

ABSTRACT

Title of dissertation: NETWORK STATE ESTIMATION VIA
PASSIVE TRAFFIC MONITORING

Ryan Lance, Doctor of Philosophy, 2005

Dissertation directed by: Professor James Yorke
Department of Mathematics

We propose to study computer network traffic as a dynamical system, with the intent of determining how predictable the traffic is over short time scales. We will use passive measurements from high capacity links, so that we may investigate traffic that consists of many diverse component flows. To study network traffic as a dynamical system one must first have a concept of what variables compose the state space. Transmission Control Protocol (TCP) regulates the dynamics of flows through two primary variables, the round-trip time and congestion window. These are obvious choices for the state variables, but they are not recorded in the passive measurements. Our main contributions in this dissertation are two algorithms that estimate round-trip times and congestion windows, and an auxiliary algorithm that determines flow orientation. We provide several validation tests for the algorithms, and use the results of the algorithms to infer the level of congestion in the measurements we use.

NETWORK STATE ESTIMATION VIA
PASSIVE TRAFFIC MONITORING

by

Ryan Lance

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:

Professor James Yorke, Chair/Advisor
Professor Brian Hunt, Co-Advisor
Professor Edward Ott
Professor Daniel Rudolph
Professor Neil Spring

TABLE OF CONTENTS

List of Tables	iv
List of Figures	v
1 Introduction	1
2 TCP Dynamics	4
2.1 Reliability	6
2.2 TCP Connection Rate	8
2.3 Routers and TCP Modifications	10
3 Passive Monitoring	13
3.1 Data Sets	13
3.2 Timestamp Errors	20
4 Flow Orientation	23
4.1 Bootstrap Algorithm	24
4.2 Overlap Algorithm	29
4.2.1 Practical Considerations	31
4.2.2 Algorithm Performance	34
4.3 Combined Algorithm	35
4.4 Validation Tests	36
4.4.1 Time To Live	37
4.4.2 Multiple Traces	40
4.4.3 Non-interspersed Traffic	42
5 Measurement of Network State	46
5.1 Interpacket Time Distribution	46
5.1.1 Bandwidth Delay Product	50
5.1.2 Cross Traffic, Queuing, and Delayed ACKs	52
5.2 RTT and Congestion Window Estimation	61
5.3 The Clustering Algorithm	62
5.3.1 Congestion Window Estimation	67
5.3.2 Real-time State Estimation	68
5.4 Frequency Algorithm	70
5.4.1 Sliding Window Upper Bound	71
5.4.2 Autocorrelation Function	73
5.4.3 Data-to-ACK-to-Data	78
5.4.4 Lomb Periodogram	80
5.4.5 RTT Lower Bound	83
5.4.6 Congestion Window Estimation	84

6	Validation and Prediction	88
6.1	Validating The Clustering and Frequency Algorithms	88
6.1.1	Clustering Algorithm Validation	93
6.1.2	Frequency Algorithm Validation	97
6.2	RTT and Congestion Window Statistics	99
6.3	Predicting TCP	104
6.3.1	Nonlinear Time Series Analysis	106
6.3.2	Incorporating Models	109
7	Conclusion	111
A	Drop Prediction	112
A.1	Congestion Statistics	113
A.2	Drop Correlation	115
	Bibliography	121

LIST OF TABLES

3.1	Description of monitoring points	14
4.1	Results of the overlap and bootstrap algorithms	37
4.2	Default TTL values	37
4.3	Consistency table for multiple monitors	42
6.1	Summary of flow orientations for all simulations	91
6.2	Properties of simulated cross traffic	92
6.3	Errors in RTT estimates of the clustering algorithm	95
A.1	Drop correlations for four simulated flows	118

LIST OF FIGURES

2.1	Protocol layers.	5
2.2	Seq-ack plot showing a drop event	7
2.3	Sawtooth transmission rate	9
3.1	Schematic network diagram	13
3.2	Header format diagram	17
3.3	Observed timestamp errors	22
4.1	Bootstrap algorithm	25
4.2	Overlap algorithm	31
4.3	Observed instantaneous bandwidth	33
4.4	Distribution of observed TTL values	38
4.5	Distributions of path length Γ	39
4.6	Distributions of path-related variable Δ	40
4.7	Non-interspersed traffic	43
5.1	Time line diagram showing the RTT, NIT and monitor	47
5.2	Interpacket time histogram - BUF flow	49
5.3	Estimated flight sizes - BUF flow	50
5.4	Interpacket time histogram and seq-ack plot under no congestion	53
5.5	Packet pacing diagram	54
5.6	Interpacket time histogram and seq-ack plot under some congestion	55
5.7	Interpacket time histograms - similar COS and FRG flows	59
5.8	Seq-ack plots - similar COS and FRG flows	60
5.9	Interpacket time histogram with NIT upper bound	63

5.10	Seq-ack plot showing the location of clusters	65
5.11	Histogram of prospective round-trip times with small variation	66
5.12	Estimated flight sizes via the clustering algorithm	67
5.13	Histogram of prospective round-trip times with large variation	68
5.14	Time series of estimated round-trip times with large variation	70
5.15	Underestimating the RTT with the sliding window method	72
5.16	Linearly interpolated interpacket times and corresponding autocorrelation function	75
5.17	Data-to-ACK-to-data RTT estimation	79
5.18	Comparison of the Lomb periodogram to the FFT	80
6.1	Network diagram for <i>ns2</i> simulations	89
6.2	Interpacket time histograms for two simulated flows	90
6.3	Estimated RTT from the clustering algorithm versus the true RTT .	94
6.4	Estimated congestion window from the clustering algorithm versus the true <i>cwnd</i>	96
6.5	Minimum RTT estimates from the frequency algorithm	98
6.6	Distribution of observed RTT values from all monitors	101
6.7	Distribution of observed RTT variation	102
6.8	Distribution of median estimated congestion window	103
6.9	Distribution of average transmission rate	104
6.10	Zeroth-order prediction errors	108
A.1	Drop indicator function, average queue and drop probability for a simulated flow	117
A.2	Distribution of observed drop correlation for a set of 400 flows	119

Chapter 1

Introduction

TCP is the protocol responsible for controlling the transmission rate of the majority of Internet connections. There are many mathematical models, both deterministic and stochastic, that describe how this rate changes. In order to use these models to predict the rate of real-world network traffic over short time scales, one must estimate the current state of the network. Considering a model as a dynamical system, the state estimate provides the initial condition. The key state variables in most models are the round-trip time and the congestion window, which are the main rate-controlling variables in TCP.

Analyzing actual network traffic data is difficult because the rate-controlling variables are not directly recorded and, hence, must be inferred. Therefore, estimating the network state is a necessary prerequisite to studying network traffic dynamics. We present two novel methods for reconstructing TCP state variables from passive network measurements. We then discuss the predictability of network flows in the context of dynamical systems and time series analysis.

The dynamics of network traffic have been studied from the theoretical point of view, resulting in mathematical models of TCP [15, 17, 30]. However, some of these models are limited by specific assumptions about the nature of TCP and the underlying network. The algorithms we have developed are quite flexible, and they

are able to handle a very wide variety of traffic conditions.

Previous studies of the statistics and dynamics of real network traffic have mainly focused on the burstiness [46, 23, 22], or self-similar properties [27, 39] of aggregate traffic. Studies that attempt to predict properties of individual connections [16, 43] have focused on coarse-grained characteristics like flow size and average rate. Our study will be concerned with the dynamical properties of individual flows and sets of flows.

By considering network traffic as a dynamical system, we can ascertain how deterministic it is; that is, given the current state of the system, we can determine the future state. Network traffic is a mixture of deterministic and stochastic components. There is no clearly defined boundary between what is deterministic and what is stochastic, but, in general, we can say that the deterministic components are large, long-lived flows, and the stochastic components are short-lived and sporadic flows. Brownlee and Claffy [7] have termed these flows elephants and mice. They found that the largest flows make up a very significant portion of the aggregate traffic. Therefore, in order to track overall network traffic conditions, it is reasonable to just track the state of the largest flows.

Our two state estimation algorithms use different yet complementary methods to approximate congestion windows and round-trip times. The first algorithm clusters packets together into coherent groups that approximate the congestion window, from which the round-trip time is then estimated. The second algorithm uses an array of approximations based on the frequency of packet spacing to estimate the minimum round-trip time, then estimate the congestion window using knowledge of

the minimum round-trip time.

We use the *ns2* network simulator [33] and publicly available packet-level traffic traces from NLANR [37] to validate and test our algorithms. With the NLANR traffic traces we encounter an interesting side problem of determining flow orientation. Before attempting to predict individual flows we set out to determine the prevalence of congestion in the flows we study.

In Chapter 2, we discuss the dynamics of TCP, define relevant terms, and introduce technical details that reappear throughout the paper. In Chapter 3, we describe the data sets and certain difficulties encountered with them. In Chapter 4, we provide a solution to the side problem of determining flow orientation. In Chapter 5, we detail two of the main contributions of this paper, the clustering and frequency algorithms for estimating round-trip times and congestion windows. In Chapter 6, we present a validation test of our algorithms, study the prevalence of congestion, and discuss prediction of individual flows.

Chapter 2

TCP Dynamics

TCP is one protocol in a layered stack of protocols, see Figure 2.1. When one uses a software application, like FTP, to send data between two computers, the application layer hands data off to the transport layer. There a protocol, usually TCP, breaks the data into segments and places a header at the beginning of each segment. The header contains information that allows the receiver to reconstruct the data in the proper order. The segment is then handed off to the network layer, where an IP (Internet Protocol) header is added. The IP header contains address information that allows routers to forward the data to the receiver. However, IP is unreliable; that is, a router is not required to pass on data packets if it becomes overwhelmed by other packets. When this happens a router may simply drop incoming data packets. At the network layer the data segment is called a packet. Below the network layer are the link and physical layers where other headers are added and the data is encoded before transmission.

TCP is the dominant transport level protocol on the Internet. According to MAWI [29] about 85% of the packets and 93% of the bytes on the Internet are TCP. In the traces we used, TCP comprises 86% of packets and 92% of bytes. These percentages can vary, but it is safe to assume that TCP carries the majority of the traffic on a given network. The main purpose of TCP is to guarantee the reliability

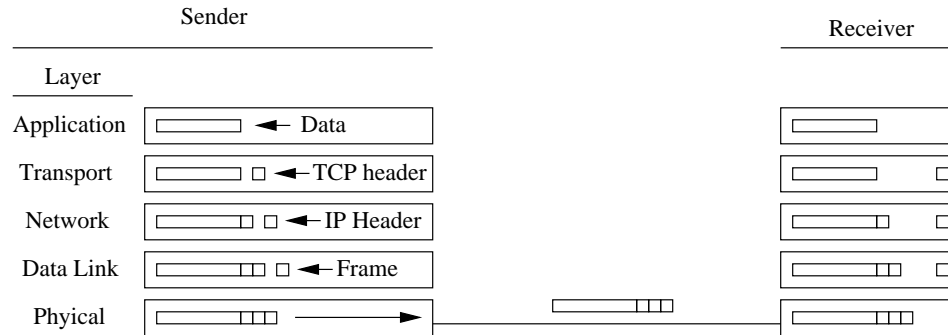


Figure 2.1: TCP is one layer in a protocol stack. TCP provides reliability and controls the transmission rate. IP controls routing and end-to-end delivery. Protocols on the data link layer further encode the packet for error correction. Bits are transferred as pulses of electrons on the physical layer.

of the connection – since IP does not – and ensure that packets are reassembled in the proper order. The secondary function of TCP is to control the rate at which data is transferred.

To illustrate the logic of TCP’s reliability and rate control algorithms consider what we call the “postcard analogy.” If one wanted to send a book to a friend in a far away country that has an unreliable mail system, then sending the whole book at once would likely fail due to the unreliable mail system. Instead one could have an arrangement whereby one would send the first few pages of the book in separate envelopes and wait for the friend to reply with a postcard for each envelope. Upon receipt of all expected postcards one would then mail more pages, increasing by one the number of pages sent each time until an envelope is not delivered. One can deduce that this is the level at which the unreliable mail system fails, and accordingly one should decrease the rate at which pages are sent. We now describe

the specifics of how TCP implements these ideas.

2.1 Reliability

In long-lived flows, such as file transfers, most of the data is going from one host, the sender, to the other host, the receiver. Reliability is achieved by requiring the receiver to acknowledge the successful receipt of each data packet. Acknowledgments (ACKs) are 40 bytes and consist of only IP and TCP headers. Data packets usually are 1500 bytes, which is the maximum transmission unit on most networks. However, that includes the header length, so packets typically contain only 1460 bytes of data. In Chapter 3 we discuss the header fields and the format of the trace files.

The TCP header also contains sequence and acknowledgment numbers, which are unique to each packet in a flow. Every connection starts with a 32-bit sequence number, chosen essentially at random. After each data packet is sent, the sender increments the sequence number (mod 2^{32}) by the number of data bytes in that packet. The receiver can deduce when a packet has arrived out of order or a packet has been dropped by checking for gaps in the sequence numbers. The acknowledgment number is the value of the next sequence number that the receiver expects to see. For example, if a data packet with sequence number S is dropped, the receiver will continue to send ACKs with acknowledgment number S . Upon receiving three duplicate ACKs with the same acknowledgment number, the sender infers that the data packet with sequence number S was dropped and retransmits it. The sender

may also infer that a packet has been dropped if it fails to receive an ACK within a given time limit, this is known as a *timeout*.

As a general rule the acknowledgment numbers always form a nondecreasing sequence. The sequence numbers form an increasing sequence except when a dropped packet is retransmitted. An extremely helpful tool for understanding the dynamics of TCP is the *seq-ack plot*, which shows the progression sequence and acknowledgment numbers over time, see Figure 2.2.

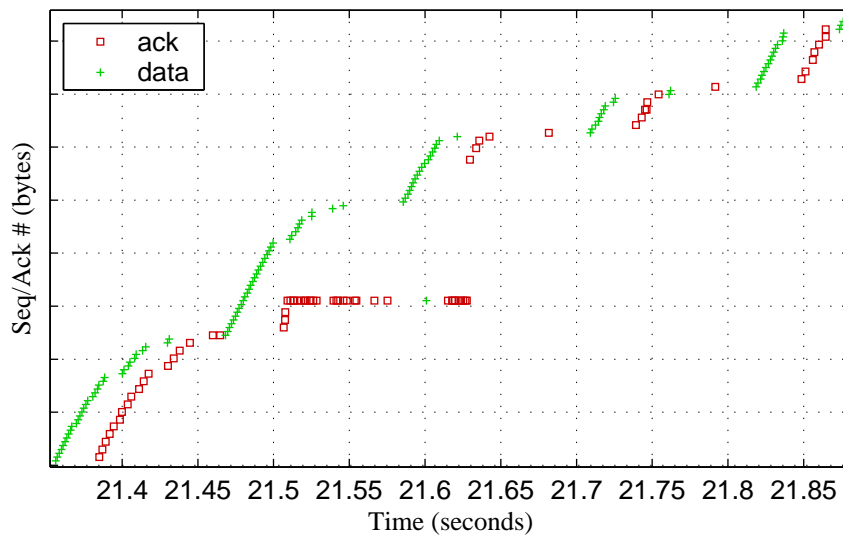


Figure 2.2: Seq-ack plot for an ssh flow from July 17, 2003 taken at the MRA monitor. A packet is dropped just before 21.5 seconds, and the acknowledgment number starts repeating soon after. The dropped packet is retransmitted at 21.6 seconds and the window is halved. The vertical axis is not labeled, since the actual values are irrelevant.

2.2 TCP Connection Rate

The transmission rate of a TCP connection is controlled by a congestion avoidance algorithm, that uses a sliding window known as the congestion window (*cwnd*). The sender is constrained to have no more than *cwnd* unacknowledged data bytes, not including header length, on the network at one time. This constraint means that under the usual regime the sender transmits packets in flights that total *cwnd* bytes.

Before discussing how *cwnd* changes, we mention two restrictions on it. Since most data packets carry 1460 bytes of data, one often finds that *cwnd* is a multiple of 1460. Hence the sender can transmit a window of packets without breaking the data into segments smaller than 1460 bytes. Because of this we may consider the window to be measured in packets instead of bytes. The second restriction on *cwnd* is that it is bounded above by the window advertised by the receiver. We will refer to this advertised window as the receiver's buffer to avoid confusion. The receiver communicates this value to the sender via the TCP header in acknowledgments. We feel the term buffer is more appropriate since the receiver maintains a buffer of newly arrived packets. The size of the receiver's buffer depends on the operating system, and can change with time if the receiver becomes too busy to process all of the incoming packets. In this way it acts as an upper bound on *cwnd*, so that the receiver is not overwhelmed by data.

The purpose of the sliding window algorithm is to find the right balance between transmitting data as fast as possible and not overwhelming the network or

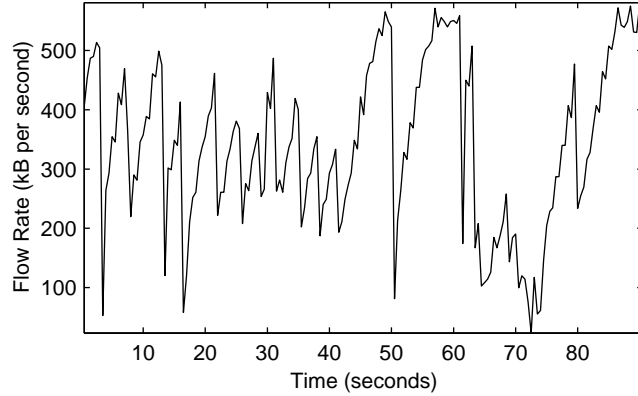


Figure 2.3: Flow rate of the same ssh flow in Figure 2.2. Note that the rate does not exceed 600 KBps.

the receiver. The network is more likely to be the bottleneck than the receiver. In fact, the receiver's buffer is artificially low in many cases. The process begins with the slow start algorithm, which finds the proper value of *cwnd* as quickly as possible. At the start of a connection *cwnd* is usually set to two packets. The value of *cwnd* doubles after each successful transmission of an entire window of packets. This doubling continues until a packet is dropped or *cwnd* reaches the value of the receiver's buffer. If a packet is dropped it is surmised that the connection has exceeded the available bandwidth, and the value of *cwnd* is halved. TCP then enters the congestion avoidance phase.

The objective of the congestion avoidance (CA) algorithm is to use as much of the available bandwidth as possible without congesting the network. The CA algorithm increments the window by one packet after all packets in the previous window have been acknowledged. The window grows linearly until it reaches the receiver's buffer or a packet is dropped. If *cwnd* reaches the receiver's buffer size,

then it is capped at that value. If a packet is dropped and the sender receives three duplicate ACKs, then *cwnd* is cut in half, see Figure 2.2. If a packet is dropped and the sender receives no ACKs at all, then a timeout occurs and *cwnd* is reset to two packets. This arithmetic increase, multiplicative decrease (AIMD) algorithm results in characteristic sawtooth behavior, see Figure 2.3. It is this behavior that we will locate, quantify and, if possible, predict in real network traffic.

Returning to the postcard analogy, we note that the pages of the books represent the data segments and the envelopes represent the TCP and IP headers. Acknowledgments are represented by postcards. One could think of the receiver's buffer as the friend's mailbox and routers as akin to unreliable mail carriers.

2.3 Routers and TCP Modifications

Another important element governing TCP dynamics are routers along the path of a connection. We define the fixed delay as the time from the departure of a data packet to the return of its corresponding acknowledgment, in the absence of cross traffic and queuing at routers. The reason for defining it this way is because larger packets take longer to transmit and the forward and backward paths of the connection might differ. The round-trip time (RTT) is defined as the fixed delay plus any queuing delay incurred; thus, most variation in the RTT is due to queuing.

Routers have various queue management schemes that use different criteria to drop packets. Droptail is the simplest; it prescribes that all incoming packets be dropped when the queue is full. Random Early Detection (RED) [14] uses an

exponentially weighted average queue and drops packets randomly with a probability that is roughly proportional to this weighted average. There are other queue management schemes, but the exact scheme used will not matter for most of our analysis. In fact, it is difficult to ascertain what queue management schemes are used in practice since that information is often considered proprietary.

Routers' queues play a large role in the dynamics of TCP connections, but it is difficult to obtain fine-grained traces of routers' queues. The reason is that recording the queue length every time it changes would put an undue burden on the router, taking away resources needed to process incoming packets. This would decrease the router's performance, perhaps affecting the flows passing through the router [18]. This is one reason we prefer passive measurements over active ones.

We note two optional TCP modifications that have practical impacts on our work. First is the use of delayed acknowledgments as described in RFC 2581 [5]. When using delayed acknowledgments the receiver acknowledges every other data packet. In practice this results in a sequence of *cwnd* values that does not increase by one packet per RTT. Instead, *cwnd* typically increases by one packet every two RTTs. The second modification of concern to us is the window scale option described in RFC 1323 [19]. The receiver's buffer is normally limited to 64 KB, or about 44 packets, because a 16-bit field in the header is used to represent the bytes available in the buffer. For flows on very high bandwidth networks or flows with very long fixed delays this limit on the receiver's buffer can be quite a hindrance. The window scale option allows the receiver to increase the value of its advertised window to 1 GB.

In the usual regime under which TCP operates, in the absence of packet loss, the sender transmits a window of *cwnd* bytes once per RTT. Since *cwnd* is normally limited to the artificially small value of 64 KB, one often finds that the data packets are not evenly spaced over a round-trip time interval, but are bunched together with a pause on the order of a round-trip time until the acknowledgments return. This shows that the flow is not close to using the available bandwidth. However, if cross traffic becomes interspersed with the flow, then the spacing between packets can increase and remain that way even after the cross traffic has subsided. This illustrates what is referred to as the self-clocking nature of TCP.

Chapter 3

Passive Monitoring

3.1 Data Sets

We use data sets from the Passive Measurement and Analysis archive at NLANR [37]. These 90 second traces are taken by monitors on OC3 (155 Mbps), OC12 (622 Mbps) and OC48(2.4Gbps) links at points around the United States. The monitors positioned as shown in Figure 3.1. Many other kinds of measurements are taken at the endpoints of a connection, but these traces are taken on links from university networks to the Abilene Backbone Network [1]. See Table 3.1 for a description of the monitoring points.

We define a connection, or flow, as the exchange of packets involving a unique combination of source and destination addresses and ports. It is also possible to use the less restrictive definition of a unique pair of addresses, but we found this to be confusing because some applications distribute the transfer of data across multiple

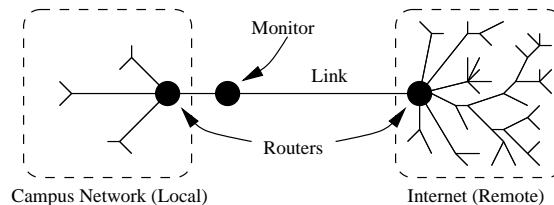


Figure 3.1: Schematic network diagram. Hosts on the local side of the network are likely to be closer to the monitor, both in physical distance and in time.

Name	Link	Traces	Dates	Packets	Flows	Top 0%
PSC	OC48	54	1/05 - 3/05	6185000	88000	45
UFL	OC12	90	5/04 - 3/05	7407000	147000	215
FRG	OC12	107	1/04 - 3/05	5279000	55000	30
MRA	OC12	126	3/03 - 3/05	6667000	201000	75
AMP	OC12	87	9/04 - 3/05	6489000	38000	30
COS	OC3	160	1/04 - 3/05	1322000	93000	150
MEM	OC3	97	1/03 - 3/05	320000	16000	10
ODU	OC3	96	1/03 - 3/05	1614000	45000	100
BWY	OC3	30	3/03 - 9/04	1583000	105000	25
BUF	OC3	21	1/03 - 9/03	1220000	7000	30
TXG	OC12	15	3/03 - 1/04	1412000	24000	40

Table 3.1: Monitoring Points: PSC is Pittsburgh Supercomputing Center, UFL is University of Florida, FRG is Front Range GigaPOP in Colorado, MRA is MERIT at Michigan State University, AMP is AMPath in Miami, COS is Colorado State, MEM is University of Memphis, ODU is Old Dominion University, BWY is NYSERNet at Columbia University BUF is University of Buffalo, TXG is Texas GigaPop at Rice University Packets and flows are averages per 90 second trace. The last column is the average number of bulk TCP flows that make up the top 50% of the aggregate bytes.

ports. Splitting the transfer across several connections is one way to overcome the limit placed on the connection rate by the receiver's buffer. There are a few unusual types of TCP flows that appear in the traces due to distributed download and file-sharing applications that use TCP in creative ways; these flows make up a nontrivial portion of the aggregate traffic.

