Networks*, 2005.

[HCSJ04] F. Hernández-Campos, F.D. Smith, and K. Jeffay. Generating realistic TCP workloads. In *Proceedings of Computer Measurement Group (CMG) Conference*, December 2004.

[HNHC02] Peter Hall, Andrew B. Nobel, and Félix Hernández-Campos. Block bootstrap approach to simulating Internet traffic. Unpublished Manuscript, 2002.

[HV03] Nicolas Hohn and Darryl Veitch. Inverting sampled traffic. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, pages 222–233, New York, NY, USA, 2003. ACM Press.

[HVA02] N. Hohn, D. Veitch, and P. Abry. Does fractal scaling at the IP level depend on TCP flow arrival processes. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, 2002.

[Inc] NetIQ Software Inc. Chariot performance evaluation platform. `http://www.netiq.com/products/chr/default.asp`.

[Int] Internet Traffic Archive. `http://ita.ee.lbl.gov`.

[Jai90] R. Jain. Characteristics of destination address locality in computer networks: a comparison of caching schemes. *Computer Networks and ISDN Systems*, 18(4):243–254, 1990.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[JBB92] V. Jacobson, R. Braden, and D. Borman. RFC 1323: TCP extensions for high performance, May 1992. Status: PROPOSED STANDARD.

[JD02] Hao Jiang and Constantinos Dovrolis. Passive estimation of TCP round-trip times. *ACM Computer Communication Review*, 32(3):75–88, 2002.

[JD04] Hao Jiang and Constantinos Dovrolis. The effect of flow capacities on the burstiness of aggregated traffic. In *Passive and Active Measurement*, pages 93–102, 2004.

[JRF$^+$99] Y. Joo, V. Ribeiro, A. Feldmann, A. Gilbert, and W. Willinger. On the impact of variability on the buffer dynamics in IP networks. In *Allerton Conference on Communication, Control and Computing*, September 1999.

[JRF$^+$01] Youngmi Joo, Vinay Ribeiro, Anja Feldmann, Anna C. Gilbert, and Walter Willinger. Tcp/ip traffic dynamics and network performance: a lesson in workload modeling, flow control, and trace-driven simulations. *ACM Computer Communication Review*, 31(2):25–37, 2001.

[KBBkc03] Thomas Karagiannis, Andre Broido, Nevil Brownlee, and Michalis Faloutsos kc claffy. File-sharing in the Internet: A characterization of p2p traffic in the backbone. Technical report, UC Riverside, November 2003.

[KcLH$^+$02] Purushotham Kamath, Kun chan Lan, John Heidemann, Joe Bannister, and Joe Touch. Generation of high bandwidth network traffic traces. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 401–410, Fort Worth, Texas, USA, October 2002. USC/Information Sciences Institute, IEEE.

[KHR02] Dina Katabi, Mark Handley, and Charles Rohrs. Internet congestion control for future high bandwidth-delay product environments. In *Proceedings of ACM SIGCOMM*, 2002.

[KL86] B. Kantor and P. Lapsley. RFC 977: Network news transfer protocol: A proposed standard for the stream-based transmission of news, February 1986. Status: PROPOSED STANDARD.

[KLPS02] Eddie Kohler, Jinyang Li, Vern Paxson, and Scott Shenker. Observed structure of addresses in IP traffic. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, pages 253–266. ACM Press, 2002.

[KP88a] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of ACM SIGCOMM*, pages 2–7, New York, NY, USA, 1988. ACM Press.

[KP88b] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of ACM SIGCOMM*, pages 2–7, New York, NY, USA, 1988. ACM Press.

[KZ97] Edward W. Knightly and Hui Zhang. D-bind: an accurate traffic model for providing qos guarantees to vbr traffic. *IEEE/ACM Transactions on Networking*, 5(2):219–231, 1997.

[LAJS03] Long Le, Jay Aikat, Kevin Jeffay, and F. Donelson Smith. The effects of active queue management on web performance. In *Proceedings of ACM SIGCOMM*, pages 265–276, New York, NY, USA, 2003. ACM Press.

[LH02] Kun-Chan Lan and John Heidemann. Rapid model parameterization from traffic measurements. *ACM Trans. Model. Comput. Simul.*, 12(3):201–229, 2002.

[LTWW93] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *Proceedings of ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[Mah97] Bruce A. Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE Infocom*, volume 2, pages 592–600, 1997.

[MDG01] Joerg Micheel, Stephen Donnelly, and Ian Graham. Precision timestamping of network packets. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, pages 273–277, New York, NY, USA, 2001. ACM Press.