As Table 3.1 shows there are many thousands of flows per trace. Most of these flows are very small, in fact, the median flow size is only a few hundred bytes. We are only interested in a small subset of the flows, the large "bulk" TCP flows. One could define a *bulk flow* as a flow that has an average throughput above a certain threshold, say, 10 Mbps over a minimum sustained period, say, 10 seconds. However, we use a different definition that is relative to the total amount of traffic on a link. We first sort flows based on their total size, and only consider the top 50%, that is, the minimal subset of flows that compose at least 50% of the total bytes in the trace. The top 50% may contain both UDP and TCP flows. We define bulk TCP flows as those in the top 50%. The reason for using this definition is that, in addition to predicting individual flows, we are concerned with predicting the aggregate traffic on a link. Although 50% is an arbitrary cutoff, one can think of the top 50% as mostly deterministic and the bottom 50% as mostly stochastic.

Since the traces are only 90 seconds most of the bulk TCP flows are not complete flows. The median number of packets for flows in the top 50% varies appreciably from monitor to monitor. For most of the monitors the median number of data packets was between 1600 and 6000, the outlier was the FRG monitor, for which the median was 28000. The lowest of the seven was the COS monitor, its

relatively small flows can be at least partially attributed to traffic shaping. The relatively large size of the FRG flows might be due to the prevalence of transfers of large files of meteorological data. The main application used to transfer this data is Unidata-LDM [48], which uses port 388, and about 40% of the FRG bulk TCP flows use port 388.

The monitors themselves consist of optical splitters for each direction of the link, connected to a fairly recent server with two DAG network cards specially designed by the WAND group at the University of Waikato in New Zealand [49]. Since the splitters take a fraction of the light from the optical fiber the measurements are completely passive. None of the monitored links use wavelength division multiplexing (WDM). If they did, then two packets traveling in the same direction might have the same timestamp.

There is, however, the possibility of traffic shaping on some of the monitored networks. Since the monitored networks are universities, the traffic comes from a mix of scientific and recreational applications. Universities may restrict the bandwidth available to traffic on ports associated with file sharing applications like KaZaA, BitTorrent and Gnutella. Traffic shaping may be implemented by placing packets associated with those applications in a separate, lower priority queue. We make no assumptions about traffic shaping in our analysis.

The monitor records the header of every packet into the trace file. The format for each record in the trace consists of a timestamp with microsecond precision along with the IP and TCP headers, see Figure 3.2. The first timestamp field is 32 bits, because the timestamp is measured in seconds since the Unix epoch,

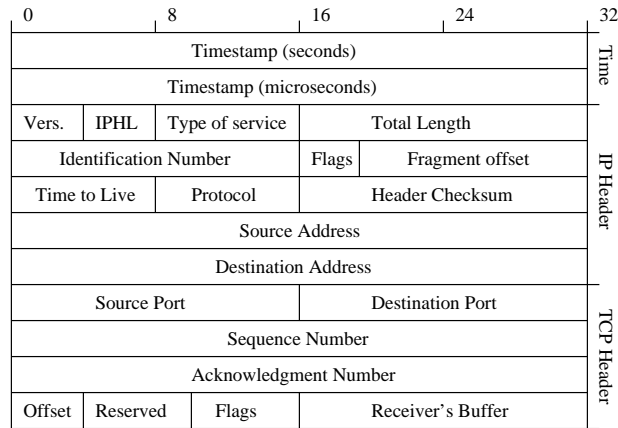


Figure 3.2: Header format diagram. Each row is 4 bytes for a total of 44 bytes per record. Note that the checksum, urgent pointer and options of the TCP header are not included in the trace record.

00:00:00 UTC January 1, 1970. In order to guarantee privacy, the IP addresses are renumbered so that the true IP addresses are anonymized, while the new addresses remain consistent throughout the course of the trace. Other fields of interest in the headers are the IP version, time to live (TTL), and the flags field of the TCP header. Almost all traffic on the Internet is IP version 4, but IP version 6 is increasing in prevalence. The TTL field has an initial value that depends on the operating system, but it is usually close to a power of two, either 64, 128, or 255 [32]. When a packet passes through a router, the TTL value is decreased by one. When a router receives a packet with a TTL of 0, that packet is discarded to prevent packets from getting stuck in routing loops. The flags field of the TCP header contains six one-bit flags including the ACK flag to denote an acknowledgment, as well the SYN and FIN flags, which signal the beginning and end of a connection, respectively.

The traces also contain UDP, ICMP, and other traffic. These other protocols

can interact unfavorably with TCP; many denial of service attacks flood the victim host with UDP or ICMP packets. Even though UDP usually composes less than 10 percent of the aggregate traffic, it is possible that percentage could increase in coming years with the growth of online gaming, streaming video, and new technologies like Voice over IP (VoIP). Some UDP flows are not aware of congestion, and continue to send data at a constant rate even when routers are overburdened.

We also note that even though the monitored link might not be the only gateway in and out of the university network, it is probably safe to assume that for each connection one host will be inside the university network and the other host will be outside. It is rare that a connection between two remote hosts would follow a path that leads into the university network on the monitored link and out of the network on another link. This is not a key assumption for our purposes, but just an indication of the nature of the traffic with which we will be dealing.

Both directions of traffic on the link are recorded to the same file without indicating the direction in which packets were traveling. This poses a problem, because we would like to separate inbound flows to the university network from outbound flows. It is possible to do this even though we do not know the real IP addresses. We have developed novel techniques that use only network trace data, to deal with this problem. We will discuss these techniques in Chapter 4. We do not circumvent the privacy given by the anonymization of the IP addresses.

There are various methods of passive and active monitoring, but we chose to use passive monitoring data despite some of its drawbacks. If one were monitoring a connection at one of the end hosts, then the problem of estimating the RTT

and congestion window becomes trivial since they could be directly measured. The trade-off is that one could only monitor the connections emanating from a single host and thus would have no information regarding cross traffic. One could conceivably coordinate many host-based monitors across an entire network to gain the same information as a single link-based monitor, but that would require synchronization of timestamps and a central collection point for the individual traces, and this seems impractical. Interesting alternatives include using a distributed approach [10] or a peer-to-peer application [35] to collect and share congestion information. By using the traces from NLANR in which all traffic on a link is measured we gain insight into how different flows interact on the network.

Active measurement involves sending probe packets into the network to infer network characteristics. Some examples of active measurement tools are skitter, scamper and Beluga from CAIDA [8], the IP Measurement Protocol from NLANR [2], Van Jacobson's *pathchar* tool [31], and the more advanced *pathrate* tool developed by Dovrolis et al. [38]. The probe packets do not transport any data from one host to another, and therefore decrease the bandwidth available to applications that do. If enough probe packets are sent it is possible that they could change the network characteristics they are being used to measure, resulting in a kind of uncertainty principle [36]. For these reasons we favor passive monitoring over active monitoring.

3.2 Timestamp Errors

There are several different kinds of timestamp errors that occur in the traces, which we describe in this section. Some errors might be hardware errors associated with the DAG network cards used in the monitor, while other errors could be introduced in the post-processing phase that anonymizes the traces. Most of the monitoring points have some form of timestamp errors. Almost all of the errors can be discovered and handled rather easily as the trace file is read in linear order.

The first type of error is the completely erroneous timestamp: the timestamps in the offending records differ from the preceding and following timestamps by thousands of seconds. When this error occurs in the first few packets of the trace file it can safely be ignored and the rest of the file can be processed as usual. However, if this error occurs after the first 100 packets or so, then it is likely to be the first in a long stretch of completely erroneous timestamps, lasting for most of the trace, see Figure 3.3(a). More than one stretch can appear in a trace file. To detect completely erroneous timestamps, we check that timestamp $n + 1$ and timestamp n differ by less than 90 seconds; if they do not, then we ignore the rest of the trace file, as it is unlikely to contain much accurate data.

Another type of error is the randomly repeated packet. When this occurs, an exact copy of the record of a packet is repeated later in the trace file. This can happen dozens of times in one trace file, and the duplicate packets almost always occur after the originals, see Figure 3.3(b) and (c). A related error is an entire sequence of repeated packets, consisting of anywhere from hundreds to millions of

packets. Typically only one direction of traffic will be repeated, see Figure 3.3(b). A rarer type of error is the non-interspersed error, wherein packets from both directions are not interspersed in the trace file for a short time, usually consisting of hundreds or thousands of packets, see Figure 3.3(d). For a lone repeated packet or a sequence of repeated packets the solution is the same. If timestamp n is strictly greater than timestamp $n + 1$, then we continue to read from the trace file without saving the records in memory, until we encounter a timestamp that is greater than or equal to timestamp n . For the non-interspersed error this means we will skip several hundred packets from one direction of the trace, but this will not have a large impact on our analysis.

The most difficult error to correct is the time shift error, wherein the timestamps from two directions of the link are not synchronized. Timestamps from one direction are shifted with respect to the other direction; the shift is almost always exactly one second, see Figure 3.3(e) and (f). This error could be globally corrected by adding one second to all the timestamps traveling in the appropriate direction, if we knew the direction that each packet was traveling along the monitored link. We can infer the flow orientation, as discussed in Chapter 4, but we still can not be sure of the orientation of every packet. Instead we opt to correct timestamps on a flow-by-flow basis. We will be able to make the necessary adjustment to the timestamps by examining the time from a data packet to its corresponding acknowledgment; if that time is approximately negative one second then we subtract one second from timestamp of the data packets, otherwise we subtract one second from the acknowledgments.

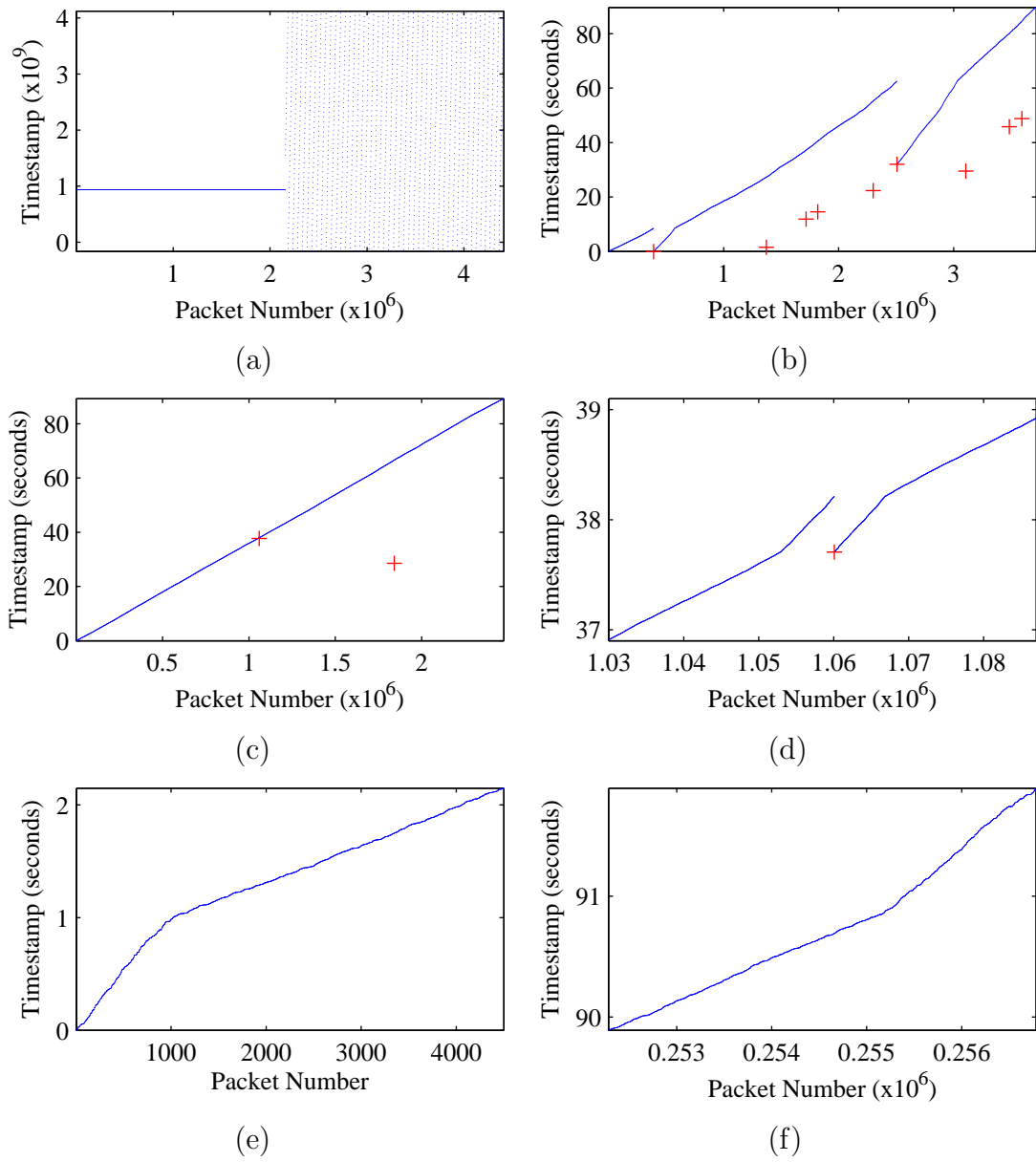


Figure 3.3: Observed timestamp errors. (a) After 2 million packets the timestamps become flawed. (b) The crosses are single repeated packets, there are also two sequential repeats. (c) The cross off the diagonal is a lone repeat, but the cross on the diagonal is actually the non-interspersed region in (d). (e,f) Although the same time period was recorded, the timestamps overhang for the first second (e) and the last second (f). The trace appears 92 seconds long, but the timestamps are shifted.

Chapter 4

Flow Orientation

Trace files do not record the direction each packet traveled on the monitored link. In this chapter, our goal is to determine *flow orientation*, that is, the direction each packet was traveling as it passed through the monitoring point. Considering the topology of the network around the monitoring point, another way of stating the problem is: Of the two hosts involved in each flow, which is inside the university network, and which is outside? This assumes that there is a notion of inside and outside the university network. The monitors are positioned at the edge of the university networks on links connecting the networks to the Abilene Backbone Network. It is very unlikely that a flow passing through the monitor would involve two hosts inside the network or two hosts outside the network. Therefore, the monitor itself acts as a boundary between hosts inside and outside of the campus network.

We will present two novel algorithms for inferring flow orientation. We call these methods the bootstrap and overlap algorithms. We will also describe how to combine the two algorithms for maximum coverage, and discuss possible extensions and improvements to the algorithms.

4.1 Bootstrap Algorithm

Consider the graph where the vertices are all hosts (IP addresses) in a given trace file and two vertices are connected with an edge if there is a flow involving the hosts in the trace file. We call this the IP connection graph. Depending on the size of the trace file, this graph may be quite large, with hundreds of thousands of vertices and edges. The graph will also have many connected components. The goal of the bootstrap algorithm is to identify the largest connected component (the one with the most vertices) of the IP connection graph. It is possible to weight the edges of the graph by the number of bytes exchanged between hosts, and in this case the largest component would be the one with the most bytes transferred. In the process of finding the largest connected component, the bootstrap algorithm also identifies a bipartite subgraph of the component. A graph is *bipartite* if its vertices can be partitioned into two non-empty subsets such that no edges join vertices in the same subset.

The algorithm is based on the assumption that for each flow one host is inside the university network and the other is outside. Suppose host A is inside the university network and connects to a host B outside. Suppose host B also connects to a host C , and this flow passes through the monitor. By our assumption it follows that host C is inside the university network. The bootstrap algorithm uses this method to assemble a list of hosts labeled as either inside or outside of the university network.

We must first posit that one host, called a seed host, is either inside or outside

the university network. We chose the name bootstrap because the algorithm works from the assumption about the seed host and uses no a priori knowledge of the real IP addresses. The bootstrap algorithm starts with a list of hosts in the trace, all of them initially unlabeled. If we posit that the seed host is on the local side of the monitor (inside the university network), then in the first step of the algorithm we label it L (local). In the second step we label as R (remote) all of the hosts connecting to the seed host. We label as L the hosts connecting to those in the second step, and so on. The algorithm usually takes 5-25 steps to label all the hosts in the component of the graph containing the seed host. As is shown in Figure 4.1, the bootstrap algorithm essentially makes a breadth-first traversal of the IP connection graph while alternating labels in order to identify a bipartite subgraph.

The choice of the seed host is critical because the output of the algorithm is very sensitive to it. We choose the seed host to be the host with the greatest

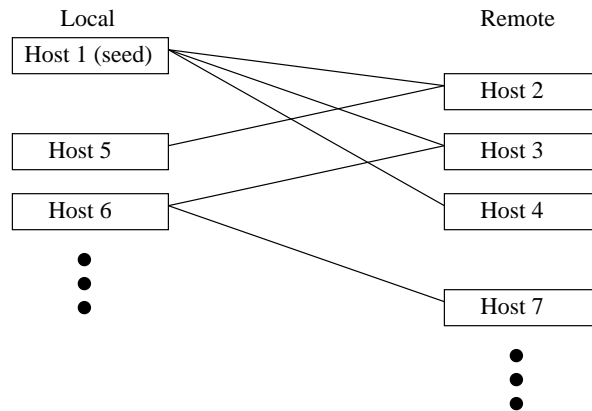


Figure 4.1: With the assumption that host 1 is on the local side of the monitor, the bootstrap algorithm traverses the IP connection graph in a breadth-first manner. Hosts are labeled local and remote on alternating steps.

number of connections to distinct hosts; in the graph, this is the node with the largest degree. There will be many hosts that the algorithm does not label, because they are in other connected components of the graph. Many of these connected components have only two nodes, where the hosts connect only to each other.

It could be the case the seed host connects to 1000 different hosts, but those hosts connect to only the seed host and no other, so the algorithm could not proceed beyond the second step. Although that is a possibility, we have generally found that choosing the seed host by degree as described above finds the largest connected component. If not, then choosing another host with a large degree often works. If we rank the hosts in terms of the number of distinct connections they make, then, on average, 8 out of the top 10 will result in the largest connected component when chosen as the seed host. One problem with initializing the algorithm this way is that it requires an extra pass through the trace file to first identify the hosts with the greatest number of distinct connections. The problem of initialization is solved by combining the bootstrap and overlap algorithms as detailed in Section 4.3.

Suppose we have a situation where, among three hosts, A , B and C , A connects to B , B connects to C and C connects to A . Also suppose that A and C are local and B is remote. Given the position of the monitor we would not expect to see any packets of the flow involving A and C in the trace file, since that would require packets to leave and reenter the university network. A similar possibility exists when A and C are outside the university network. To deal with such situations, we keep a list of host pairs, called an inconsistency list, where the hosts have a connection between them, but they are labeled as being on the same side of the monitor. Note

that these inconsistencies can be attributed to several factors, although we can not be absolutely certain that these factors explain all inconsistencies. Routing errors and address spoofing are two possibilities. Address spoofing – where a host uses the IP address of another host in place of its own – is the most likely candidate. Whatever the cause of inconsistencies, there is a way to correct them. Upon completion of the bootstrap algorithm, we run the algorithm a second time, this time discounting flows involving hosts on the inconsistency list, thus forming a bipartite subgraph of the largest connected component. We do not claim that this is a maximal bipartite subgraph; for a rigorous approach to finding bipartite subgraphs see Erdős et al. [12, 13].

Recall that the algorithm must be initialized by choosing a seed host that we posit to be on the local or remote side of the monitor. If that initial assumption was incorrect, say we assigned the seed to be on the local side when it was actually on the remote side, then all the hosts labeled by the algorithm would be labeled incorrectly. Once we realize the initial imposition was incorrect we can simply reverse all of the labels, but how do we recognize that the initial assumption was wrong? In general, more hosts should be labeled as remote than local if the output is correct, but that is not always guaranteed. A better approach is the use of the Time to Live (TTL) values of the labeled hosts.

The TTL is found in the IP header and prevents packets from getting stuck in routing loops. At the source host the TTL is set to a value between 1 and 255, depending on the operating system. Each router the packet passes through decrements the TTL by one. If a router receives a packet with a TTL of zero the

packet is discarded. Common starting values for the TTL are often powers of two, usually 32, 64, 128, or 255. Therefore, it is appropriate to assume that the TTL of a packet from a source host inside the university network will be closer to a power of two than a packet from a source host outside the network, since the packet coming from outside will likely travel through more routers. Since packets only contain the TTL for the source address, we need to examine bidirectional flows to exploit TTL values. For example, a common type of flow is a web download from a Linux server with a TTL of 64 to a Windows PC with a TTL of 128. If data packets from the server have a TTL of 61 and acknowledgments from the PC have a TTL of 112 by the time they reach the monitor, then we may conclude that the server is 3 hops away from the monitor, and the PC is 16 hops away. In this case, one can infer that the server is inside the university. By examining a few bidirectional flows in this way, one can determine if the initial assumption was correct.

Using TTL values has some drawbacks since some hosts might not use the default TTL, and some operating systems use a starting TTL of 60. There are many large unidirectional UDP flows, and bidirectional TCP flows in which only one direction of the flow passes through the monitor. For these flows we will only know the TTL of the sender and not the receiver. Because of these flows and the uncertain nature of TTLs, it does not seem feasible to determine flow orientation from TTLs alone, although using them as a check on the output of the algorithm is appropriate. We check the results of our algorithms against TTL values in Section 4.4.

4.2 Overlap Algorithm

We say that two packets in the trace file *overlap* if one packet arrives at the monitor before the end of the transmission of the previous packet. Suppose the first packet in the trace file, call it P_1 , has a timestamp of t_1 and the second packet, P_2 , has timestamp t_2 . To see if the two packets overlap, we need to first calculate the transmission duration of the first packet as seen by the monitor. We call this the Nominal Interpacket Time (NIT), because if two packets traveling in the same direction arrive, back-to-back the NIT is the time from the head of the first packet to the head of the second packet. The NIT, which we denote by τ , is the size of the packet divided by the capacity of the link. For example, with a 155 Mbps OC3 link and a 1500 byte packet the NIT is:

$$\tau_{OC3} = \frac{12000 \text{ bits}}{155 \text{ Mbps}} = 0.000077 \text{ sec} = 77 \mu\text{s}.$$

In general, we say that packets P_1 and P_2 overlap if the following inequality holds:

$$t_1 + \tau > t_2. \tag{4.1}$$

If two packets overlap, then as a consequence they must have necessarily been traveling in opposite directions on the link so as not to interfere with each other. Upon discovering overlaps in the trace file, one might ask whether there could be another possible explanation. In particular, the overlaps could be due to the parallel transmission of packets on different wavelength channels, as is done in wavelength division multiplexing (WDM). If this were the case, we would expect many more overlaps than actually occur. In fact, the links used in the NLANR traces are

not utilizing WDM and the overlaps are due solely to packets traveling in opposite directions.

As in the bootstrap algorithm, we initially posit that the seed host is inside the university network. If we choose this seed host at random, then there is no assurance that the host's packets form any overlaps at all. We have found that a good way to initialize the algorithm is to process the trace file until 100 overlaps have occurred. Out of these overlaps, we choose the host that appears most often. We do not know a priori whether this seed host is inside the university network (it does not matter whether we correctly label the host as such). All the labels can be reversed after the completion of the algorithm if our initial assumption was wrong. However, we can make an educated guess about the location of the seed host based on the TTL, as described in the previous section.

The goal of the overlap algorithm is to label as many hosts as possible as either local (L) or remote (R). Starting from the beginning of the trace file and reading sequentially, we compare every two neighboring packets. If they overlap, then we try to label the four hosts (possibly fewer than 4 *distinct* hosts) in the two packets. If one or more hosts out of the four are already labeled, then we can orient the unlabeled hosts in the group. To illustrate this, call the four hosts involved in an overlap A, B, C and D , with A as the source address and B as the destination address in the first packet, and C and D as the source and destination in the second packet, respectively. If A is labeled L and no other packets are initially labeled, then the algorithm proceeds in labeling B as R . Since the orientation of the second packet is the opposite of the first, we label C as R and D as L , see Figure 4.2.

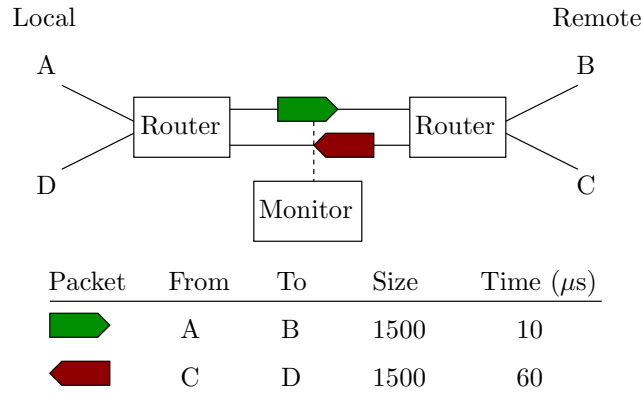


Figure 4.2: We can determine if two packets overlap from their timestamps, sizes and the link capacity. If two packets overlap, then they must have been traveling in opposite directions.

4.2.1 Practical Considerations

There are uncertainties in the timestamps and packet sizes that lead us to make a slight modification of the overlap algorithm. As noted in the introduction, the trace file includes a timestamp of the packet arrival that has microsecond precision, but not necessarily microsecond accuracy. Although the individual timestamps are likely very accurate, we have found the timestamps for one direction of the link can fail to be perfectly synchronized with the other direction. Because the monitor has different clocks for each direction. In the worst cases, the timestamps can be desynchronized by as much as one second. In these cases, by examining bidirectional flows, it appears that the timestamps have been corrupted in the trace file. The overlap algorithm does not work correctly for these traces, although the bootstrap algorithm does.

As stated above, the NIT for an OC3 link and a 1500 byte packet is $77 \mu s$, while

for a 40 byte acknowledgment the NIT is $2 \mu s$. For an OC12 link, the corresponding NITs are $19 \mu s$ and $0.5 \mu s$. We need to take into account the fact that the smallest NITs are on the order of the timestamp precision, and given the many different kinds of timestamp errors that occur we are uncertain if the timestamp accuracy is the same as the timestamp precision. Therefore, the algorithm needs to be modified to avoid using overlaps involving small packets. Since we are sure of the packet size and the link capacity, but not the timestamp, the modification should be independent of the packet size and link capacity. To that end, we introduce a minimum overlap threshold, t_* , that represents how long two packets must overlap before we use them to determine flow orientation. The overlap condition given in Equation 4.1 now becomes:

$$t_1 + \tau - t_* > t_2 \tag{4.2}$$

In practice, we choose t_* to be $10 \mu s$ for OC3 and OC12 links, and $2 \mu s$ for OC48 links. This choice is a balance between the need to find as many overlaps as possible and the need to avoid spurious overlaps. We are not able to detect spurious overlaps directly, but spurious overlaps will lead to inconsistencies in the way hosts are labeled, as described in Section 4.1. Increasing t_* decreases the number of the inconsistencies, but also decreases the number of the overlaps. The inconsistencies are handled in the same manner as the bootstrap algorithm.

Before a packet is actually transmitted at the physical layer extra bytes are added to the packet in the link layer. The overhead in the link layer is due to frame or cell headers that are added on top of IP and TCP headers. The exact nature of

these headers are not important for our purposes, but note that in the traces we use the frames were either Packet-over-SONET (PoS) or ATM-AAL5. These extra bytes are not accounted for in the calculation of the NIT, and the inclusion of the extra bytes decreases the bandwidth available to the network layer and increases the value of the NIT, in effect increasing the potential for more overlaps. For example, with ATM-AAL5 frames the overhead is approximately 15%, increasing the NIT of an OC3 link from $77 \mu\text{s}$ to $89 \mu\text{s}$.

Cavanaugh [9] found that most of the protocol overhead was due to ATM-AAL5. We investigated protocol overhead by computing the instantaneous band-

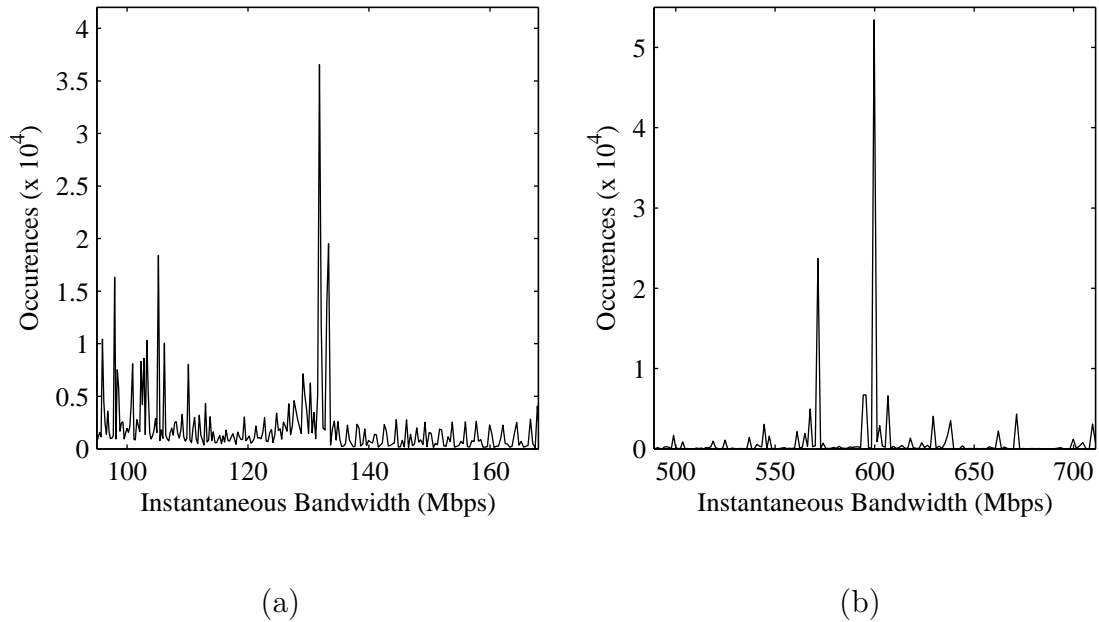


Figure 4.3: (a) The instantaneous bandwidth at the COS monitoring point is about 132 Mbps, for an overhead of 15%. ATM-AAL5 frames are used on this OC3 link. (b) The instantaneous bandwidth at the FRG monitoring point is about 600 Mbps, for an overhead of 3.5%. PoS frames are used on this OC12 link.

width for each packet, that is, packet size divided by interpacket time. By computing a histogram we are able to discern the most common value for the instantaneous bandwidth, which coincides with the bandwidth available to IP packets at the network layer of the monitored link, see Figure 4.3. We found that PoS frames are far more efficient than ATM-AAL5, with an overhead of only 3.5%. By using the instantaneous bandwidth to compute the NIT, instead of the total capacity of the link, we discover more overlaps, and as a consequence the overlap algorithm labels more hosts.

4.2.2 Algorithm Performance

As the algorithm progresses and the list of labeled hosts grows, we have a greater likelihood of labeling even more hosts, since with each new overlap there is a greater probability of having at least one of the four hosts already labeled. Hosts that are only active at the very beginning of the trace have a lesser likelihood of being labeled since the list is not as full. This can be addressed by running the algorithm a second time from the beginning while using the list of labeled hosts accumulated from the first run as a starting point. In fact, it may be even more advantageous to process the trace file in reverse chronological order the second time. The algorithm can be run multiple times until the list saturates and stabilizes.

We find that running the algorithm once is enough. The list of labeled hosts saturates fairly quickly once the hosts involved in the largest flows are labeled, since these flows have the most packets and, therefore, the most potential overlaps.

The key to optimizing the performance of the algorithm is the initialization step as described above. Choosing the host with the most overlaps among the first 100 overlaps processed allows the number of labeled hosts to grow rapidly.

An interesting point about the performance of the algorithm is that it works better on larger trace files. Since all traces files are about 90 seconds long, the larger the trace file (in terms of bytes), the more congested the traffic is. This results in packets being spaced closer together, and hence more overlaps.

4.3 Combined Algorithm

The result of the overlap algorithm is quite different from the bootstrap algorithm. The final list is not a single connected component of the IP connection graph. Instead it spans multiple components, but the list does not label every host in each component it covers. This suggests a way to combine the algorithms for maximum performance.

By running the overlap algorithm first, we orient flows from different connected components, thus orienting the components relative to each other. Second, we run the bootstrap algorithm to fill out each component reached by the overlap algorithm. Note that the relationship between the algorithms is non-commutative; running the bootstrap first followed by the overlap would give different results.

The algorithms perform differently on traces from different monitors. Typically, the bootstrap algorithm performs better on traces from OC12 links, and the overlap algorithm does better on OC3 links. OC12 links handle traffic from larger

networks like Merit and Colorado State, and therefore there are more flows and more distinct hosts in the trace files. This means the bootstrap algorithm has a larger, more connected IP connection graph with which to work. The overlap algorithm does not work as well on OC12 traces because we require that packets overlap by at least $10 \mu s$, therefore packets involved in an overlap must be at least 778 bytes. In practice, most overlaps are between two 1500 byte packets. We can relax this requirement by reducing the parameter t_* , but we have found that the combined algorithm still works well with $t_* = 10 \mu s$. Conversely, the overlap algorithm works better on traces from OC3 links since these links have less capacity and are more likely to be saturated, leading to more overlaps.

Despite the differences in the performance of the individual algorithms, the combined algorithm performs well on almost all traces. The most obvious measure of the success of the algorithms is the number of hosts labeled. An alternative measure is the total number of bytes transferred by the labeled hosts. This is perhaps a better measure, since most of the hosts that remain unlabeled are involved in very few flows and do not transfer many bytes. See Table 4.1 for a summary of the results.

4.4 Validation Tests

We can not absolutely confirm that the results of the combined overlap/bootstrap algorithm are correct. However, we are confident that the results are highly accurate. In this section, we describe three methods of validating the combined algorithm, multiple traces, time to live, and non-interspersed traffic.

Monitor	MRA	TXG	COS	BUF	BWY
Overlap	14 (94)	23 (99)	14 (98)	69 (98)	40 (98)
Bootstrap	64 (60)	67 (25)	90 (81)	20 (20)	69 (40)
Combined	90 (99.1)	92 (99.9)	93 (99.9)	93 (98.5)	99 (99.9)

Table 4.1: The first number in each column denotes the percentage of hosts oriented by the algorithm in that row. The number in parentheses denotes the percentage of bytes oriented. MRA and TXG are OC12 links, the rest are OC3.

4.4.1 Time To Live

As stated in Section 4.1, it is possible to use TTL values to infer flow orientation. We choose not to do so because of the drawbacks associated with it. However, a suitable application of TTL values is to use them as part of a validation test for the combined algorithm. We consider only bidirectional flows, so that we see both the sender and receiver TTL. We also require that both the sender and receiver TTL be greater than 16. We impose this condition since most operating systems start with a TTL greater than or equal to 32, and therefore, TTL less than or equal to 16 are unusually low. It is possible, but unlikely, for the TTL to change over the course of a flow, because the route taken by packets may change.

OS	Windows	Linux	OpenBSD*	Solaris	Cisco	AIX
TTL	32 or 128	64	64	64 or 255	60 or 255	60

Table 4.2: These values depend on the version of the operating system see [32]. (*)

FreeBSD and Mac OS X should also have a default initial TTL of 64.

To determine if the TTL values are consistent with the orientation of a flow as given by the combined algorithm, we must estimate the most likely initial TTL value for the sender and receiver. See Table 4.2 for a list of default initial TTL values for several common operating systems. In light these values, we estimate the initial TTL as the next power of two greater than the TTL observed at the monitor. Figure 4.4 shows the distribution of observed TTL values across all monitoring points (except for MRA) for the month of December 2004.

We use the estimate of initial TTL to determine the number of hops from the monitor to both the sender and receiver. Consider a bidirectional flow for which the combined algorithm has labeled one host as remote (R) and the other as local (L). Let T_R^i and T_L^i be the estimates of the initial TTL values for the remote and local

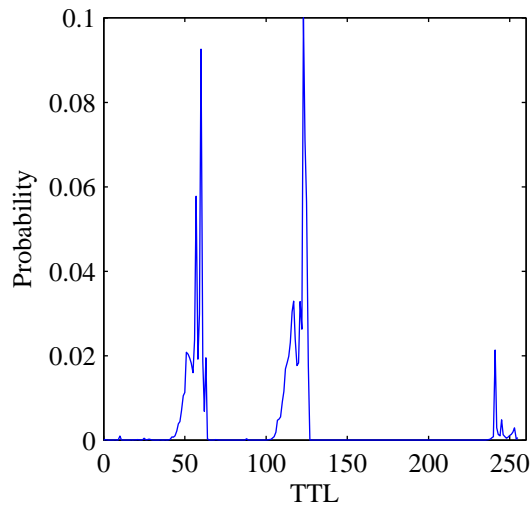


Figure 4.4: The distribution of observed TTL values across all monitoring points. The spike at 241 was caused by a denial of service attack at the AMP monitor on December 10, 2004.

hosts, respectively. Also, let T_R^o and T_L^o be the TTL values observed by the monitor.

Set $H_R = T_R^i - T_R^o$ and $H_L = T_L^i - T_L^o$. Define Γ and Δ as:

$$\Gamma = H_R + H_L \quad (4.3)$$

$$\Delta = H_R - H_L. \quad (4.4)$$

Γ is an estimate of the path length from sender to receiver in hops. When the algorithm has worked correctly in labeling hosts as either remote or local, Δ should be greater than zero for the vast majority of hosts, since the remote host will be further from the monitor in most cases. Figures 4.5 and 4.6 show the distributions of Γ and Δ for all monitoring points.

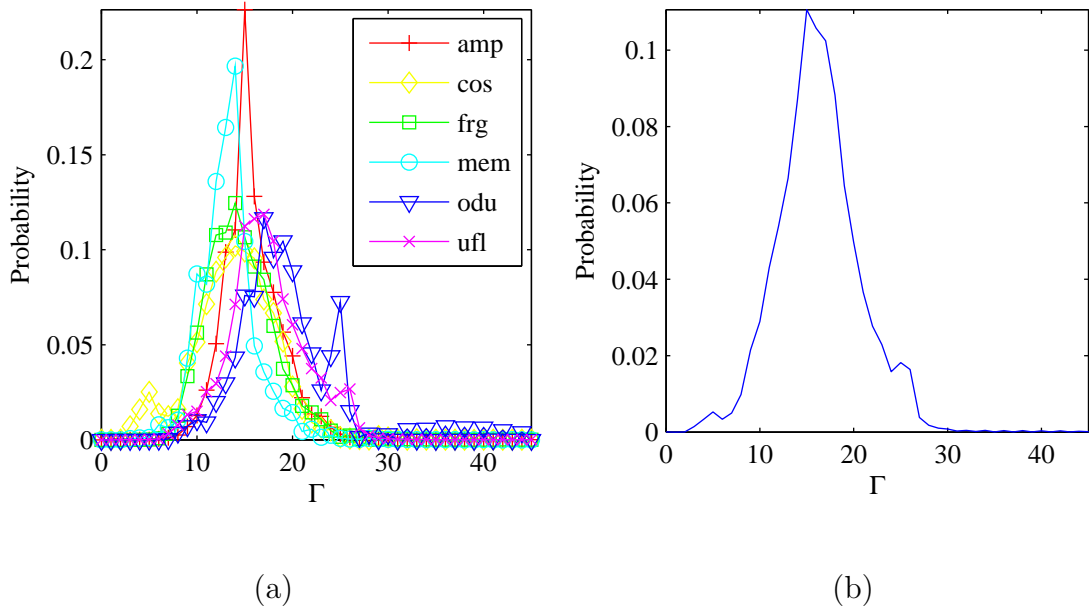


Figure 4.5: (a) Individual Γ distributions for six monitoring points. (b) The distribution of observed Γ values across all monitoring points. Γ is less than or equal to 30 for 99.5% of flows.

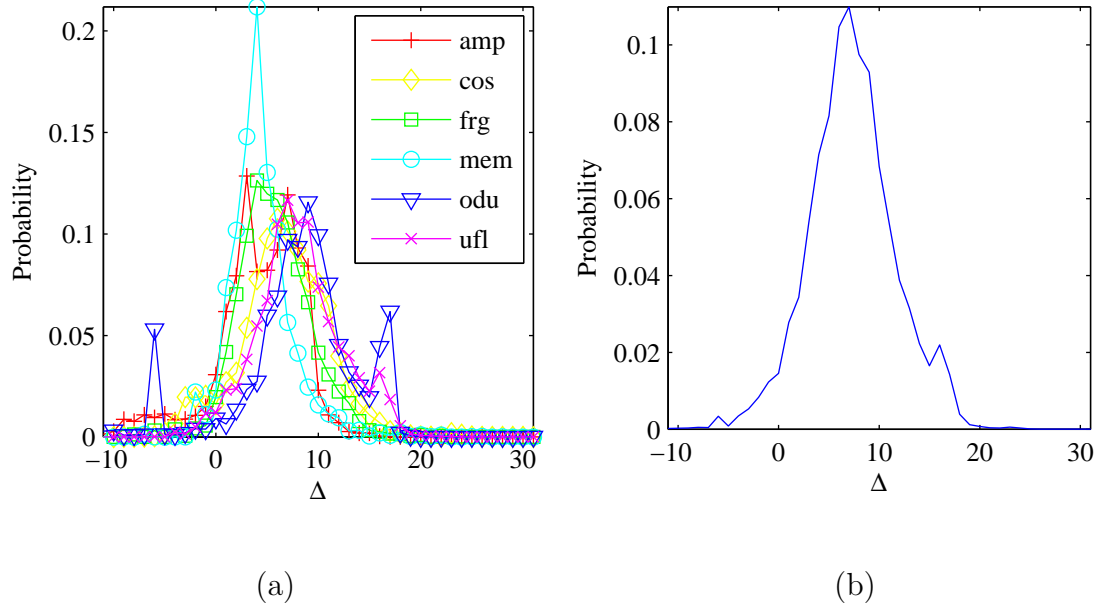


Figure 4.6: (a) Individual Δ distributions for six monitoring points. (b) The distribution of observed Δ values across all monitoring points. Only 3% of flows have a negative value of Δ .

4.4.2 Multiple Traces

The 90 second NLANR traces are taken eight times per day; most of the monitors begin recording traffic at the same time. This introduces the possibility that the same flow will be observed at two different monitoring points. For example, such a flow could involve one host inside the Colorado State network and another inside the Texas GigaPOP network. To identify flows common to two trace files, we make summaries of all flows in each trace and check for flows that have the same source and address ports, similar start times and durations, and very similar number of packets and bytes transferred. The reason we do not require the number of packets and bytes to be identical is because we want to allow for the possibility

that packets might be dropped between the monitors, and the possibility that one trace starts or ends slightly out of synchronization with the other. Note that IP addresses will be of no help, since they are renumbered differently in each trace.

Once we have identified the flows common to two trace files, we run the combined overlap/bootstrap algorithm on both traces. If the labels for the common flows are the same in both traces, then exactly one of the traces is mislabeled. This is because what is local relative to one monitor is remote to the other. If we check the TTL values of one trace to ensure it is labeled properly, as we describe in the previous section, then we can make a table to check the consistency of labels for the other traces. Table 4.3 shows a case where the results of the algorithm have been checked for the MRA trace. Boxes marked with “C” mean that the host labels for the traces in that row and column are consistent, “I” means they are inconsistent. We may conclude that the labels for the TXG and COS traces are also correct, since their host labels are compatible with those from the MRA trace. It follows that the labels in the BWY and BUF traces need to be reversed.

This method of cross checking labels from flows that appear in different traces at least ensures the consistency of the labels. We can definitively ascertain the orientation of the flows that pass through two monitoring points. The accuracy of all other flow orientations can be based upon this.

	MRA	TXG	BUF	BWY	COS
MRA	-				
TXG	C	-			
BUF	I	I	-		
BWY	I	I	C	-	
COS	C	C	I	I	-

Table 4.3: C denotes that the traces in that row and column are labeled consistently, I means they are labeled inconsistently. The orientation for the BUF and BWY traces must be reversed in order for all of the labels to be consistent.

4.4.3 Non-interspersed Traffic

A fortuitous side-effect of the timestamp errors discussed in Section 3.2 is that occasionally instead of having repeated packets from one direction, the traffic from the two directions of the link is not interspersed at all in the trace file. This allows us to definitively ascertain the orientation of every flow. Data from the FRG monitoring point taken on December 20, 2004 exhibited non-interspersed traffic, see Figure 4.7.

After eliminating the repeated packets, we break the packets into two sets corresponding to the two directions of the link. Let set A be packets number 1 to 2266504, and define set B as packets number 2266505 to 4518217. There are no overlaps within set A or set B , confirming that each set contains unidirectional traffic. Since there are many more hosts outside the university network than inside it, we expect there to be more unique addresses on the remote side of the monitor.

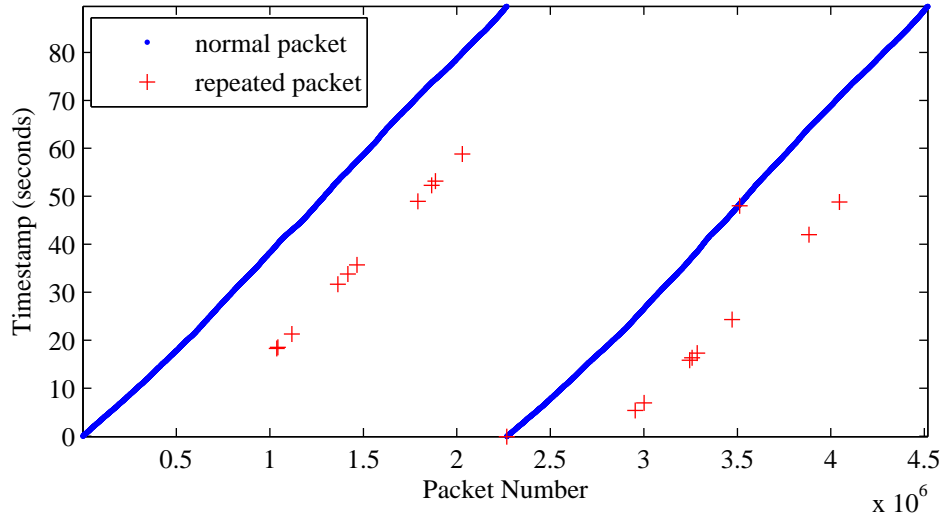


Figure 4.7: This plot shows that packets from the two direction of the link are not interspersed. There are also 19 repeated packets.

Packets in set A have 22245 unique source addresses and 9590 unique destination addresses, whereas packets in set B have 6469 unique source addresses and 27347 unique destination addresses. The imbalance of addresses leads us to conclude that set A contains inbound traffic, and set B contains outbound traffic.

Let S_i be the set of unique source addresses in set A , and let D_i be the set of unique destination addresses in set A . Also, let S_o and D_o be the unique source and destination addresses, respectively, in set B . Let $R = S_i \cup D_o$ and $L = S_o \cup D_i$. Set R should contain only addresses on the remote side of the monitor (outside the university network), set L should contain only addresses on the local side, and the intersection $R \cap L$ should be empty. However, there is one address in $R \cap L$. All flows involving this address use Protocol Independent Multicast (PIM), an alternative transport layer protocol to TCP and UDP. Since these flows are

multicast the address could actually represent many different hosts. Therefore, this does not contradict our assumptions about the network and the bipartite nature of the IP connection graph.

To use the combined overlap/bootstrap algorithm on this particular trace we first had to intersperse the packets by sorting them according to their timestamps. The results of the combined algorithm of this particular trace were rather atypical, labeling only 93.8% of the hosts and 95.2% of the bytes. This leads us to believe that there is something unusual about this trace file that hinders the performance of the combined algorithm. The monitor started recording traffic from the outbound direction of the link 91.2 milliseconds before the inbound direction, perhaps leading to errors in the overlap algorithm. Also, the presence of the single address using PIM should cause many inconsistencies in the bootstrap algorithm, which do not occur in other traces.

In spite of the poorer than average performance of the combined algorithm on this trace, we found that the results largely agree with the sets L and R . Let L' and R' equal the set of hosts labeled as local and remote, respectively, by the combined algorithm. The number of incorrectly labeled hosts is $|R \cap L'| + |L \cap R'| = 4895$, leaving only 34759 correctly labeled hosts out of 42276 total hosts. This may seem like a rather negative result at first, but upon consideration of the size of the flows of incorrectly labeled hosts, the results are still quite good. The incorrectly labeled hosts accounted for only 4681100 bytes of traffic, meaning that of bytes labeled by the combined algorithm, only 0.16% of them were wrongly labeled in this particular trace.

In conclusion we note that orienting flows relative to each other is potentially useful when using real world data to model network traffic. For example, one would expect that two flows will only affect each other if the flows are oriented in the same way. In this case, the packets are more likely to pass through the same routers, and thus, they mutually increase congestion along the shared part of their paths. The algorithms can be used in conjunction with round-trip time estimates and congestion sharing algorithms as a way of characterizing network traffic and the overall flow of data in and out of a local network.