[MGT00] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of ACM SIGCOMM*, pages 151–160, New York, NY, USA, 2000. ACM Press.

[MH00] A. Mena and J. Heidemann. An empirical study of Real Audio traffic. In *Proceedings of IEEE Infocom*, March 2000.

[MHCS02] J. S. Marron, F. Hernández-Campos, and F. D. Smith. Mice and elephants visualization of Internet traffic. In *Proceedings of 15th Conference on Computational Statistics*, August 2002.

[Mit04] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2004.

[MJ93] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter USENIX Technical Conference*, pages 259–269, January 1993.

[MJ98] David Mosberger and Tai Jin. httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.

[MSM97] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM Computer Communication Review*, 27(3), 1997.

[Nag84] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, January 1984. Status: UNKNOWN.

[NIS06] NIST/SEMATECH. e-handbook of statistical methods. `http://www.itl.nist.gov/div898/handbook`, 2006.

[nlaa] National Laboratory for Applied Network Research (NLANR). `http://www.nlanr.net`.

[nlab] NLANR's Passive Measurement and Analysis (PMA) project. `http://pma.nlanr.net`.

[NSSW02] Carl Nuzman, Iraj Saniee, Wim Sweldens, and Alan Weiss. A compound model for TCP connection arrivals for LAN and WAN applications. *Computer Networks*, 40(3):319–337, 2002.

[Ost] David Ostermann. Tcptrace: a TCP connection analysis tool. `http://www.tcptrace.org`.

[Pax94] Vern Paxson. Empirically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, 1994.

[Pax97] V. Paxson. Fast, approximate synthesis of fractional gaussian noise for generating self-similar network traffic. *ACM Computer Communication Review*, 27(5):5–18, October 1997.

[PF95] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proceedings of ACM SIGCOMM*, pages 303–314, New York, NY, USA, 1998. ACM Press.

[PHCL$^+$] C. Park, F. Hernández-Campos, L. Le, J. S. Marron, J. Park, V. Pipiras, F. D. Smith, R. L. Smith, M. Trovero, and Z. Zhu. Long-range dependence analysis of Internet traffic.

[PHCM$^+$06] Cheolwoo Park, Félix Hernández-Campos, J. S. Marron, Kevin Jeffay, and F. D. Smith. Correlations of size, rate, and duration in tcp connections: the case against. In submission, 2006.

[PHCMS05] Cheolwoo Park, Félix Hernández-Campos, J. S. Marron, and F. Donelson Smith. Long-range dependence in a changing internet traffic mix. *Computer Networks*, 48(3):401–422, 2005.

[Pos81] J. Postel. RFC 793: Transmission control protocol, September 1981.

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

[PW00] Kihong Park and Walter Willinger, editors. *Self-Similar Network Traffic and Performance Evaluation*. Wiley-Interscience, 2000.

[RDFS04] Andy Rupp, Holger Dreger, Anja Feldmann, and Robin Sommer. Packet trace manipulation framework for test labs. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, pages 251–256. ACM Press, 2004.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[SB04] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, pages 68–81, New York, NY, USA, 2004. ACM Press.

[SBDR05] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. Improving accuracy in end-to-end packet loss measurement. In *Proceedings of ACM SIGCOMM*, August 2005.

[SGG02] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, 2002.

[SHCJO01] F. Donelson Smith, Félix Hernández-Campos, Kevin Jeffay, and David Ott. What TCP/IP protocol headers can tell us about the web. In *Proceedings of ACM SIGMETRICS*, pages 245–256, 2001.

[SRB01] S. Sarvotham, R. Riedi, and R. Baraniuk. Connection-level analysis and modeling of network traffic. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[Ste94] R. W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[SYB04] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, pages 82–87. ACM Press, 2004.

[tcpa] Tcpdump public repository. `http://www.tcpdump.org`.

[tcpb] Tcpreplay: Pcap editing and replay tools for *nix. `http://tcpreplay.sourceforge.net`.

[Wal99] J. Walker. *A primer on wavelets and their scientific applications*. CRC Press, 1999.

[WP98] Walter Willinger and Vern Paxson. Where mathematics meets the Internet. *Notices of the American Mathematical Society*, pages 961–970, September 1998.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[YVIB05] Tao Ye, Darryl Veitch, Gianluca Iannaccone, and Supratik Bhattacharyya. Divide and conquer: PC-based packet trace replay at OC-48 speeds. In *Tridentcom*, Trento, Italy, February 2005.

[ZRMD03] Z.-L. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-time scaling behaviors of Internet backbone traffic: An empirical study. In *Proceedings of IEEE Infocom*, San Francisco, March 2003.