The methods discussed in this chapter could have broader impact. For instance, consider the related problem of detecting flows that share common links. Katabi et al. [25, 26] discuss a method to detect flows that pass through shared bottleneck links. Knowledge of flow orientation could also be useful in network tomography [42, 47]. If one wanted to use the publicly available traces from NLANR to test hypotheses about network topology and traffic characteristics, then determining flow orientation would be an important first step.

Chapter 5

Measurement of Network State

In this chapter we discuss methods of inferring flow characteristics and determining network state. The congestion window and round-trip time are the most important rate-controlling variables. We present two methods of estimating these variables: the clustering algorithm and the frequency algorithm. Interpacket times play a key role in these algorithms, and can also be used to infer the amount of queuing delay experienced by each flow. The round-trip times and congestion windows of all bulk TCP flows may be considered as the state of the network at any given time. Considered as a whole, these measurements may allow us to detect and predict congestion.

5.1 Interpacket Time Distribution

The RTT is the time, measured at the sender, between a data packet and its acknowledgment. However, our vantage point at the monitor can be anywhere along the path between the sender and receiver, so we will not be able to use this definition, see Figure 5.1. The interpacket time is defined as the time from the beginning of one data packet to the beginning of the next data packet in the same connection. Due to the dynamics of TCP and the nature of networks, interpacket times vary greatly. In fact, it is common for interpacket times for a single connection

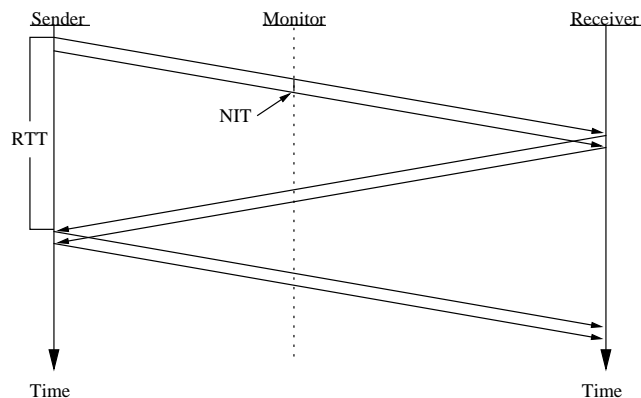


Figure 5.1: Time line diagram showing the RTT, NIT and the position of the monitor.

to vary over three or more orders of magnitude. Therefore, we mainly work with the logarithm of the interpacket time.

To illustrate why interpacket times span several orders of magnitude, consider two successive 1500 byte packets from the same connection traversing an OC3 link. If the packets arrive at the router adjacent to the monitored link without any cross traffic packets intervening, the router will send them out on the link one after the other. In this case the two packets will have as small an interpacket time as possible (for 1500 byte packets and an OC3 link). Define this interpacket time to be the Nominal Interpacket Time (NIT). For an OC3 link the NIT is:

$$\frac{12000 \text{ bits/packet}}{155 \text{ Mbps}} = 0.077 \text{ ms.}$$

At the opposite end of the spectrum, two data packets will have a long interpacket time if they are in different windows, these data packets can have an interpacket time that is about one RTT. Common RTTs can be anywhere from a few milliseconds to a few hundred milliseconds, depending on the distance from the sender to the

receiver and how much congestion there is in the network.

Katabi et al. [25, 26] noted that a bottleneck link along a connection will impose a certain structure on the distribution of the interpacket times. This structure can be used to estimate the NIT. For example, consider a flow that passes through an uncongested link to a slightly congested OC3 link, whereupon the flow is then measured. Suppose that most packets from this flow go through the OC3 link back-to-back and thus be spaced by one NIT of the OC3 link. But sometimes cross traffic will arrive between two packets from the flow. If the cross traffic consists of, say, one 40 byte packet, then there would be little difference in the interpacket time. However, if the cross traffic consisted of multiple 1500 byte packets, then the interpacket time would be that multiple of the NIT. In this situation we would expect the distribution of interpacket times to have a mode at the NIT and a train of decreasing spikes at integer multiples of the NIT. Other more complicated scenarios are possible and are discussed fully in [26]. For our purposes we just need the fact that the interpacket time distribution will have spikes or spike trains around the NIT of the congested links in the path.

Now consider a histogram of the logarithm of the interpacket times. For many flows, the standard operating regime of TCP is to have a flight of *cwnd* data packets followed by a pause roughly equal to one RTT before another flight of packets. Therefore, we should expect the distribution of interpacket times to have its mode at the NIT of the most congested link along the path and another peak approximately at the average RTT for the connection. However, such a histogram would be poorly scaled. The peak at the NIT would dwarf the peak at the RTT since many more

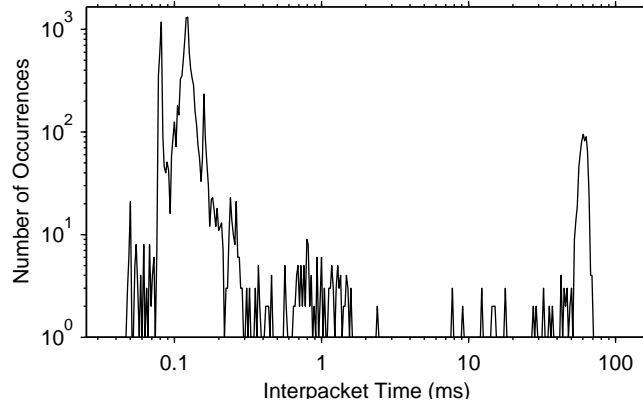


Figure 5.2: Log-log scaled histogram of interpacket times for a KaZaA file-sharing flow from Jan. 9, 2003 taken at the BUF monitor. The two peaks to the left are at the NITs for OC3 and 100 Mbps Ethernet. The peak at about 60 ms is close to the RTT.

packets will be separated by the NIT than the RTT. To visualize the statistics of the interpacket times, we plot the logarithm of the probability (or in this case the number of occurrences, leaving the histogram unnormalized) on the vertical axis, creating a log-log histogram of interpacket times. An example of such a plot is shown in Figure 5.2.

If the logarithm interpacket time histogram has the bimodal form as described, then it is possible to determine a cutoff value between the NIT and the RTT. For example, in Figure 5.2, one might choose 4 ms. We then group packets into flights; if two consecutive packets have an interpacket time that is less than the cutoff value, then they are part of the same flight, otherwise they are in different flights. By computing flight sizes for an entire connection we will have approximately reconstructed the dynamics of the congestion window, as shown in Figure 5.3. Unfortunately, very

few TCP flows have an interpacket time distribution with a bimodal structure as this example. In Section 5.2 we discuss more sophisticated algorithms to estimate the RTT and *cwnd*.

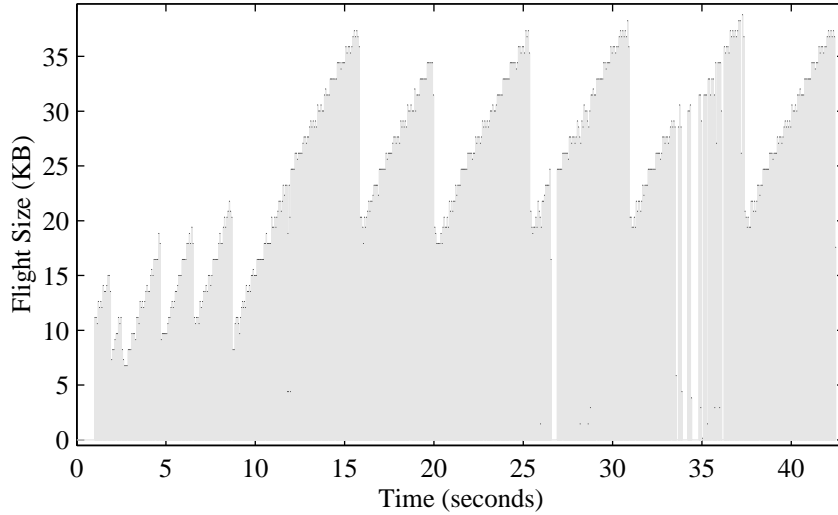


Figure 5.3: Reconstructed flight sizes of the flow in Figure 5.2. The gray background is added for clarity. The flight sizes are consistent with the dynamics of *cwnd*. The gaps around 35 seconds are due to the receiver’s buffer being zero for a short period.

5.1.1 Bandwidth Delay Product

This analysis is predicated upon the common modeling assumption that the RTT is much larger than the time it takes to transmit a full congestion window’s worth of packets. See Barakat [6] for an analysis of the dependence between the RTT and congestion window when this assumption does not hold. Another way to state this assumption is that the most congested link along the path should have a NIT that is significantly smaller than the RTT, otherwise it will be impossible to discriminate between the two. When this assumption holds it is usually due to the

fact that the receiver's buffer is limited to 64 KB or less.

One way to measure how efficiently a TCP flow uses available bandwidth is to compare the receiver's buffer to bandwidth delay product (BDP) [40]. The bandwidth delay product is defined as the product of the capacity of the slowest link along the path and fixed delay in the absence of any queuing. For example, if 100 Mbps Ethernet is the slowest link encountered by a flow that has a delay of 50 ms, then that flow has a BDP of 625000 bytes. This is nearly ten times the 64 KB limit on the receiver's buffer. The BDP in this example is not that large; for some TCP bulk transfers, the BDP can exceed 10 MB. Weigle and Feng [50] found that most TCP connections have a bandwidth-delay product that is much larger than the receiver's buffer size.

It is often recommended that the receiver's buffer be set to the BDP to achieve maximum throughput; this has necessitated some clever ways of overcoming the 64 KB limit. One way to achieve this is to split the file into several chunks of equal size and then simultaneously transfer the chunks over different ports. An interesting twist on this is that if an exact copy of a file exists on multiple hosts then different chunks of the file may be transferred in parallel from all hosts at once; the file sharing application BitTorrent uses this method. The TCP window scale option defined in RFC 1323 [19], allows the receiver to advertise a window larger than 64 KB, but the sending host will also have to accommodate this change by increasing its send buffer to twice the size of the receiver's buffer. Setting the receiver's buffer to the BDP does not guarantee that the throughput will ever attain this maximal level, most of the time cross traffic will limit the available bandwidth. Determining

the optimal size of TCP buffers in high performance grid computing is known as dynamic right-sizing [45, 50]. Although the window scale option is used by some of the monitored flows, it is likely that very few are using dynamic right-sizing.

5.1.2 Cross Traffic, Queuing, and Delayed ACKs

Even though many of the flows we study have a BDP that is much greater than 64 KB, it is still uncommon to see a flow with an interpacket time distribution that has a bimodal structure as distinct as Figure 5.2. This is due to cross traffic, queuing, and delayed ACKs, as well as the sequence of links along the path. These influences can either diffuse the distribution of interpacket times, or concentrate its mass around certain characteristic values.

Rather than simply referring to bottleneck links we will instead use the terms narrow link and tight link [11]. The link with the lowest capacity along a path is known as the *narrow* link. Let C be the capacity of a link, and let $U(t)$ be the fraction of that capacity that is utilized at time t . Then $C(1 - U(t))$ is the available bandwidth. The *tight* link is defined as the link along the path with the lowest available bandwidth. The narrow link and the tight link are not necessarily the same, but they will coincide for many flows.

The narrow link for bulk TCP flows is usually 10 or 100 Mbps Ethernet, given the ubiquity of Ethernet in LANs. T3 links (45 Mbps) and fractional T3 links are also common narrow links, and cable modems (usually 3 Mbps) are increasingly common as well. Even though the narrow link along the path is usually considered

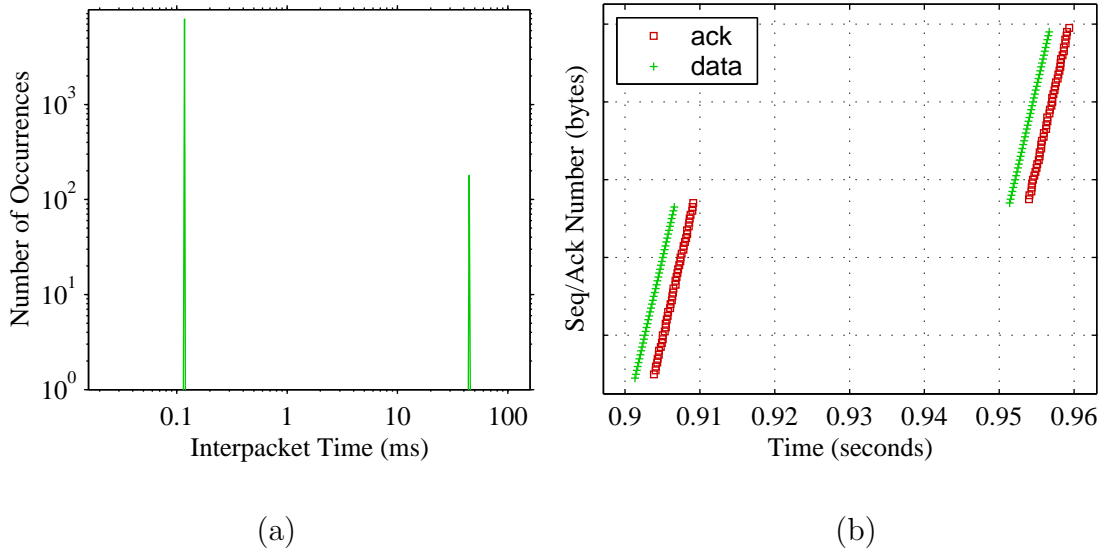


Figure 5.4: Interpacket time histogram (a) and seq-ack plot (b) under ideal conditions. In (a) the mode to the left is at 0.12 ms, the NIT of 100 Mbps Ethernet, and the mode to the right is at 45 ms, just less than the RTT. In (b) note that the available bandwidth is only about 10% utilized.

as the bottleneck, congestion, queuing, and dropped packets can occur anywhere along the network path [4]. However, the monitored links are rarely the source of congestion for the flows we study; they are rarely more than 66% utilized over any 1 millisecond span.

Consider again the example of a flow with a 100 Mbps narrow link and a RTT of 50 ms, call this flow *A*. The BDP for this flow is 625000 bytes, so if there was no cross traffic, then we would expect the available bandwidth to be about one tenth utilized. Figure 5.4 shows what the histogram of interpacket times and the seq-ack plot of flow *A* would look like under these ideal circumstances.

Figure 5.5 shows cross traffic from flows *B* and *C* interspersed with flow *A* at

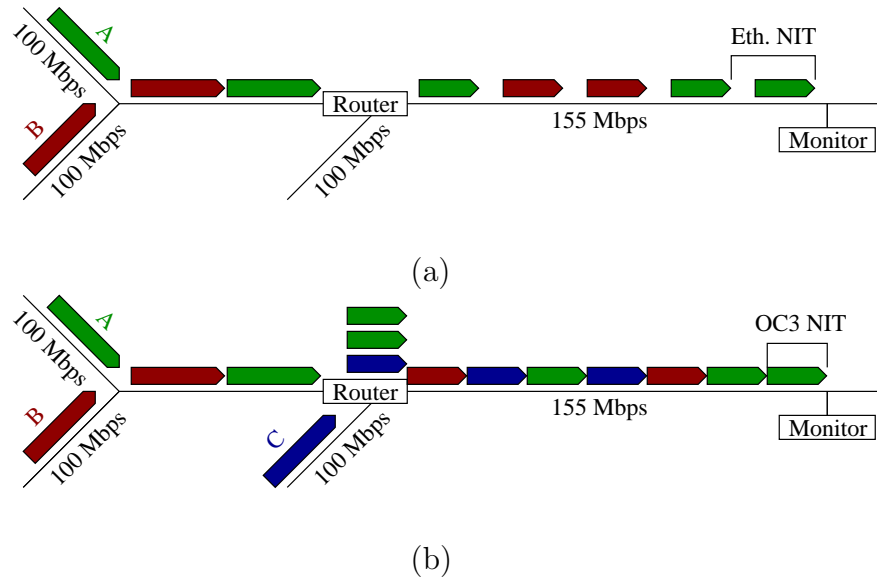


Figure 5.5: (a) When only one Ethernet link feeds into an OC3 link there is no queue and the pacing of packets stays fixed. (b) With traffic from two Ethernet links there is a queue at the router and the packets are now spaced at multiples of 0.077 ms.

two different points in the network. For the situation shown in Figure 5.5(a), where there is no queue at the router adjacent to the monitored OC3 link, the spacing between packets remains the same as it was on the Ethernet link. We illustrate this in the figure by drawing shorter packets on the OC3 link than on the Ethernet link. The packets are not actually shorter they just occupy less of the capacity on the OC3 link; about one third less in this example. The interpacket times on the OC3 remain the same as they were on the Ethernet link, multiples of 0.12 ms. Figure 5.5(b) shows another situation where additional cross traffic from host *C* creates a queue at the router. When there is a queue the capacity of the monitored link will be fully utilized and flow *A* will have interpacket times that are multiples of the

OC3 NIT, 0.077 ms.

If flow A experiences a mix of cross traffic, some from B alone, some from B and C together, then the resulting interpacket time histogram and seq-ack plot might look like Figure 5.6. This scenario explains the distribution of interpacket times seen in Figure 5.2, except that the RTT is greater in that case, and there is additional diffusion around the local modes due to cross traffic packets smaller than 1500 bytes. There are many other possible arrangements of links and cross traffic, but the point is that queuing not only increases interpacket times, it can decrease them as well. Also note that queuing need not increase the RTT, especially if the queuing occurs on a link with higher capacity than the narrow link.

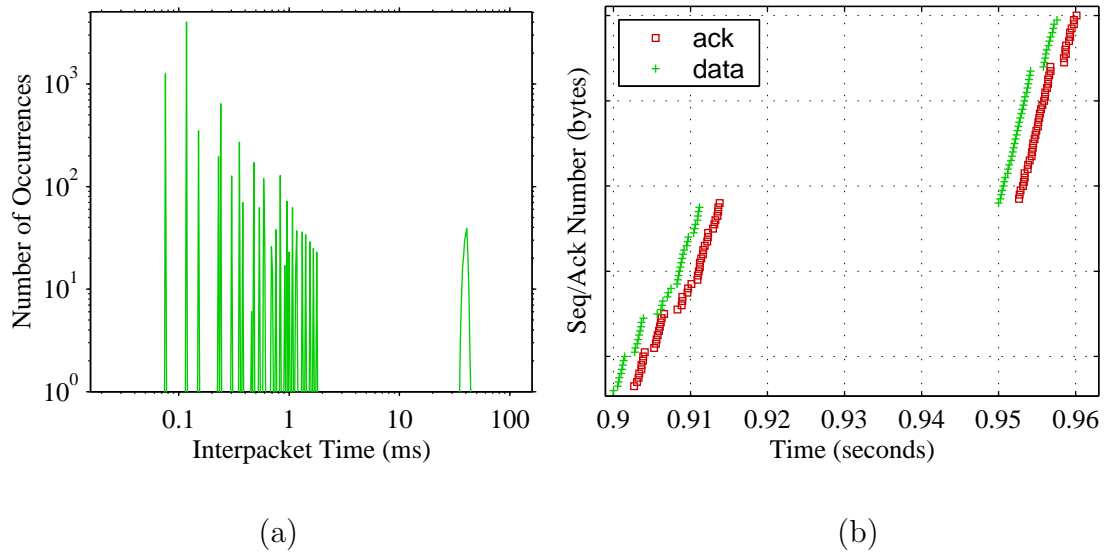


Figure 5.6: (a) Local modes in the interpacket time distribution are at multiples of 0.12 ms and 0.077 ms. Some interpacket times are now smaller than when there was no congestion. (b) The flights now last about 10 ms as compared to 5 ms in Figure 5.4(b), but the RTT has not appreciably increased due to queuing.

Most of the time the forward and backward paths traverse the same sequence of links, although that is not always the case. Therefore, it is possible for the monitor to see one direction of the traffic, but not the other. In a sample of 33000 bulk TCP flows from seven monitoring points taken over 30 days the monitor did not see ACKs for 20% of the flows. As such, we place more importance on the data packets than the ACKs when estimating the RTT and congestion window.

There is, however, much useful information in the acknowledgments and the time between ACKs, which we will refer to, as the interack time, so as to avoid confusion with the interpacket time. We mainly use interpacket and interack times, instead of instantaneous bandwidth, because the two quantities are directly comparable, and we are not expressly concerned with estimating the bandwidth of congested links. In the bulk TCP flows we are studying, over 70% of data packets are between 1420 and 1500 bytes and 94% of ACKs are between 40 and 52 bytes, so most interpacket and interack time distributions have modes within a certain set of values corresponding to the NITs of common links.

The capacity of links along a network path can vary greatly, and the interpacket time distribution depends heavily on sequence of link capacities. Consider the following example, for a given flow the path from sender to receiver might first pass through a 100 Mbps Ethernet link, followed by an OC3 link and several high capacity backbone links before traversing another OC3 link, and finally a 10 Mbps Ethernet link. The forward and backward paths might have very different levels of congestion, and the congestion could be on the local side, remote side, or both sides of the monitor. This highlights the importance of knowing flow orientation, as the

monitored link could be either OC3 link listed in the example. Knowledge of the flow orientation allows us to analyze the path from the sender to the monitor and the path from the monitor to the receiver.

Continuing with this example, suppose the monitor is on the OC3 link closest to the sender. Even though the narrow link between the sender and the monitor is a 100 Mbps Ethernet link, one would still expect the interpacket time distribution to closely mirror the interack distribution. Specifically, one would expect to see the mode of both distributions to be at the NIT of 10 Mbps Ethernet, since that is the tight link overall. This occurs because each ACK frees the sender to transmit one data packet, and the ACKs should arrive at the sender spaced apart by about the NIT of 10 Mbps Ethernet. The ACKs traverse many potentially congested links before they arrive at the sender, and as we have seen this can cause the interack time to increase or decrease, but the interack time should remain relatively stable from receiver to sender.

The use of delayed acknowledgments can change the interpacket time distribution dramatically. In this example, instead of having its mode at the 10 Mbps NIT the distribution will now have its mode at the 100 Mbps NIT. When using delayed ACKs the receiver acknowledges every other packet, and the sender transmits two packets as soon as an ACK is received. In this case, since the sender is close to the monitoring point, pairs of data packets are likely to arrive at the monitor back-to-back with an interpacket of the 100 Mbps NIT. Consequently, the interpacket times will alternate between the 100 Mbps NIT and the 10 Mbps NIT.

We have found delayed acknowledgments to be very common. While acknowl-

edging every other packet is the most common implementation, there are others where every third, fourth or fifth packet is acknowledged. Another common implementation acknowledges two out of every three packets. In a sample of 22000 bulk TCP transfers with at least 100 ACKs, 80% of the flows had an ACK to data packet ratio of 0.75 or less, indicating delayed ACKs. Only about 10% had an ACK to data packet ratio greater than 0.9.

Consider the following two flows that exhibit much of the same network path characteristics in the example described above. The first flow is a HTTP transfer from the COS monitor taken May 18, 2004. The second flow is connection on port 1450 from the FRG monitor taken May 19, 2004. Figure 5.7 shows the interpacket and interack time histograms, and Figure 5.8 shows a seq-ack plot from each flow. In both flows the narrow link between the sender and the monitor is 100 Mbps Ethernet and 10 Mbps Ethernet between the monitor and receiver, and both flows use delayed ACKs. The COS flow has a RTT of about 270 ms, whereas the FRG flow has a RTT of about 45 ms.

The distributions in Figure 5.7 look different, despite have similar underlying path characteristics, because the monitor is closer to the receiver in the COS flow, whereas the monitor is closer to the sender in FRG flow. The interack time distributions are partially obscured by the overlying interpacket time distributions, but this shows that the right half of the distributions match as one would expect due to the self-clocking nature of TCP. In Figure 5.7(a) the mode of the interack times is slightly greater than 2 ms, which is about twice the NIT of 10 Mbps Ethernet. This is due to delayed ACKs; since the monitor is closer to the receiver, we are more likely

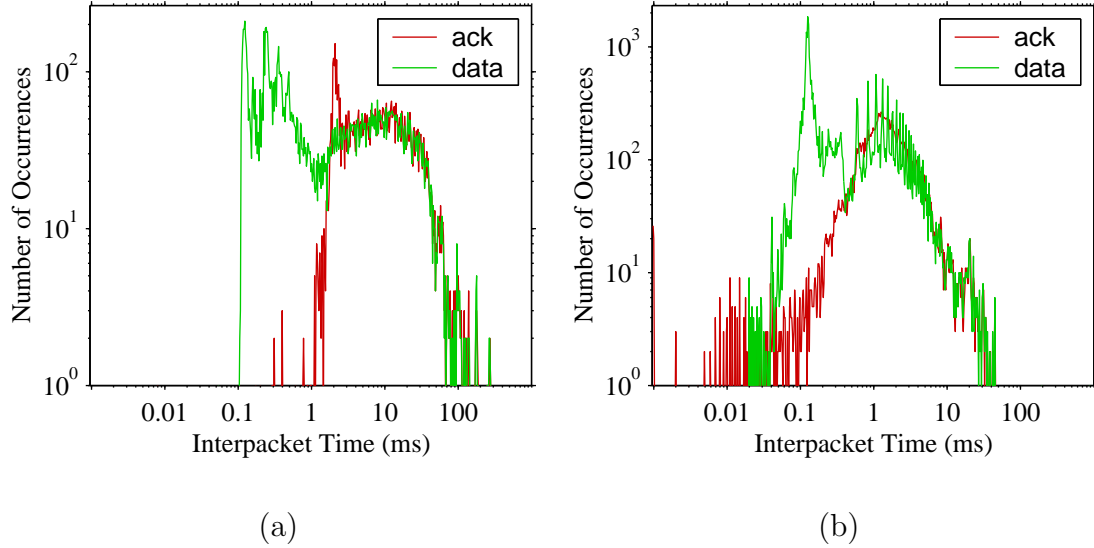


Figure 5.7: Interpacket and interack time histograms for a COS flow (a) and a FRG flow (b).

to see the ACKs spaced as the receiver initially sent them. In contrast, the FRG interack time distribution in Figure 5.7(b) is very diffuse, since the ACKs traverse a wide variety of congested links that compress or expand their spacing before they are measured.

The modes of the interpacket time distributions are both equal to the NIT of 100 Mbps Ethernet, 0.12 ms. The COS flow exhibits local modes at multiples of that NIT, but no interpacket times less than 0.12 ms, whereas the distribution for the FRG flow is symmetrically diffused in small region around 0.12 ms. This implies that most of the congestion between the sender and the COS monitor occurs on the 100 Mbps Ethernet link, while the congestion between the sender and the FRG monitor is partially due to faster links. The oscillations in the FRG interpacket time histogram around 1 ms have period 0.24 ms, which is twice the 100 Mbps NIT.

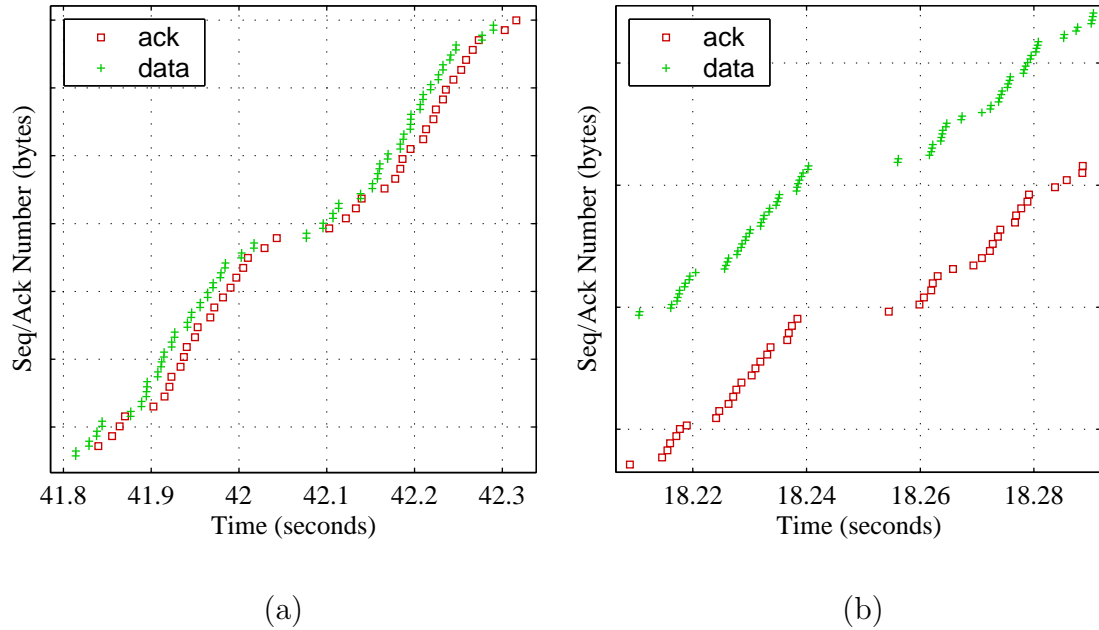


Figure 5.8: Seq-ack plots for the COS flow (a) and the FRG flow (b).

This is due to other flows on the Ethernet link between the sender and the monitor sending pairs of packets back-to-back, because of delayed ACKs. If the monitoring point for the FRG flow was closer to the receiver, then these oscillations might be damped, as is the case in the COS flow.

The position of the hosts in relation to the monitors is made clear by the seq-ack plots in Figure 5.8. Soon after a data packet from the COS flow is recorded at the monitor its corresponding ACK is recorded, indicating that the receiver is closer to the monitor. From this perspective the pacing of the data packets determines the pacing of the ACKs. The opposite is true of the FRG flow. Here the ACKs arrive at the monitor well after their corresponding data packets and the pacing of the ACKs determines the pacing of the next flight of packets.

In this section, we have described properties of interpacket times and interack

times. We must consider cross traffic, queuing, and delayed ACKs when inferring flow properties from the interpacket time distribution. In the next two sections, we use these properties to estimate the RTT and congestion window.

5.2 RTT and Congestion Window Estimation

RTT estimation and *cwnd* estimation are equivalent problems. One can estimate the RTT first and then estimate the sequence of congestion windows, or one can estimate the sequence of congestion windows first and then estimate the RTT. If one chooses to estimate the RTT first, then *cwnd* can be estimated by grouping packets into flights that are about one RTT in duration. We use flight and window somewhat interchangeably, since a window of packets are transmitted in a flight. If one estimates the sequence of *cwnd* values first, then the time from the beginning of one window of packets to the next is approximately one RTT. The first algorithm we developed, called the clustering algorithm, is a *cwnd*-first algorithm. Our second algorithm, called the frequency algorithm, is a RTT-first algorithm.

There are a variety of simple techniques to estimate the RTT, perhaps the simplest of which is to use the time from the SYN to the SYN/ACK during the three-way handshake at the set-up of the TCP connection. Another simple method is to use the time between the first two flights during slow start [21]. These are not very reliable methods since the RTT will change throughout the course of a flow, and these measurements are based on the beginning of the connections. We cannot use these techniques since most of the bulk TCP flows we are interested in

will already be in progress once the trace is started and continue after the trace stopped. Therefore, we will not see the connection set up with the initial SYN and slow start phase.

The alternate approach of grouping packet together is used by Zhang et al. [51], who assume that the number of packets per flight should follow certain patterns. They start with a pool of exponentially spaced candidate RTT values, which they use to partition the packets into groups. The best candidate RTT is the one that results in a sequence of groups that best fits standard TCP window behavior. Jaiswal et al. [20] use a finite state machine that “replicates” the TCP sender’s state. The state of the machine is updated based on observed ACKs and depends on TCP flavor. The algorithms we describe require only one direction of a flow and assume very little about the underlying traffic. Our methods have the flexibility to deal with any flavor of TCP, varying RTTs, and nonstandard TCP connections.

5.3 The Clustering Algorithm

For many large flows there is not always a clear distinction between the NIT and the RTT, hence the interpacket time histogram looks like Figure 5.9 rather than Figure 5.2. In this case the only characteristic time scale that is discernible is the NIT. Even though this seems like a setback we can use it to our advantage. Our clustering strategy is to identify long stretches of packets with interpacket times that are approximately equal to the NIT. The time between these stretches will give an estimate of the RTT. The subtlety of the algorithm is determining what

constitutes a “long stretch of packets,” and what is meant by “approximately equal to the NIT”.

To address the latter issue, note that distribution around the NIT has some width due to cross traffic. There can even be two dominant NIT values corresponding to links with different capacities as in Figure 5.2. If we simply take the NIT to be the mode of the distribution and look for stretches of packets with interpacket times less than or equal to that NIT, we would overlook many potential flights in which there is an interpacket time slightly larger than our chosen NIT. To ensure that we do not undershoot the range of NIT values, we need to develop a simple way of bounding that range from above.

The structure of the interpacket time distribution around the NIT will normally consist of equally spaced local modes. Thus, one might choose an integer

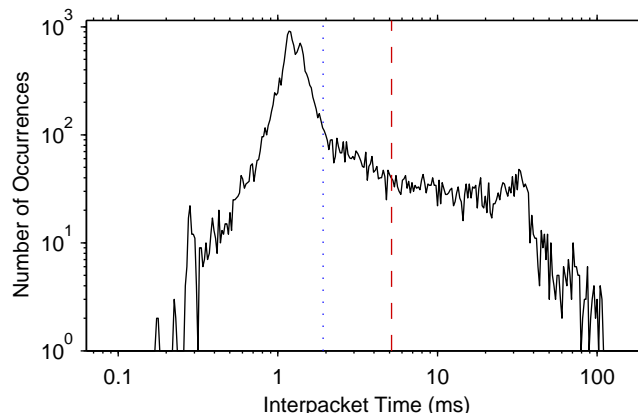


Figure 5.9: Log-log scaled histogram of interpacket times for the same ssh flow in Figures 2.2 and 2.3. The single peak is at the NIT of 10 Mbps Ethernet, but there is no clear peak corresponding to the RTT. The dotted line is at the mean of the distribution and the dashed line is at the mean plus one standard deviation (NUB).

multiple of the mode as an upper bound for the NIT, but this has the disadvantage of introducing a parameter into the algorithm, namely, which integer multiple of the mode to choose. For almost all flows the mean of the distribution will be slightly greater than the mode, since the region around the mode makes up a large portion of the mass of the distribution. To take into account that there may be more than one dominant NIT value, we add one standard deviation of the logarithm of the interpacket times to the mean to yield a parameterless upper bound on the range of NIT values. This is, in some sense, an arbitrary value, but for the majority of distributions it is a value midway between the largest NIT and the smallest RTT. We will refer to this value, the mean plus one standard deviation of the logarithm of the interpacket times, as the NIT upper bound, or NUB, see Figure 5.9.

Once we have computed the NUB, we group packets into clusters with interpacket times less than the NUB. Clusters can be as small as two packets, and they can be very large if the flow saturates the available bandwidth on the tight link. This brings us to the second subtlety of the algorithm, determining what constitutes a large cluster of packets. We will refer to a cluster that is deemed to be the appropriate size as a *valid* cluster; valid clusters should be on the order of *cwnd*. Since *cwnd* changes due to drops, and the packets in one flight can be dispersed over time by queuing, we need to have a flexible concept of what comprises a valid cluster.

To achieve this flexibility, we use a weighted moving average of cluster sizes. Let R be the number of bytes that is considered to compose a valid cluster. We initially set R equal the median of the receiver's buffer. Let M be the maximum of the receiver's buffer. If the monitor sees only the data packet and not the ACKs,

then we initially set both R and M to 65536 bytes. Let C be the current cluster size. We update R as follows:

$$R = \begin{cases} R & \text{if } C > M \\ 0.02R + 0.98C, & \text{if } 0.8R < C < M \\ 0.98R + 0.02C, & \text{if } C < 0.8R \end{cases} \quad (5.1)$$

In the second case we that the current cluster is big enough to be considered valid, yet not too big, i.e., $0.8R < C < M$. The weighted average used to update R heavily favors the current cluster, C . In the third case, when the cluster is not big enough to be considered valid, we still want to factor it in to the update of R , but not very heavily, so we reverse the weights in the average.

In Figure 5.10, the first two flights have very tight interpacket spacing, but that spacing becomes relaxed in the last four flights. The algorithm did not identify a valid cluster in the group of packets at 62.1 seconds, because there were interpacket times greater than the NUB within that group. The last two valid clusters are

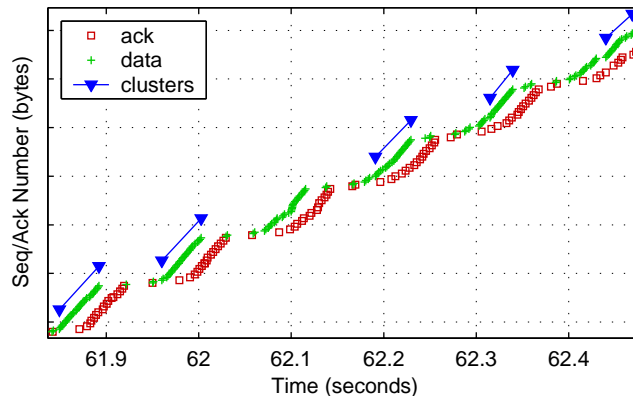


Figure 5.10: Seq-ack plot for the same ssh flow in Figures 2.2, 2.3, and 5.9. This plot shows that valid clusters are spaced about one RTT apart.

smaller than the first two; this is due to the weighted averaging of clusters. The trade-off for the flexibility of this algorithm is that when a drop occurs, it will take a few round-trip times for the weighted average to adjust to the smaller cluster sizes.

Once valid clusters have been determined, we estimate the RTT by taking the time from the beginning of one valid cluster to the beginning of the next. To determine the appropriate range of RTTs we create a histogram of the logarithm of the prospective RTTs, as in Figure 5.11. We use 75 equally spaced (on a log-scale) bins between 1 ms and 1000 ms. Any prospective RTTs less than 1 ms or greater than 1000 ms are not included. Outliers are eliminated by finding the number of occurrences of the mode (160 in Figure 5.11) and taking the full width of the distribution at half of this value. This width is indicated in Figure 5.11 by the two vertical dashed lines. We take this interval to be the range of allowable RTT values.

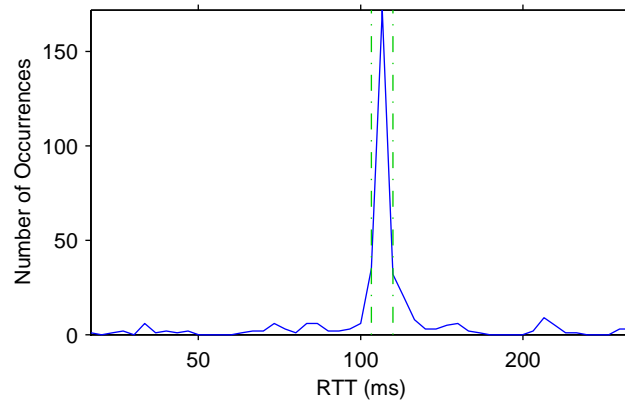


Figure 5.11: Histogram of prospective RTT values for the ssh flow. The best estimate of the RTT is 110 ms.

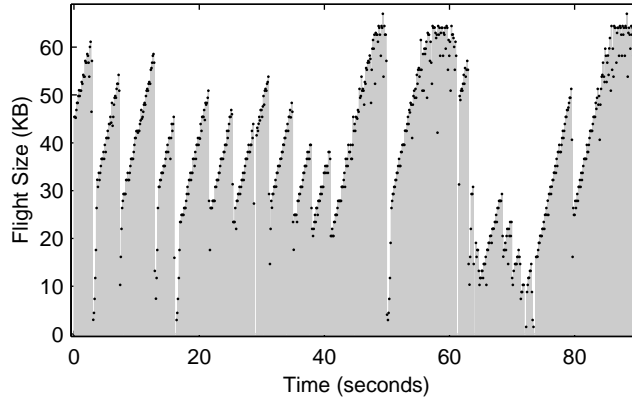


Figure 5.12: Reconstructed flight sizes for the ssh flow (compare to Figure 2.3). Note that flight sizes are reduced by both timeouts and triple duplicate ACKs. The 64 KB receiver’s buffer limits the flight size around 60 seconds and 90 seconds.

5.3.1 Congestion Window Estimation

Having calculated a satisfactory range of RTTs, we can more carefully group packets together to estimate the congestion window. Zhang et al. [51] have a similar method, but it relies heavily on the choice of parameters. To begin, we find the start of first discernible flight in the flow by taking the first packet after the largest interpacket time in the first 50 packets. Suppose this first packet has a timestamp of S , and suppose the range of round-trip times is $[T_1, T_2]$. Consider the collection of packets with timestamps between $[S + T_1, S + T_2]$, along with the last packet with a timestamp less than $S + T_1$. Within this collection, we again find the first packet after the largest interpacket gap; this packet will be the start of the next flight. Figure 5.12 shows that the flight sizes are consistent with the dynamics of the congestion window. We found that these algorithms give reliable results for more than 70% of the most active TCP flows from the NLANR data.

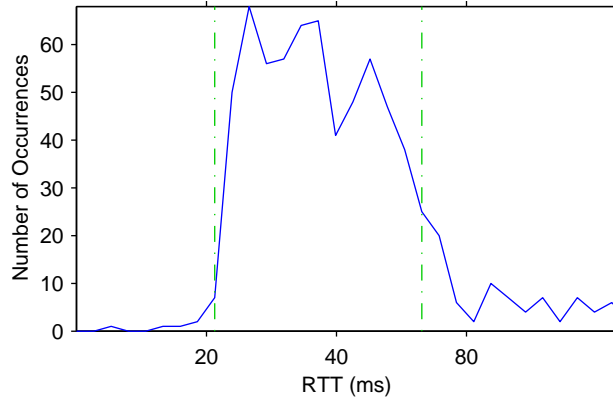


Figure 5.13: Histogram of prospective RTT values for a flow on port 3560 from March 4, 2004 taken at the MRA monitor. The best estimate of the RTT is 25 ms, but the range of RTTs, $[T_1, T_2]$, is 20 to 65 ms.

5.3.2 Real-time State Estimation

These algorithms result in a global estimate of the RTT, since all of the data are used to make the estimate. To estimate the RTT in real-time, the algorithm would have to be modified to make use of the data in a sequential manner. In fact, this will make the algorithm more flexible, as it allows for variations in RTT over the course of the flow. Not all flows will have small variance in their prospective RTT distribution as in Figure 5.11. As an example of a flow with large variations in its RTT, consider Figure 5.13. For this flow, the range of possible RTTs estimated by the algorithm is 20 to 65 ms. If we were to use such a large RTT range, then we would risk greatly overestimating *cwnd*. Thus it is important to modify the algorithm so that it works in real-time and results in a range that accurately tracks the RTT as it changes.

Instead of a fixed NUB value, we will use a moving average that changes with

the interpacket spacing. Let L_i be the sequence of the logarithm of the interpacket times and let A_i and S_i be the moving average and moving standard deviation of L_i . We compute the moving NUB value as follows:

$$A_{i+1} = (1 - w)A_i + wL_i \quad (5.2)$$

$$S_{i+1} = \sqrt{(1 - w)S_i^2 + w(A_i - L_i)^2} \quad (5.3)$$

$$\text{NUB}_{i+1} = A_{i+1} + S_{i+1} \quad (5.4)$$

The weight, w , should be small enough to give the moving averages a long memory; we choose $w = 0.01$.

The clusters and prospective RTTs are computed as before, except using NUB_{i+1} instead of a fixed value. To track fluctuations in the RTT, we use a histogram of the 30 most recent prospective RTTs. The mode of this histogram will be the best estimate of the RTT at a given time. It is computationally easy to keep track of the mode, M ; however, it is not feasible to compute the width of the histogram, $[T_1, T_2]$, as we did before. A simple method for calculating the range of round-trip times is to set $[T_1, T_2] = [0.7M, 1.3M]$. This range is narrow enough so that it will encompass only one window of packets. With this RTT range we compute flight sizes as before, then we can reestimate the RTT as the time from the beginning of one flight to the beginning of the next. This final estimate of the RTT is shown in Figure 5.14 along with the moving mode, M .

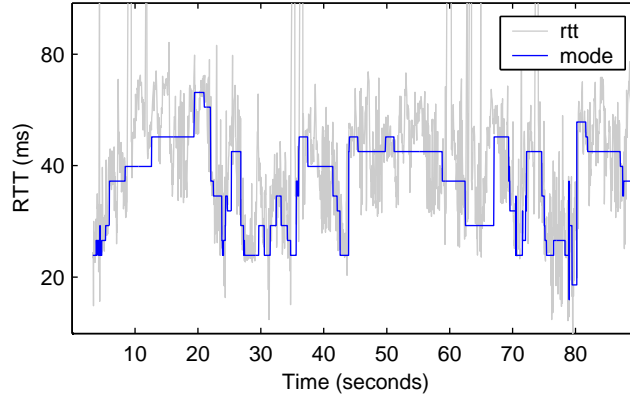


Figure 5.14: Estimate RTT time series for the flow from Figure 5.13. Both the moving mode and the final RTT estimates are plotted.

5.4 Frequency Algorithm

The frequency algorithm uses the self-clocking nature of TCP to determine the RTT by observing the approximate period of interpacket times. We use four separate methods to estimate the RTT, then combine the results to estimate a lower bound on the RTT. This estimate should be the greatest lower bound, that is, it should closely approximate the fixed delay. Once the RTT has been estimated we can reconstruct the sequence of flights.

The four components of the frequency algorithm are a sliding window estimate, the autocorrelation function of the interpacket times, the data-to-ACK-to-data time, and the Lomb periodogram. The four components are combined in a nonlinear way that first identifies the most likely group of RTT values, then finds the smallest of those values to generate a lower bound.

The frequency algorithm was designed to work in real-time. To generate the initial estimate of the RTT, we restrict the algorithm to use only the first 256

interpacket times (257 packets). This is because the Lomb periodogram is based on the FFT and it is most efficient on sequences that are powers of two in length. Also, 257 packets will comprise at least five complete flights, whereas 128 packets would only guarantee two complete flights in most cases. We want to use enough data to obtain a good initial estimate of the RTT, but not so much that it becomes computationally expensive.

5.4.1 Sliding Window Upper Bound

The purpose of the sliding window estimate is to obtain an upper bound on the fixed delay. Let M be the maximum receiver's buffer. We let $M = 65536$ if ACKs are not seen by the monitor. Let t_i be the timestamps of the data packets, and let b_i be the number of data bytes in the payload of the packets. For each i , let n_i be the smallest integer such that

$$\sum_{k=i}^{n_i} b_k > M.$$

As i increases, packets i to n_i will form a sliding window of at least M bytes. The sender is required to send no more than M bytes per RTT. Therefore, packet i and packet n_i must necessarily be in different flights, and hence they are separated by at least one RTT. Let i^* be the largest integer such that $n_{i^*} < 257$, and for each $i \leq i^*$ let

$$u_i = t_{n_i} - t_i \tag{5.5}$$

be a RTT upper bound.

If the sender is transmitting less than M bytes per RTT, then u_i might grossly

overestimate the RTT. On the other hand, if the congestion window is equal to M , then it is possible for u_i to actually underestimate the RTT. To see how this is possible consider the following simplified scenario where the receiver's buffer is two packets and delayed ACKs are used. Suppose the RTT is 5 ms and the three links from sender to receiver are 100 Mbps, 155 Mbps and 10 Mbps, with the monitor on the middle link. This situation is illustrated by a seq-ack plot in Figure 5.15. In the absence of cross traffic the interpacket time between two packets in the same flight will be 0.12 ms at the monitor, and 1.2 ms at the receiver. Suppose that in one flight the first packet experiences no queuing, but when the second packet arrives at the OC3 link, it is queued after nine other 1500 byte packets from a faster link. This will cause the interpacket time between the first and second packet to be about 0.7 ms. Since the narrow link is 10 Mbps the receiver will still detect the same interpacket

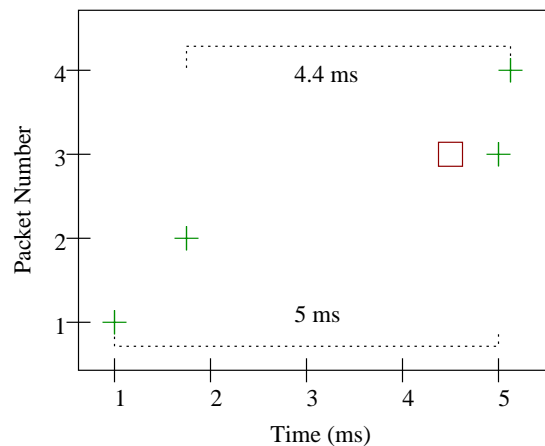


Figure 5.15: This seq-ack plot shows how the sliding window method can underestimate the RTT. Packets 1, 3 and 4 are not queued, but packet 2 is queued for 0.7 ms before passing through the monitor. The time between packets 2 and 4 is 4.4 ms, which is less than the true RTT of 5 ms.

time of 1.2 ms, as if there had been no queuing. The delayed ACK would be sent after the receipt of the second packet, and the sender would receive that ACK at the same time it would have had there been no queuing. Now suppose that the next flight experiences no queuing at all. Then the time between the second packets in each flight will be about 4.4 ms, which underestimates the RTT.

Even though it is unlikely to underestimate the RTT in such a way, we intend for u_i to be a strict upper bound on the fixed delay. We would like to obtain the least upper bound for the RTT from the values u_i . Therefore, instead of taking $\min_{i \leq i^*} u_i$ as the least upper bound we use the more conservative estimate of the fifth percentile of u_i . Since i^* will be greater than 200 in most cases, Choosing the fifth percentile equates to ignoring the lowest ten values. More formally, let \hat{u}_i be the values u_i sorted in increasing order. The conservative estimate of the least upper bound is

$$u = \hat{u}_{\lfloor i^*/20 \rfloor}. \tag{5.6}$$

5.4.2 Autocorrelation Function

One technique to estimate the period of an almost periodic time series is the autocorrelation function. Let $x_i, i = 1 \dots n$ be a normalized time series. The autocorrelation function is defined as

$$A(k) = \frac{1}{n-k} \sum_{i=1}^{n-k} x_i x_{i+k} \tag{5.7}$$

for lags $k = 0 \dots n-1$. Note that the autocorrelation function as defined is unbiased, that is, the normalizing coefficient $\frac{1}{n-k}$ takes into account the lag. If a time series

has an approximate period of T , then one would expect $A(k)$ to have its largest local maximum at $k = T$.

We expect the interpacket times to be approximately periodic because of the self-clocking nature of TCP. Unfortunately, the interpacket times are unevenly spaced in time. One could ignore this fact and treat them as if they were an evenly spaced time series. But if the flow is in the linear increase phase of congestion avoidance or a drop occurs, then the sequence of interpacket times will be misaligned.

We handle these issues by linearly interpolating the logarithm of the interpacket times on an evenly spaced grid. We choose the step size of the grid, s , to be 1 ms. This is a trade-off between accuracy and computational workload. If we made the step size smaller, then we would more closely approximate interpacket times less than 1 ms, but then the grid would be larger and computing $A(k)$ would take longer. Figure 5.16 shows the sequence of interpacket times, their linear interpolation, and $A(k)$ for the interpolated sequence.

One can choose the step size adaptively according to the average of the interpacket time or the average of their logarithms. This introduces more complexity than is necessary, but we describe one adaptive step size here. Let m_1 be the mean of the first 256 interpacket times and let m_2 be the mean of their logarithms. We

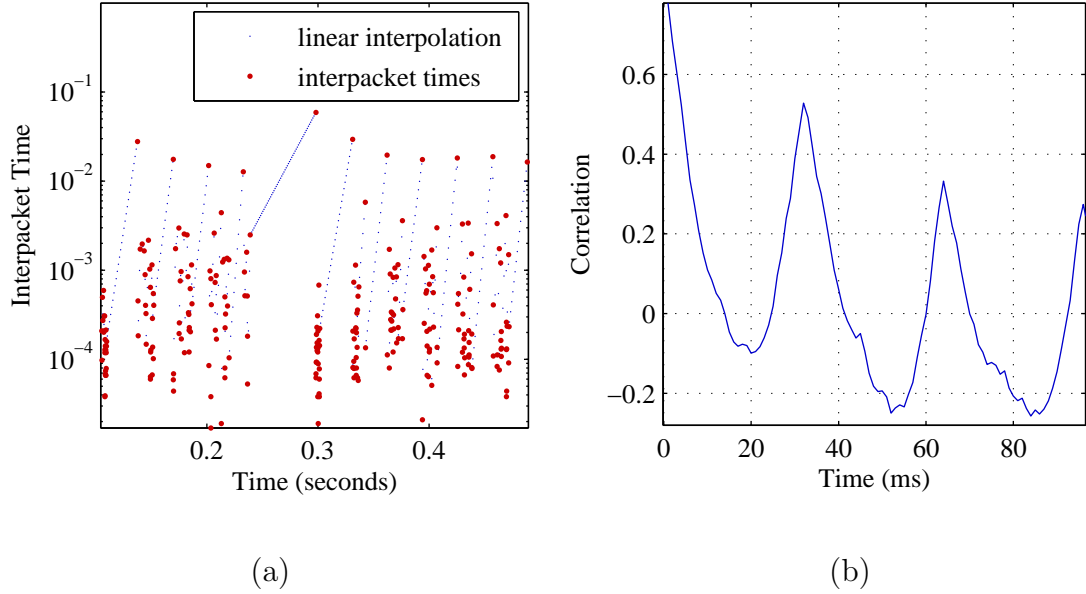


Figure 5.16: (a) The first 256 interpacket times of an FTP flow from the UFL monitor taken November 30, 2004. The smaller dots are the linearly interpolated values on a grid with step size 1 ms. (b) The autocorrelation function of the interpolated data. Based on this plot one can estimate the RTT to be about 32 ms.

define $m_3 = \frac{1}{2}\sqrt{m_1 m_2}$ and choose the step size, s , to be:

$$s = \begin{cases} 10\text{ms} & \text{if } 10\text{ms} < m_3 \\ m_3 & \text{if } 1\text{ms} < m_3 < 10\text{ms} \\ 1\text{ms} & \text{if } m_3 < 1\text{ms} \end{cases} \quad (5.8)$$

This choice of s reduces the amount of computation for flows with longer interpacket times, but does not significantly alter the final estimate of the RTT. Depending on the RTT and congestion window the first 257 packets generally require between 0.1 and 4 seconds to transmit. Therefore, the grid size, $g = (t_{257} - t_1)/s$, can be up to 4000 points. If the sender has no data to transmit for a brief period or a timeout

occurs, then the grid can potentially be larger than 4000 points.

This raises the question of what the best method is to compute the autocorrelation function. Let $\{x_i\}_{i=1}^g$ be the time series of linearly interpolated interpacket times, and assume that the time series is normalized to have mean zero and standard deviation one. The Wiener-Khinchin theorem [] states that the Fourier transform of the autocorrelation function equals the power spectrum. Therefore, we can approximate the autocorrelation function by

$$\tilde{A}(k) = F^{-1} (|F(x)|^2) \quad (5.9)$$

where F is the Fourier operator. $\tilde{A}(k)$ is only an estimate of the autocorrelation function because of the finite length of the time series.

Computing $A(k)$ using the definition requires $O(g^2)$ operations, but by using the FFT to compute the transforms in Equation 5.9 we reduce the complexity to $O(g \log_2 g)$. However, we do not need to compute $A(k)$ for all lags k . Since we have already computed u , the upper bound for the RTT, we only need to calculate $A(k)$ up to $k = \lceil u/s \rceil$. Now using the definition will only require $O(gu/s)$ operations. Suppose that we use both methods to compute the autocorrelation function for a flow with a RTT of 40 ms and receiver's buffer equal to 12 packets. The grid size will be $g = 40 \cdot 256/12 \approx 850$. Using the definition will take about $40g \approx 34000$ operations. When using the FFT it is best to have g equal to power of two, so x is padded with zeros to make g exactly 1024. The order constant for the split-radix FFT algorithm is 2, and there are 2 transforms in Equation 5.9. Therefore, using Equation 5.9 instead of the definition requires about $4g \log_2 g \approx 40000$ operations.

One can see that using the FFT is not necessarily faster due to the relatively small size of g . One could use this type of analysis for each flow to determine whether it is more efficient to use the definition or Equation 5.9. However, we always use the definition since it is a better estimate of the autocorrelation function and it is not a significant computational burden at this point.

We approximate the RTT by $q = k^*s$, where k^* is the lag at which $A(k)$ attains its greatest local maximum on the interval $(0, \lceil u/s \rceil)$. $A(k)$ can fail to have its greatest local maximum at the value that corresponds to the true RTT. This can happen in two basic ways. First, $A(k)$ could be greater at the value that corresponds to twice the RTT; we can encounter this if u greatly overestimates the RTT. Secondly, there can be a spurious local maximum close to $k = 0$, which corresponds to a fairly large interpacket time that is repeated at even intervals.

By filtering $A(k)$ we have a better chance of finding the correct local maximum. To correct for q being too small we use a simple low-pass filter by applying to $A(k)$ a moving average of the last β values. To correct for q being too large we multiply $A(k)$ by a linear envelope that equals 1 at $k = 0$, and α at $k = \lceil u/s \rceil$. After originally calculating the unbiased autocorrelation function, multiplying by a linear envelope reintroduces bias, but it does so on the time scale of the RTT rather than the time scale of the grid. In practice we use two filtered versions of $A(k)$, along with the $A(k)$ itself. Let $A_1(k) = A(k)$. Let $A_2(k)$ be a filtered version of A with $\beta = 4$ and $\alpha = 3/4$. Let $A_3(k)$ be a filtered version of A with $\beta = 8$ and $\alpha = 1/2$.

From the three versions of $A_j(k)$, we obtain three estimates of the RTT,

$$q_j = k_j^* s, \quad j = 1, 2, 3. \quad (5.10)$$

If $A_j(k)$ has no local maximum on the interval $(0, \lceil u/s \rceil)$, then $q_j = 0$. In general, these estimates should be decreasing order, $q_1 > q_2 > q_3$.

5.4.3 Data-to-ACK-to-Data

The sliding window algorithm produces an upper bound, u , on the fixed delay. To obtain a lower bound we estimate the time between data packets that occupy the the same relative positions in two successive flights of packets. This method assumes ACKs, so this method will not work if the monitor does not see the ACKs for a particular flow (this happens for about 20% of flows. This method also assumes the sender is closer to monitor. If this assumption does not hold, and the sender is on the remote side of the monitor, then this method may result in an estimate that is spuriously low. Since we intend for this method to produce a lower bound, spuriously low values are not a fatal flaw.

We now describe the data-to-ACK-to-data method. Suppose we have a flow that meets the requirements laid out above, and suppose the data packets have timestamps t_i and sequence numbers s_i and the ACKs have timestamps t_j^a and acknowledgment numbers a_j . For data packet i find the next corresponding ACK, that is, the first ACK such that $t_j^a > t_i$ and $a_j > s_i$. Call the index of this ACK j' . Now find the first data packet that ACK j' liberates, that is, the data packet with index m_i such that $t_{m_i} > t_{j'}^a$ and $s_{m_i} > a_{j'}$. As in the sliding window algorithm,

let i^* be the largest integer such that $m_{i^*} \leq 257$. Our overall estimate of the RTT lower bound is defined as

$$\ell = \min_{i \leq i^*} (t_{m_i} - t_i). \quad (5.11)$$

Figure 5.17 illustrates this method with two examples, one where the sender is closer to the monitor, and another where the receiver is closer to the monitor.

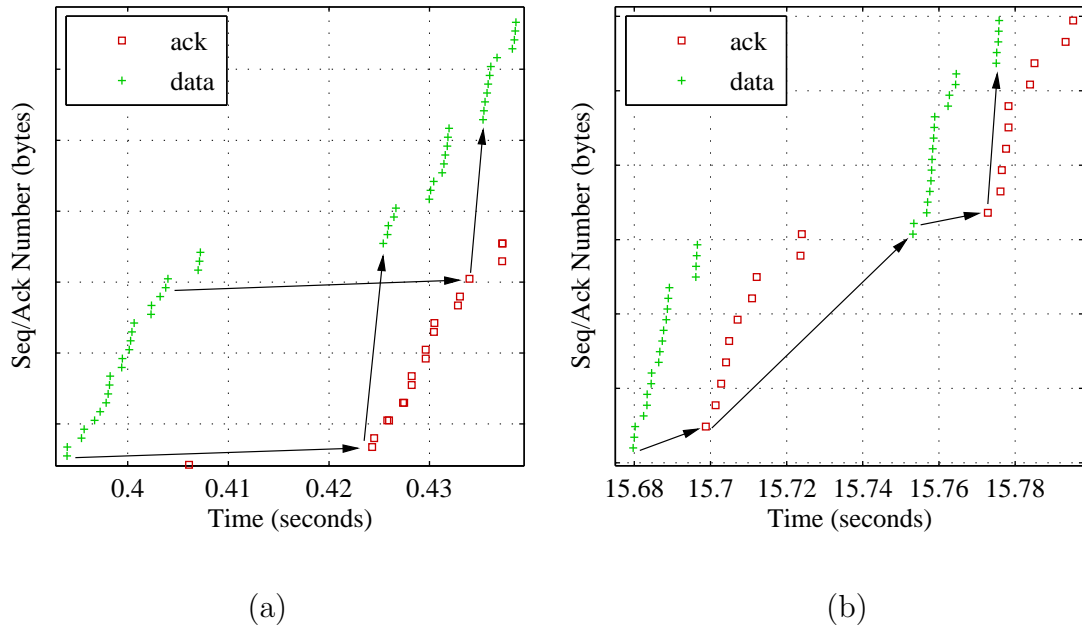


Figure 5.17: The seq-ack plots in (a) and (b) are for two flows from the same UFL trace. In (a) the monitor is closer to the sender, in (b) it is closer to the receiver. By following each pair of arrows in (a) we find a valid RTT estimate. In (b) the first pair of arrows gives a good estimate, but the result of the second pair is too small, because the ACKs start arriving at the monitor before the end of the flight of data packets.

5.4.4 Lomb Periodogram

The Lomb periodogram [41] is a method for the spectral analysis of unevenly spaced data, such as interpacket times. We could instead use the FFT on linearly interpolated data, but the Lomb periodogram is more naturally suited to analyze data in which the sampling is highly irregular. If there are long stretches without data, as there would be after a timeout or when a sender has no data to transmit, then the FFT can exhibit erroneously large power at low frequencies. This further supports our choice of not using the FFT to estimate the autocorrelation function in Section 5.4.2. Figure 5.18 shows a comparison of the Lomb periodogram versus the FFT after interpolation.

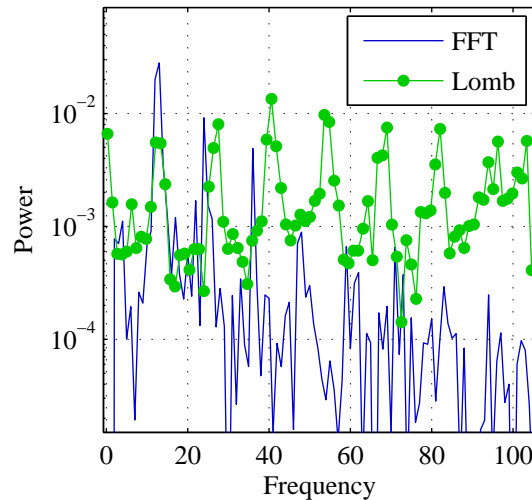


Figure 5.18: The Lomb periodogram and FFT have the same fundamental frequency of about 16, corresponding to a RTT of 65 ms. The periodogram picks up the harmonics, while the FFT leaks power to lower frequencies and decays rapidly. In flows with RTT greater than 100 ms this can cause the fundamental frequency to be distorted or obscured. This plot is based on data from the flow in Figure 5.17(b).

Suppose we are given data x_i measured at times t_i , $i = 1 \dots n$ and suppose the data have mean \bar{x} and standard deviation σ_x . The Lomb periodogram is a measure of spectral power at angular frequency $\omega = 2\pi f$ defined by:

$$L(\omega) = \frac{1}{2\sigma_x^2} \left[\frac{(\sum_i (x_i - \bar{x}) \cos \omega(t_i - \tau))^2}{\sum_i \cos^2 \omega(t_i - \tau)} + \frac{(\sum_i (x_i - \bar{x}) \sin \omega(t_i - \tau))^2}{\sum_i \sin^2 \omega(t_i - \tau)} \right] \quad (5.12)$$

In this Equation τ causes $L(\omega)$ to be invariant under any shift of the measurement times, which is a key property it shares with the Fourier transform. The value of τ depends on ω as follows:

$$\tan(2\omega\tau) = \frac{\sum_{i=1}^n \sin 2\omega t_i}{\sum_{i=1}^n \cos 2\omega t_i}. \quad (5.13)$$

Lomb [28] showed that for each fixed ω , Equation 5.12 is equivalent to solving the least squares problem where the data are fit to the model

$$y(t) = a \cos(\omega t) + b \sin(\omega t).$$

This makes clear the difference between this method and the FFT on interpolated data. The FFT weights each time interval equally, whereas the Lomb periodogram weights each data point equally.

The algorithm we use to compute the Lomb periodogram makes indirect use of the FFT and requires $O(n \log n)$ operations. Since we are dealing with such a small data set it is not computationally expensive to use the definition directly. In fact, one can maintain an estimate of $L(\omega)$ as new packets arrive by assuming that τ is fixed and keeping track of the four summations in Equation 5.12.

Let ω^* be the angular frequency that maximizes $L(\omega)$ on a suitable grid over the interval Ω , where $\Omega = [2\pi/u, 2\pi/\ell]$ if both u and ℓ are greater than 5 ms, and

$u > \ell$, otherwise let $\Omega = [2\pi/1.0, 2\pi/0.005]$. See [41] for information on how to compute the grid. We then define an estimate of the RTT by

$$\theta = \frac{2\pi}{\omega^*}. \quad (5.14)$$

One might ask if a peak in $L(\omega)$ is truly significant. In this case the null hypothesis is that data x_i are independent samples from a Gaussian distribution. Scargle [44] showed that if the null hypothesis is true, then for each fixed ω , $L(\omega)$ follows an exponentially distribution with mean 1. If we compute the Lomb periodogram for m independent frequencies, $\omega_1 \dots \omega_m$, then the probability that none of the powers $L(\omega_1) \dots L(\omega_m)$ are greater than z is $(1 - \exp^{-z})^m$. The null hypothesis will result in at least one power greater than z with probability

$$p_z = 1 - ((1 - \exp^{-z})^m). \quad (5.15)$$

One can view p_z as a significance level of a peak of size z in the Lomb periodogram.

In practice we would like to use the significance level to ascertain when it is appropriate to incorporate θ into the overall estimate of the RTT. However the significance level comes with a caveat: it is only valid if the data is not clumped. By clumped we mean there are several data points close to each other with nearly equal spacing, with large spaces at regular intervals separating the clumps. This accurately describes the interpacket times. Clumping effectively decreases the number of independent frequencies, so the value p_z should be viewed as lower bound for the true significance of any peak. For instance, we might claim that only peaks with $p_z < 0.2$ should count toward the overall RTT estimate, but a value of p_z equal to 0.2 might correspond to a true significance level much closer to 1. Therefore, we

always use θ , but we combine it with the other RTT estimates in such a way that a spurious value of θ is unlikely to affect the overall estimate of the RTT.

5.4.5 RTT Lower Bound

We have defined a set of six RTT estimates, u , q_1 , q_2 , q_3 , ℓ and θ ; call this set S . We now describe a way to combine the unreliable estimates in S into one reliable RTT estimate, \underline{r} . We say informally that two estimates *agree* if they are within 10% of each other. The basic idea is to find the smallest value in S that agrees with at least one other estimate.

We set 5 ms as the minimum feasible RTT; any estimate less than this value is discounted. If u is less than 5 ms, then it is inferred that the flow must have been using window scaling. Since the TCP options are not included in the trace format this is the only way to infer the presence of window scaling. After determining \underline{r} based on the other five estimates we attempt to deduce the multiple used for window scaling by increasing M , the maximum receiver's buffer, by a factor of two and reevaluating u until it agrees with \underline{r} .

The meaning of *agree* is formalized by defining a relation \cong , such that $a \cong b$ if $0.9 < a/b < 1.1$ or $0.9 < b/a < 1.1$ or $|a - b| < 3$ ms. Let $S' = \{s \in S | s > 5\text{ms}\}$. As an intermediate step toward defining \underline{r} we define r' as:

$$r' = \min\{s \in S' \mid \exists t \in S', t \cong s\}. \quad (5.16)$$

If none of the elements of S' agree, then r' will be undefined. In such a case let $r' = \ell$ if the sender is on the local side of the monitor as determined by the over-

lap/bootstrap algorithm, otherwise let $r' = u$. Finally, we define \underline{r} , the estimated lower bound of the RTT, as

$$\underline{r} = 0.95r'. \quad (5.17)$$

The reason for multiplying by 0.95 is that r' can potentially equal u , which is an upper bound for the fixed delay.

It is also possible for the RTT to be elevated well above the fixed delay for the entirety of the first 257 packets. If the congestion later subsides, then the initial estimate of \underline{r} will no longer be valid. It is fairly simple to continue evaluation of u and ℓ in real time as new packets arrive. It is also possible to update the autocorrelation function and the Lomb periodogram if one uses their definitions instead of FFT approximations. We only estimate \underline{r} based on the first 257 packets and do not attempt to keep track of a real time estimate as conditions change.

5.4.6 Congestion Window Estimation

The method of congestion window estimation in the frequency algorithm is similar to that of the clustering algorithm. The main difference is that we now have a better estimate of minimum RTT, and place no restriction on the upper limit of the RTT. The basic idea of the current *cwnd* estimation scheme is as follows: given a past history of window sizes $w_1 \dots w_{k-1}$ and the first packet in the k th window, increment w_k by the number of data bytes in each packet that arrives until one minimum RTT has elapsed, and then continue incrementing w_k until it is greater than w_{k-1} or it equals M , the maximum receiver's buffer.

As in the clustering algorithm, we start with the first packet after the largest interpacket time among the first 50 packets. Assume that the congestion window is initially equal to the receiver’s buffer and let $w_0 = M$ be a placeholder that allows us to compare w_1 to w_0 . This assumption is necessary since most of the flows are already in progress once the trace starts, thus, we normally do not see the slow start phase.

Two timers are needed for this algorithm, one limits the allowable time between packets in the same flight, and the other is the expected time between flights. The first timer, c_1 , is set at $0.75r$. If two packets are separated by more than c_1 seconds, it is assumed a new window has begun and the second of the two packets is the initial packet in the new window. Let \bar{b} be the median number of data bytes per packet, and let $p = 1 - \bar{b}/M$. If the sender is transmitting M bytes per RTT and the bandwidth delay product is much greater than M , then we expect flow to have M/\bar{b} packets per RTT, which leads to $M/\bar{b} - 1$ small interpacket times followed by one large interpacket time. Thus, we define the second timer, c_2 , as the p th percentile of the interpacket times. In practice we handle cases where almost all interpacket times are the same by constraining c_2 to the interval $[0.05r, 0.5r]$, clipping it to the endpoints if it is outside them.

As before, t_i is the timestamp, s_i is the sequence number, and b_i is the number of data bytes of the i th packet. Define the i th interpacket time as $\delta_i = t_i - t_{i-1}$. Suppose the current (k th) window has w_k bytes. Suppose that i is the index of the first packet in current window, and current packet had index $j \geq i$. We increment the window size, $w_k \rightarrow w_k + b_j$, if and only if the following is true about conditions

A-E listed below: condition A is satisfied, at least one of conditions B-D is met, and condition E is not satisfied:

- A - *Receive Window*: $w_k + b_j \leq M$
- B - *Congestion Avoidance*: $w_k < w_{k-1} + \bar{b}$
- C - *Slow Start*: $w_k < 2w_{k-1} < M$ and $\delta_i < c_2$
- D - *Two Packet Increase*: $w_k < w_{k-1} + 2\bar{b} < M$ and $\delta_i < c_2$
- E - *Early Break*: $\delta_j > c_1$ or $(t_j - t_i > r$ and $\delta_j > c_2)$

If condition A is false, or conditions B-D all fail, or condition E is true, then packet j is deemed to be part of the next flight, and we set $w_{k+1} = b_j$. Note that this algorithm takes care to determine flight boundaries, that is, the first and last packet in each flight. There are simpler ways to obtain a crude estimate of the congestion window, if that is all one cared about, but the flight boundaries are necessary to compute per-flight statistics that are related to congestion.

A few of the inequalities in conditions A-E require further elucidation. The inequality $\delta_i < c_2$ means that flights k and $k - 1$ were closer than expected, which is a sign of congestion. The inclusion of $\delta_i < c_2$ in conditions C and D ensures those conditions are only satisfied when there is some indication of congestion. Condition D allows *cwnd* to increase by two packets per RTT; this is mainly due to delayed ACKs and changes in the receiver's buffer. For some flows, conditions C and D will be true most of the time, and without condition E this would cause w_k to increase unduly. The first inequality in condition E allows the loop to terminate

when $t_j < t_i + r$, that is, before one minimum RTT has elapsed. This allows for realignment of flights to more natural boundaries, but can lead to spuriously small *cwnd* or RTT values that will need to be filtered out in post-processing. The second and third inequalities in condition E allow the loop to terminate once the minimum RTT has elapsed, even if conditions A-D still hold.

We feel that the above rules apply to most bulk TCP flows without modification. However, this method can be improved by utilizing ACKs. Doing so will require knowledge of flow orientation. In cases where the monitor is closer to the sender, one may use the data-to-ACK-to-data method, described in section 5.4.3, to predict the first packet in flight $k + 1$ given the first packet in flight k . If the monitor is closer to the receiver, then the first packet in a flight is often preceded by an ACK with acknowledgment number equal to the sequence number of that packet. This rule of thumb is not always necessary, due to delayed ACKs, and it is far from sufficient. Although the congestion window algorithm might be improved by using ACKs, the difficulties in doing so currently outweigh the possible benefits.

Chapter 6

Validation and Prediction

In this chapter, we show that the clustering and frequency algorithms described in the previous chapter are robust and accurate for data produced with the *ns2* network simulator. We then present results of applying the clustering and frequency algorithms to a large collection of NLANR trace data. Finally, we discuss the predictability of network traffic using nonlinear time series analysis.

6.1 Validating The Clustering and Frequency Algorithms

The *ns2* network simulator [33] is a widely-used tool to design and test networks and network protocols. We create a simulation to validate the clustering and frequency algorithms. Although this simulation is highly simplified, it was designed to reflect the key features of the university networks from which the trace data originated. We used two network configurations; in configuration S, the monitoring point is positioned closer to the senders, in configuration R, it is closer to the receivers.

Figure 6.1 shows the network diagram of configuration S; configuration R is similar, except the positions of the senders and receivers are reversed, as are the cross traffic sources and sinks. We chose a 3 Mbps link with a one-way delay of 75 ms, as link 6 – 8 does, because it is easier to visualize congestion this way. A more realistic scenario is to replace link 6 – 8 with a 155 Mbps link with a one-way delay

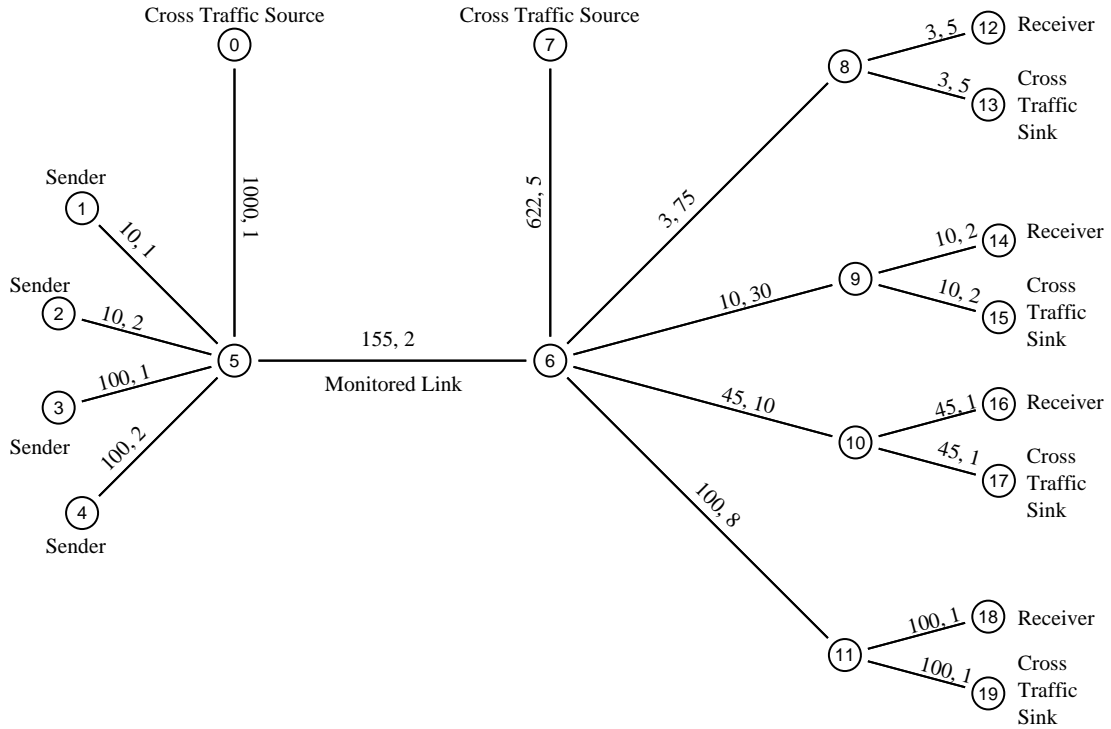


Figure 6.1: This network diagram illustrates configuration S. The pair of numbers on each link are the link capacity in Mbps and the one way delay in milliseconds.

For example, the fixed delay for sender receiver pair (4,18) is 26 ms.

of 75 and combine links 8 – 12 and 8 – 13 into a single 3 Mbps bottleneck link. This modification fits the paradigm of a high capacity backbone and a low capacity edge link, but it would not drastically change the RTT or other statistics of the flow.

We studied four main flows between sender-receiver pairs (1,12), (2,14), (3,16), and (4,18). We will refer to these as flows 1, 2, 3 and 4. Each flow is a bulk FTP transfer with the receiver’s buffer set to 22 packets for flows 1 and 2, and 44 packets for flows 3 and 4. The transfers were 90 seconds in duration, so that the amount of traffic generated would be similar to that of real flows. Cross traffic was also generated. All TCP flows used TCP Reno and delayed acknowledgments. All

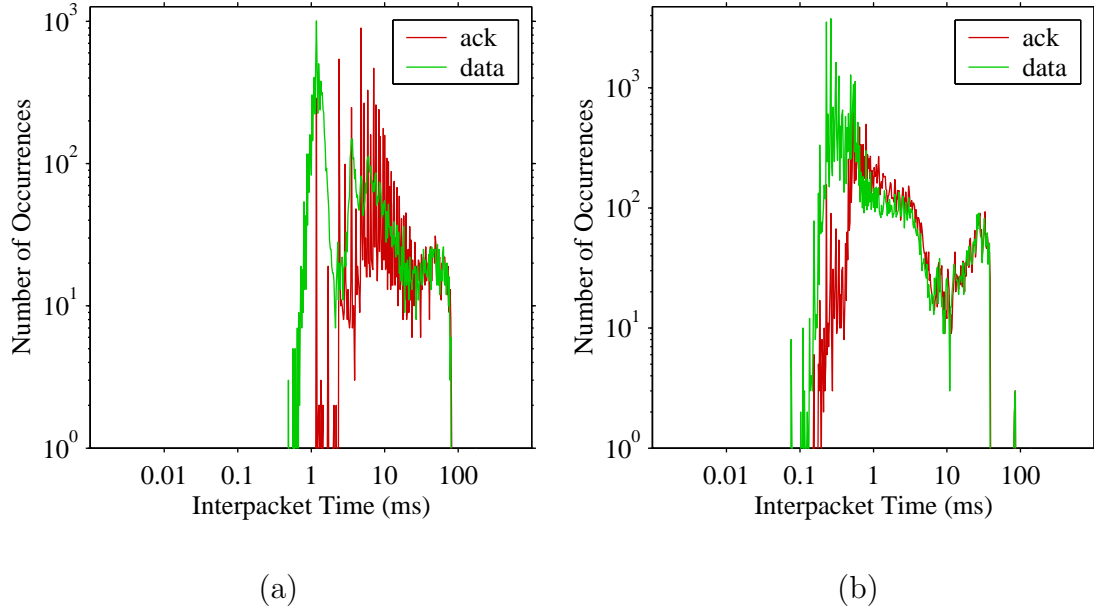


Figure 6.2: Interpacket time histograms for flow 2 of simulation S2 (a), and flow 3 of simulation R2 (b). They are qualitatively similar to interpacket time histograms we have seen for real data.

routers used the RED queue management scheme with a maximum queue size of 45 packets. Limiting the queue to this low, potentially unrealistic value keeps the round-trip times from varying more than one order of magnitude. We note that none of the queues ever reached more than 40 packets. The similarity between the *ns2* generated traffic and the real traffic is borne out by the close correspondence of the interpacket time distributions, as is shown in Figure 6.2.

For each network configuration, two simulations were performed with different cross traffic characteristics. In simulation 1, cross traffic does not travel across the monitored link. For example, in configuration S simulation 1 (S1), cross traffic flows between source host 7 and sink hosts 13, 15, 17, and 19. In simulation 2, cross traffic travels on both the tight link and the monitored link. This makes it possible for

	R1	R2	S1	S2
Senders	12,14,16,18	12,14,16,18	1,2,3,4	1,2,3,4
Receivers	1,2,3,4	1,2,3,4	12,14,16,18	12,14,16,18
Source(s)	13,15,17,19	13,15,17,19	7	0
Sink(s)	7	0	13,15,17,19	13,15,17,19

Table 6.1: Each column summarizes the flow orientations for the simulation at the top of the column. For example, in simulation R2 flow 3 has host 16 as sender and host 3 as receiver. Note that the host numbering remains fixed in Figure 6.1, but the role of each host can be either sender or receiver.

drops to occur before or after the monitoring point. Table 6.1 summarizes the flow orientation for each simulation.

The cross traffic is a mix of TCP and constant bit rate UDP flows, as well as exponential on-off (EOO) flows. An EOO flow is essentially a UDP flow that sends bursts of traffic where the duration of the burst is a random variable sampled from an exponential distribution with mean λ_{ON} , and the idle time between bursts is also exponentially distributed with mean λ_{OFF} . In all simulations we let $\lambda_{\text{ON}} = \lambda_{\text{OFF}} = 0.5$ seconds. Simultaneous low bandwidth EOO flows were used to generate random oscillations in the cross traffic. It is also easier to work with dozens of EOO flows than hundreds or thousands of transient TCP flows. All packets were 1500 bytes, except for the UDP traffic, which we made smaller to prevent interpacket times from being integer multiples of each other. We made UDP packets 640 bytes long, since that is a common packet size associated with Real Time Streaming Protocol (RTSP),

which uses UDP. The amount of cross traffic was identical in all four simulations, the only difference was the source and destination hosts, as detailed in Table 6.1. The characteristics of the cross traffic are summarized in Table 6.2.

We tested the clustering and frequency algorithms on the four main flows. We expect the algorithms to perform better on simulations using configuration S, since the perspective of the monitor is closer to that of the sender. The simulator records the true value of the following variables: congestion window, RTT, RED drop probability, instantaneous and average queue length at each router, and the timestamps of data packets and ACKs when they leave or enter (depending on the network configuration) the queue for link 5 – 6 at host 5.

	13		15		17		19	
Traffic type	Rate	Flows	Rate	Flows	Rate	Flows	Rate	Flows
UDP	0.1	1	0.25	1	2.0	1	5.0	1
EOO	0.05	6	0.125	8	1.0	12	2.5	16
TCP	22	3	22	4	44	6	44	8

Table 6.2: The numbers at the top of each column refer to the cross traffic source (configuration R), or sink (configuration S). The numbers in the flows columns are the number of concurrent flows of that type. The rate columns for the UDP and EOO rows are the constant rates (in Mbps) at which the flow(s) send. In the TCP row, 22 and 44 denote the size of the receiver’s buffer.

6.1.1 Clustering Algorithm Validation

We tested the real-time version of the clustering algorithm described in Section 5.3.2, not the version that uses all data at once. The real-time version is more appropriate for the simulated flows, since their round-trip times can vary appreciably. The real-time clustering algorithm requires an initialization period that is on the order of 30 times the RTT. Therefore, we will be unable to compare the RTT and *cwnd* estimates during the slow start phase.

We are not able to directly compare RTT estimates to the true values, since they are recorded at different times. The simulator records the true RTT, but during drop events it occasionally skips times when there should be a value recorded. For these reasons, we linearly interpolate the real and estimated RTT values on a grid with a step size equal to the fixed delay of the flow. This will allow us to compare interpolated values directly, but there are still problems with this approach. We determine an estimate of the RTT for a flight of packets when the first packet in the next flight arrives at the monitor, while the simulator determines the RTT when the ACK that liberated that packet arrives at the sender. If the monitor is closer to the receiver, as it is in configuration R, then the times at which the real and estimated RTT are recorded are likely to be shifted. Another problem is that the simulator measures the RTT with millisecond precision, whereas we estimate it with microsecond precision. Thus, we are not likely to approximate the true RTT to an accuracy greater than 0.25 ms. Despite these problems the estimated RTT tracks the qualitative behavior of the true RTT, as is evident in Figure 6.3.

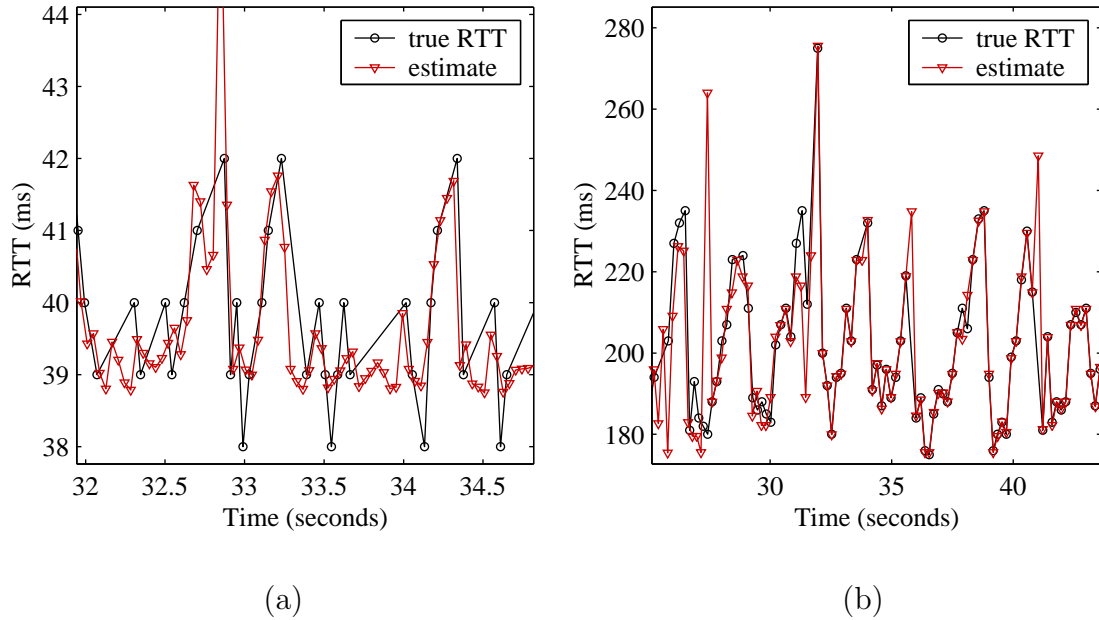


Figure 6.3: The RTT for flow 3 of simulation R2 is shown in (a), and flow 1 of simulation S1 is shown in (b). In (a) one can see the true RTT is confined to integer values in the simulator. Most importantly, the estimate tracks the true RTT as it increases, this is clear in (b).

Let \tilde{r}_i be the i th interpolated RTT estimate, let r_i^T be the interpolated true RTT, and let $E_i = \tilde{r}_i - r_i^T$. There is no statistically significant bias in the RTT estimates, that is, E_i is equally likely to be greater than or less than 0. There is some bias for individual flows, but there is no systematic bias across all flows. There are occasionally very large deviations between the approximate RTT and the true RTT; the maximum absolute error, $\max_i |E_i|$, can be greater than the RTT itself. The mean absolute error tends to be skewed upward by these larger errors. Therefore, the median absolute error is a better measure of the accuracy of the algorithm. Table 6.3 summarizes the errors for all simulated flows. The median

	R1		R2		S1		S2	
Flow	med	mean	med	mean	med	mean	med	mean
1	4.8	13.3	9.9	39.9	1.1	8.8	2.9	10.0
2	1.2	2.7	1.1	3.5	0.4	1.4	0.5	2.0
3	0.5	1.4	0.4	1.0	0.4	1.3	0.4	0.7
4	0.3	0.9	0.3	0.7	0.3	0.7	0.4	0.8

Table 6.3: The first number in each box in the median absolute error, $\text{med } |E_i|$, of the RTT estimate, and the second number is the mean absolute error, $\text{mean } |E_i|$. All entries are in milliseconds.

absolute error of 9.9 ms for flow 1 in simulation R2 was by far the largest, but it is still only a 5% relative error.

Due to the way in which the sender updates the congestion window the true value of $cwnd$ recorded by the simulator is not necessarily an integer. However, our $cwnd$ estimate is given by integer number of packets. This is, in a sense, the opposite of the problem with the true RTTs, they were measured with low precision, whereas $cwnd$ is measured with an artificial precision that is higher than our estimate. We fix this by replacing the nonintegral true congestion windows by the floor of their values. With the congestion window we again encounter the problem of not being able to directly compare the estimates to the true values. This is fixed the same way as before, by interpolating on a grid.

The errors between the real and approximated congestion windows are similar to that of the RTT errors with one important difference: the $cwnd$ estimates are

biased toward underestimating the true *cwnd*. The greatest median absolute error is 1.0 packets; this is again for flow 1 of configuration R2. The mean absolute error in this case was 3.0 packets. The median and mean errors for all other flows were between 0.4 and 1.1 packets, which correspond to relative errors of about 3%.

The estimates are biased because the true value is greater than the estimate about 80% of the time. The bias is caused by delayed ACKs, which prevent the flight sizes from being equal to *cwnd*. With both of our *cwnd* estimation methods we are in fact estimating the number of packets per flight and, in reality, this is a

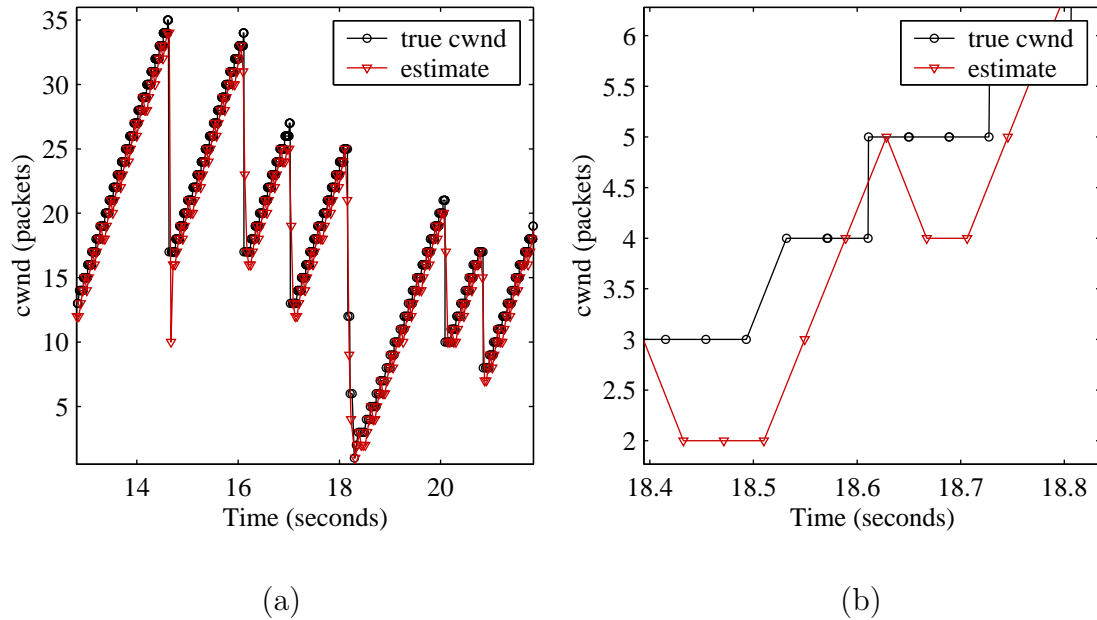


Figure 6.4: This plot is again for flow 3 of simulation R2. In (a) one sees that the estimate closely shadows *cwnd*. Zooming in on the area around 18.6 seconds we get plot (b), which shows how our estimate, the number of packets per flight oscillates as it increases. The oscillations follow a pattern that is determined by the even-odd parity of *cwnd*, but we are more concerned with tracking the flight boundaries.

lower bound on the true value of the congestion window. Overall, the number of packets per flight tracks the dynamics of *cwnd* very well, but at a small scale the flight sizes oscillate instead of monotonically increasing as *cwnd* does; this oscillation is illustrated in Figure 6.4.

The results of the real-time clustering algorithm are quite encouraging for simulator data. There are inherent difficulties and biases, but the algorithm still performs well. With simple nonlinear filtering the results should be even better. A median filter with a symmetric window of 3 samples is appropriate for the RTT estimate. As for the *cwnd* estimates a good choice is a max filter with a window of the current and two previous samples, but this filter should not be applied for 3 samples after a drop is detected.

6.1.2 Frequency Algorithm Validation

The first test of the frequency algorithm was to validate its method of estimating the RTT lower bound. We broke each flow into segments of 257 packets and computed the estimate of the RTT (\underline{r}) for each segment. The reason for testing this way is that at the start of real traffic traces most of the flows are already in progress, and may be in any stage of the congestion avoidance algorithm. The resulting estimate from each segment is compared to the minimum true RTT over the same segment. We consider the algorithm successful for a given segment if the estimate was less than the minimum true RTT, but within 15% of that value. Figure 6.5 shows the estimate and the minimum true round-trip times for all segments in

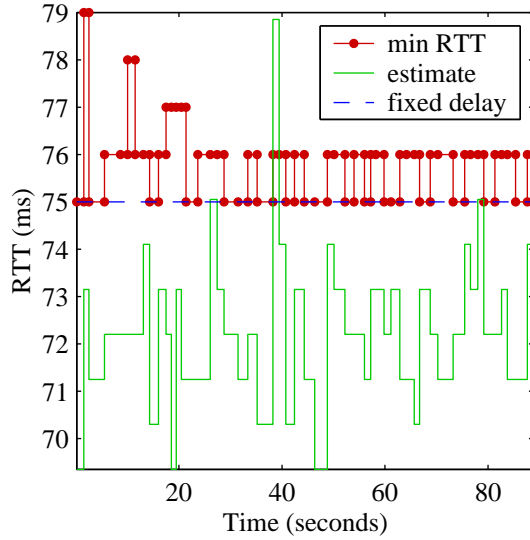


Figure 6.5: The interval $[0,90]$ was broken into 53 segments of 257 packets each. The lower curve is the RTT estimate, the upper curve is the minimum true RTT on each interval, and the dashed line is the fixed delay, which we are trying to estimate without exceeding. This plot is for flow 2 of simulation R2.

one flow. This particular flow was successful on 52 out of 53 segments for a success rate of 98%. Four out of sixteen simulated flows had a success rate of 100%. The median success rate was more than 98%, but the algorithm did perform better on network configuration S, where the median success rate was over 99%, whereas the rate was 95% for configuration R. For the purposes of estimating $cwnd$ it is best for $\underline{\tau}$ to be a slight underestimate, but it is not necessary since the congestion window estimation algorithm is designed to handle situations where the true RTT is less than the estimated value.

Upon obtaining an estimate of the RTT lower bound we proceed to estimate the sequence of flights using the method outlined in Section 5.4.6. The sequence of round-trip times are then estimated as the time from the start of one flight to the

start of the next flight. The results are nearly identical to the clustering algorithm, although the frequency algorithm performed better on flow 1 and worse on flow 2. All of the issues raised in the previous section still hold for the frequency algorithm including the bias of the *cwnd* estimate. The fact that the results of two very different algorithms agree so closely is validation in itself.

Our conclusion is that both algorithms work well on simulator data. But in practice we have found the frequency algorithm to be marginally more robust. The frequency algorithm also has an edge because it has a shorter initialization phase, and it requires a fixed number of packets.

6.2 RTT and Congestion Window Statistics

In this section, we study statistics related to round-trip times and congestion windows to determine how often congestion is present in the NLANR data. We also discuss factors affecting the rate of TCP connections and gauge their prevalence in real traffic.

The receiver's buffer can change throughout the course of a flow. This is a major concern in understanding the dynamics of the congestion window, since the receiver's buffer limits the window size. If the receiver is busy processing other data and cannot accept as much data from the sender, then the receiver may decrease the value of its buffer. The buffer may be set to 0 when the receiver can not accept any data at all. This usually has the effect of suddenly decreasing the transmission rate for a short period of time. This may resemble a decrease in the window size

due to a timeout, but the mechanism causing this decrease has nothing to do with network congestion.

We computed the maximum, median and minimum of the receiver's buffer from a sample of nearly 14000 bulk TCP flows from seven monitoring points in the month of December 2004. Let x_i , d_i , and n_i represent the maximum, median, and minimum receiver's buffer in bytes, respectively. For many flows the receiver's buffer is fixed, so that $x_i = d_i = n_i$, but any change can alter the transmission rate. If the median is less than the maximum, then the buffer changes often enough to potentially impact the transmission rate. We have found that $x_i > d_i$ for 11% of flows, but a more significant difference of one packet, $x_i - d_i \geq 1460$, occurs only 7% of the time. Additionally, $n_i = 0$ for 16% of the flows. Nearly 70% have $x_i > n_i$, meaning that the receiver's buffer changes at least once during the flow. Of course, many of these changes will not affect the rate at all.

Window scaling is another factor for the congestion window. We estimate that nearly 4% of the flows we studied used window scaling. Most of these flows were from the FRG and UFL monitors, where they accounted for up to 12% and 8% of the flows, respectively. We also attempted to recover the factor by which the windows were scaled. The most common scaling factor was 4, but most flows using window scaling had a receiver buffer of about 6 KB, resulting in a scaled window of only 24 KB, which could have been represented without window scaling. The largest scaled window we encountered was about 250 KB.

Figure 6.6 shows the distribution of RTT lower bound estimates from the frequency algorithm for the same sample of flows used above. It is perhaps slightly

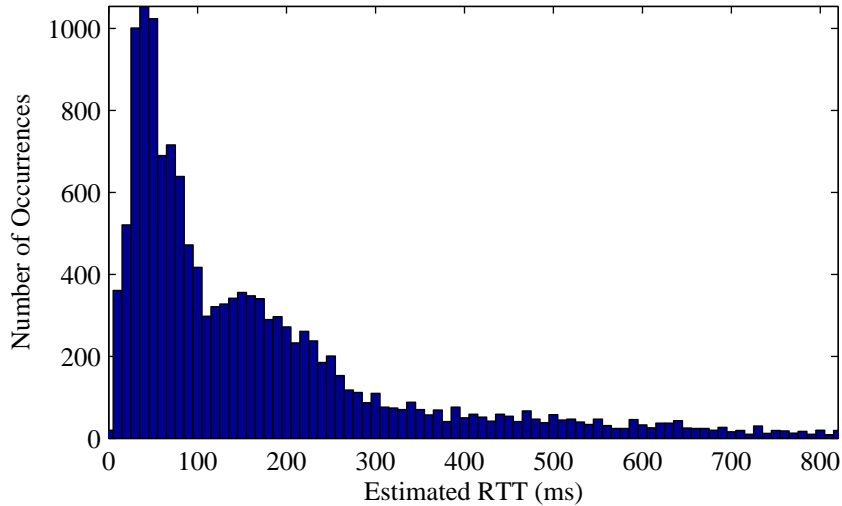


Figure 6.6: The main mode of the RTT distribution is at 45 ms, RTTs in this range correspond to flows that remain in North America. The secondary mode at 150 ms corresponds to flows that travel over transoceanic links.

misleading to accumulate all the monitoring points since they are geographically diverse, and geography plays a large role in determining RTT. Consider the three monitoring points that collect the most data, UFL at the University of Florida, FRG in Colorado, and MRA at Michigan State University. The most common RTT for these monitors are 33 at UFL, 50 at FRG, and 66 at MRA.

Because Figure 6.6 shows only the *minimum* RTT, we investigate how much the RTT changes over time. Aikat et al. [3] have studied the variability of RTTs. Depending on the amount of congestion, the RTT could fluctuate around its mean, with deviation above and below being equally likely, in which case the standard deviation is a better measure of variation. It could instead fluctuate around the fixed delay, with only positive deviations, making the median deviation from \underline{r} a better choice. We tested both deviations. Outliers were eliminated by discarding

samples greater than four times the median. Even then, the standard deviation was strongly influenced by outliers. When the estimated lower bound, \underline{r} , is considerably less than the true fixed delay, it is often an integer fraction of the fixed delay, hence the deviation from \underline{r} can result in an integer greater than one. Let \tilde{r}_i be the round-trip times estimated by the frequency algorithm, let \bar{r} be their median, and define

$$\rho = \frac{\text{median}(|\tilde{r}_i - \bar{r}|)}{\bar{r}}. \quad (6.1)$$

This deviation avoids the problems with the standard deviation and the deviation relative to \underline{r} . Figure 6.7 shows the distribution of ρ for 4000 bulk TCP flows from the UFL monitor. All remaining plots in this section use this data set.

Another measure of congestion for a given flow is how often the rate is below its maximum. Assuming the RTT is approximately constant, the rate is entirely

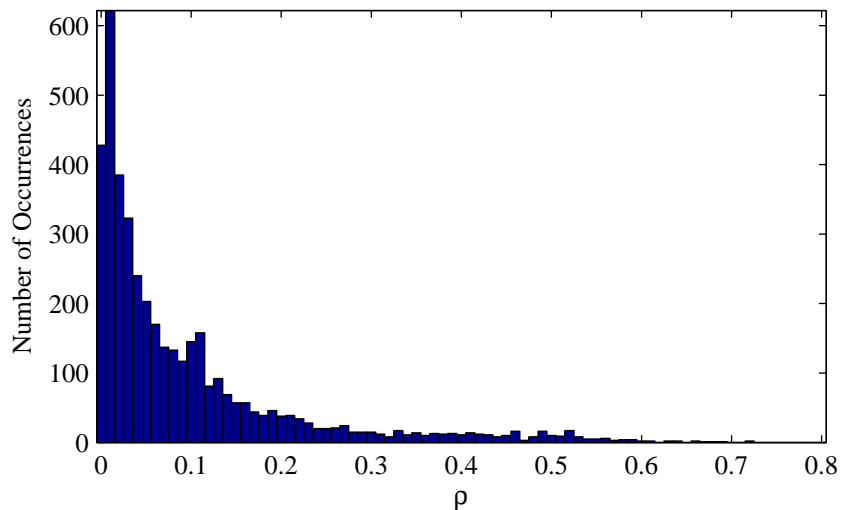


Figure 6.7: Distribution of observed RTT variation. The definition of ρ eliminates much of the noise that would result from using the RTT lower bound estimate in place of \bar{r} in Equation 6.1.

determined by the congestion window. The maximum rate is achieved when $cwnd$ equals the receiver's buffer. Let \tilde{w}_i be the congestion window estimates from the frequency algorithm, let \bar{w} be their median, and define

$$\kappa = \frac{\bar{w}}{M} \tag{6.2}$$

where M is the maximum receiver's buffer. This quantity will always be in the interval $[0, 1]$, and it is close to 1 when the estimated window is nearly equal to M for most of the flow's duration. Figure 6.8 shows the distribution of κ . Recall that \tilde{w}_i actually estimates the size of a flight, which underestimates $cwnd$. So if we plotted a histogram of the median value of $cwnd$ over M , then the plot would look similar to Figure 6.8, but more of the mass would be shifted toward 1.

Since \tilde{w}_i is slightly biased, it is better to use the congestion window estimate

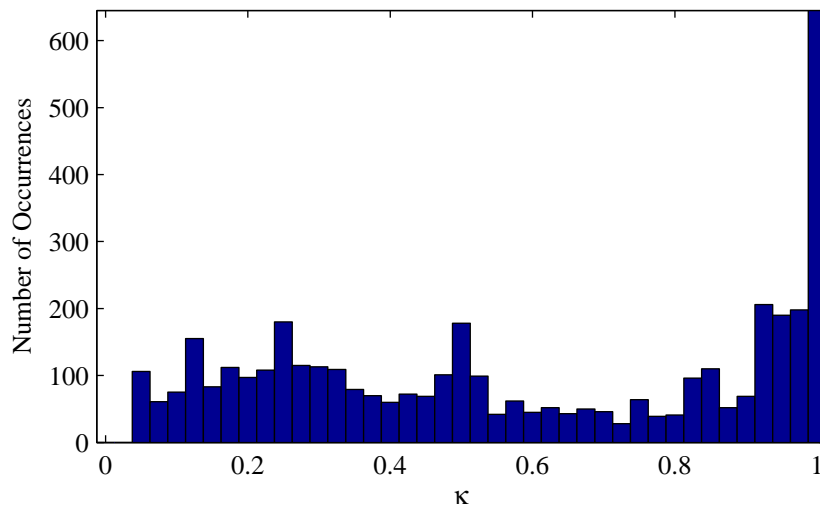


Figure 6.8: $\kappa = 1$ means that the $cwnd$ estimate is equal to M at least 50% of the time, implying that the sender is often transmitting near the maximum rate, with few dropped packets.

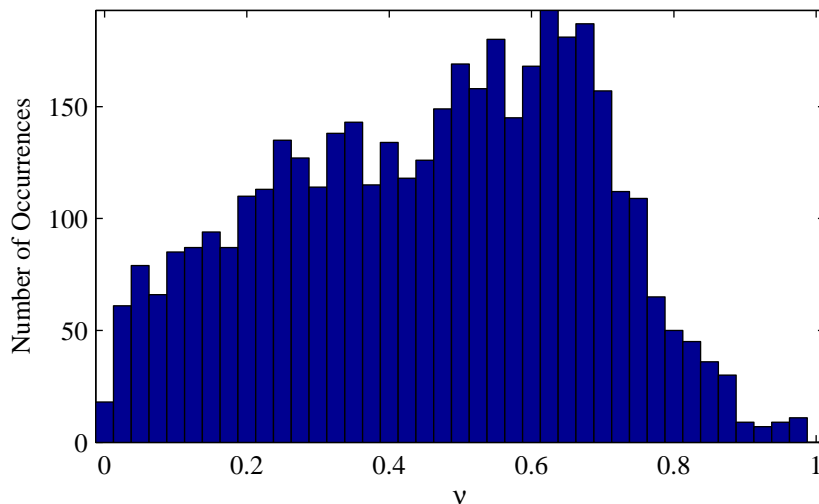


Figure 6.9: This plot shows that even though *cwnd* is often close to the receiver’s buffer, most flows can not maintain sending at the maximum achieved rate.

divided by the RTT estimate. If, at a given time, the RTT estimate is less than the true RTT, then the *cwnd* estimate will also be less than the true value, and the errors will cancel each other to some degree. Let $z = \max_i(\tilde{w}_i/\tilde{r}_i)$, and define

$$\nu = \frac{\langle z - \tilde{w}_i/\tilde{r}_i \rangle}{z}. \quad (6.3)$$

Figure 6.9 shows the distribution of ν .

6.3 Predicting TCP

We would like to use time series analysis and mathematical models to predict TCP flows over short time scales. This requires us to think of network traffic as a dynamical system. We could consider every flow on a network as part of the system, but for our purposes we will only consider the flows on one link. The rates of these flows will be the main elements of the dynamical system, and they will be coupled,

albeit weakly, by routers that the flows share.

We assume the state variables are the congestion window and round-trip time. There are obviously many other quantities that could be included, like router queue size, TCP timers and parameters, but the congestion window and round-trip time are the most important. Router queue size is not appropriate, because the paths of different flows pass through different routers. Instead, we propose more general measures of congestion based on the round-trip time and congestion window, see Appendix A.

As was stated in the introduction, there are deterministic and stochastic components of the network traffic. There is no clear line between the two, but as a general principle if a TCP flow lasts long enough to enter the congestion avoidance phase for a significant time, then it can be viewed as deterministic. The stochastic components are the short-lived flows and the flows that do not continually have data to transmit. Most web traffic is in this category, since most web pages and images are fairly small. As there is no clear distinction between deterministic and stochastic traffic, it is difficult to put a value on the proportion of the two types, but in the NLANR traces at least 30% of the traffic, and possibly much more, is deterministic. For that reason we study bulk TCP flows in the top 50%, as defined in Section 3.1.

6.3.1 Nonlinear Time Series Analysis

The concept of state space is fundamental to nonlinear time series analysis [24]. In network traffic modeling, the state variables are quantities like round-trip time, congestion window and router queue length. We have demonstrated novel techniques to infer round-trip times and reconstruct congestion windows, but we have no way to accurately estimate router queue lengths. The number of flows we include in our state vector will depend on the capacity of the link and the overall amount of traffic on the link. Consider OC3 with a moderate amount of traffic. We have found that the largest 10 to 100 bulk TCP flows can account for up to 50 percent of the total bytes in the aggregate traffic. Therefore, we combine the round-trip times and windows of the bulk TCP flows into a single state vector.

One of the remarkable successes of nonlinear time series analysis is its ability to deal with noisy or even missing state variables. It is our hope that this capability will allow us to predict flow rates without queue measurements. To understand why nonlinear time series analysis can handle incomplete state information, consider a frictionless pendulum. The state variables of this system are the angle and angular velocity. If we only measured the angle of the pendulum at evenly spaced time intervals, but not its velocity, then we would use the technique of time-delay embedding to predict the behavior of the system.¹ In the process of embedding, we

¹The time intervals should be short enough so that the system state does not drastically change in one interval. Time-delay refers to the opposite condition – the interval is so short that the state barely changes. In this case we would use every, say, tenth measurement. We will assume that the measurements are taken at appropriate intervals, so we will only discuss the embedding process.

take as our alternative state vector, known as a *delay vector*, the angles at times t and $t + 1$. This vector provides the necessary state information since the difference between the angles is proportional to the velocity. In the same manner, it is likely that router queue length is a function of the congestion windows.

Consider an m -dimensional discrete time dynamical system $x_{i+1} = f(x_i)$, and a scalar measurement of the system $s_i = g(x_i)$. To approximate the true state of the system, we construct a delay vector $y_i = (s_i, s_{i-1}, \dots, s_{i-n+1})$, where the length of the vector, n , is the *embedding dimension*. The idea is to make the embedding dimension large enough so that two different delay vectors correspond to different states of the true system. If the dynamical system x has an attractor, then under what circumstances does the reconstructed system y have an attractor that is equivalent up to a change of coordinates? In general, if the attractor for the true system x has dimension² d , then an embedding dimension $n > 2d$ is guaranteed to give an equivalent attractor for the system y .

To better understand the embedding process and how to determine an appropriate embedding dimension, suppose we choose an embedding dimension $n = 2$. Suppose we then plot the time series y_i in the plane, and the attractor of the system y is a curve shaped like a figure eight. If we choose two points y_i and y_j near the intersection of the curve then it is possible that the forward images, y_{i+1} and y_{j+1} , of the points will be on different sides of the figure eight. That contradicts the basic assumption of continuity of the dynamical system. If instead we choose $n = 3$ as

²By dimension we mean the box-counting dimension. If the attractor is fractal then this number might not be an integer.

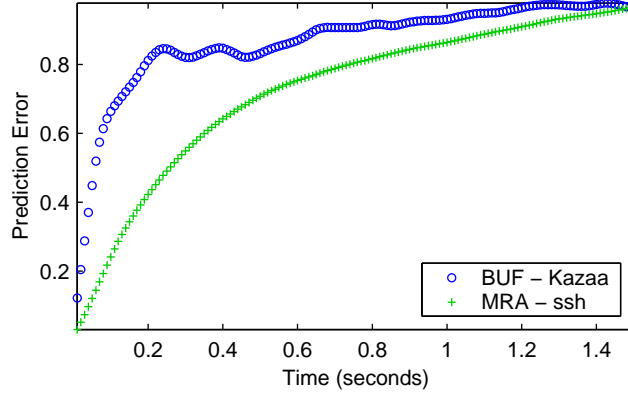


Figure 6.10: Zeroth-order prediction errors for the example flows used in Figures 2.2 and 5.3.

the embedding dimension, then the self-intersection will almost surely be resolved, and the curve will be a non-intersecting loop in three dimensions. Once the embedding dimension is large enough, there will be no points where continuity fails, and consequently, the system y will have the same information content as x .

Upon determining a proper embedding dimension we use the following prediction algorithm known as zeroth-order prediction. Suppose we have a past history of k vectors, y_1 through y_k , and we want to predict the next value, y_{k+1} . Suppose that the vectors have a measurement error of ϵ . Since the current vector is y_k , we find the vectors in the past history within ϵ of y_k ; call this collection of vectors A . This collection of vectors will be measurements of the system x when the system was close to its current state, and because of the determinism of the system we expect y_{k+1} to be close to the forward images of A . Therefore, we use the average of the forward images of A as a predictor of y_{k+1} .

Under the assumption that the flows are weakly coupled we can treat them

as separate dynamical systems. We have found that round-trip times usually vary randomly, and therefore they add little state information; as such, we use only the observed congestion windows in the delay vector. Figure 6.10 shows the results for the two example flows used in this paper. These predictions are made in-sample, meaning that predictions are made for all data points using the entire history, instead of just the past history. Out-of-sample prediction is preferable, but in this case we have a limited amount of data, and we only want to see if the time series is predictable at any level. We used $n = 2$ as the embedding dimension for the scalar time series. The plot shows the average prediction error for all points normalized by the standard deviation of the time series. After one second the predictions are little better than guessing the mean of the time series.

Our investigation using time series analysis indicates that network traffic, when considered as a dynamical system, is not very predictable. This was to be expected since dropped packets cause discontinuous changes in the dynamics of flows, and the drops are determined by an inherently random process in routers using RED. This implies that we need to incorporate mathematical models or develop new empirical models in order to predict traffic.

6.3.2 Incorporating Models

Some mathematical models of TCP [15, 30, 34] start with the following assumptions: (i) one sender, one router, one receiver; (ii) the congestion control algorithm is RED; (iii) three state variables: sender's congestion window, router's

instantaneous queue and exponentially weighted average queue. The models start with this scenario, then increase in complexity until they simply mimic all the technical details of TCP [17]. It would be nearly impossible to directly use such a complicated model to predict the state of the network, since there are so many variables and parameters in the model.

A better approach is to use a simpler model with fewer variables. Assuming one can reliably track a given sender's congestion window, what more can one infer from traffic measurements that will help to predict the dynamics of TCP? Measuring the queues of all routers on the path would be best, but no such traces are available. Inferring queue lengths at any routers other than the two at the ends of the monitored link seems dubious. The best we can do is to find some variable that indicates of the overall level of congestion experienced by each flow.

Even though network traffic is not amenable to the direct use of nonlinear time series analysis, we believe more accurate prediction is still possible via a synthesis of time series analysis and a simple mathematical model. The model would overcome the inability of standard time series analysis techniques to cope with the inherent discontinuities in the rate of TCP flows. The real difficulty is predicting when the drops are likely to occur. We think the most promising way to do this is to find and incorporate an appropriate measure of congestion, in absence of router queue measurements. We discuss several possibilities in Appendix A.

Chapter 7

Conclusion

Understanding the dynamics of network traffic is important for the design and implementation of TCP and routers. We approach the issue from the perspective of dynamical systems. Our initial goal was to predict the behavior of large, persistent flows from network trace data. In order to do so we needed to infer round-trip times and congestion windows. Our novel algorithms for estimating these rate-controlling variables are quite flexible and robust. While we have explained why simple nonlinear time series analysis methods are unsuited to the prediction of network traffic, it is still beneficial to think of TCP traffic in terms of dynamical systems.

Appendix A

Drop Prediction

In this appendix, we introduce several statistics related to congestion, and use them to predict drops. By congestion, we mean that the transmission rate for a given flow is constrained by the network to be less than the maximum receiver's buffer over the fixed delay time. One might assume that since the RTT reflects the total queuing time, it would be the best predictor of congestion. However, as the example in Figure 5.15 shows, the RTT can remain unchanged even when queuing occurs. Realistic situations will be more complicated, but the basic idea is that the RTT will not vary significantly if the congested tight link has a much larger capacity than the narrow link.

We define a collection of eight key statistics, each of which should correlate with congestion to some degree. Although it is a precursor to dropped packets, congestion does not necessarily lead to drops. Since we are concerned more about drops than congestion, we test the statistics to determine which of them most closely corresponds to actual drops. This is accomplished by finding the linear combination of the eight statistics that best correlates with a drop indicator function.

A.1 Congestion Statistics

The statistics are calculated once per flight. We denote the eight statistics for flight i as $\chi_{i,j}$, $j = 1 \dots 8$. Suppose we have n samples of the eight statistics, then we can consider χ as an $n \times 8$ matrix. We will sometimes drop the first index on χ and simply refer to the j th statistic as χ_j , and by dropping the second index we refer to the i th sample of the statistics as χ_i . The two most important statistics are the estimated congestion window and RTT, so we set $\chi_{i,1} = \tilde{w}_i$ and $\chi_{i,2} = \tilde{r}_i$. The third quantity, χ_3 , is the normalized flight duration, which is defined as the time from the first packet in a flight to the last packet in a flight divided by the estimated RTT. The ratio of flight duration to RTT is a measure of how much of the available bandwidth the flow is consuming. If two flights have the same RTT and the same number of packets, but one has a longer duration, then that flight likely experienced more congestion up to the monitoring point. How much the flight duration reflects congestion depends on where the monitor is along the path. If the monitor is closer to the sender, then we are less likely to see the effects of congestion on the spacing of the data packets than when the monitor is closer to the receiver. However, the self-clocking nature of the TCP congestion avoidance algorithm should impose similar spacing on the data and acknowledgments no matter where congestion occurs.

The remaining five quantities are related to the logarithm of the interpacket times for flight i . We only consider the interpacket times *within* the flight boundaries, so for a flight with n packets we will use $n - 1$ interpacket times. When a flight has only one packet the following quantities are zero by default. We let χ_4 through χ_6 be

the mean, median and standard deviation of the logarithm of the interpacket times. It is clear that the mean and median are proportional to the flight duration. As bursts of cross traffic become interleaved with packets of a given flow, the interpacket times will increase or decrease commensurately with the amount of cross traffic. The cross traffic bursts are likely random in size, and this will increase the standard deviation of the interpacket times.

The final two statistics expand on the idea that cross traffic disperses packets and increases the randomness of interpacket times. We create a histogram of the interpacket times with 100 exponentially spaced bins from 10^{-6} to 10^0 seconds. Let h_k be the fraction of interpacket times in the k th bin. Most bins will be empty, since there is usually a maximum of 44 packets per flight. We let $\chi_{i,7}$ equal the entropy of the interpacket times in the i th flight, which is defined as

$$H_i = - \sum_{h_k > 0} h_k \log_2(h_k). \quad (\text{A.1})$$

After the last packet in flight i , we compute a histogram of all interpacket times up to that point, using the bins defined above. Let \hat{h}_k be the values of this histogram. Suppose the current flight of n packets has interpacket times δ_m , which fall into bins k_m . We then define

$$\chi_{i,8} = \frac{1}{n-1} \sum_m \log(\hat{h}_{k_m}). \quad (\text{A.2})$$

This statistic reflects the probability of a given sequence of interpacket times relative to the global distribution. The reason for defining this statistic is that when there is no congestion, the majority of interpacket times will be approximately equal to the most common NIT, but when a flow is experiencing congestion the cross traffic

will spread the interpacket times to less common values.

A.2 Drop Correlation

We test our hypothesis that the eight statistics should be correlated with congestion by using data from our *ns2* simulations. We focus on simulation S1, because there is only one router at which packets are dropped in that simulation. The simulator records the instantaneous queue length, average queue length, and RED drop probability every time the queue changes. In order to compare these variables to $\chi_{i,j}$, we interpolate them at times corresponding to the end of each flight. We denote interpolated average queue length and RED drop probability by the variables AQ_i and DP_i , respectively.

The correlation coefficient for two time series x_i and y_i , with means \bar{x} and \bar{y} , and standard deviations σ_x and σ_y , is defined as

$$C(x, y) = \frac{\langle (x_i - \bar{x})(y_i - \bar{y}) \rangle}{\sigma_x \sigma_y}. \quad (\text{A.3})$$

For a given statistic, χ_j , we define congestion correlation as the maximum of $C(AQ, \chi_j)$ and $C(DP, \chi_j)$. Of the eight statistics, we found that RTT, flight size and flight duration usually had the greatest congestion correlation – about 0.2 to 0.5. However, each of the eight statistics had a congestion correlation greater than 0.1 for at least one of the simulated flows. It is hard to say whether these correlation coefficients are statistically significant, since the statistics we are studying are not normally distributed. For random time series of the same length and with the same distribution, we found that the correlation coefficient was greater than 0.1 about 2% of the time.

We can not compare our statistics to the average queue length or drop probability for the the real data. Therefore, we compare them to the drops via a drop indicator function. Let t_k^* be the times at which the monitor learns of dropped packets, that is, when the monitor sees three duplicate ACKs or a retransmitted data packet. The drop indicator function is defined as

$$D(t) = \begin{cases} 1 & \text{if } \exists i \text{ s.t. } t_k^* - 5\mathcal{L} < t < t_k^* - \mathcal{L} \\ 0 & \text{otherwise} \end{cases} \quad (\text{A.4})$$

where \mathcal{L} is the estimated minimum RTT. This function acts as an “oracle” that knows when drops will occur. If a statistic correlates very well with the drop indicator function, then we could accurately predict when drops will occur. Arguably, there are better ways to define the drop indicator function, perhaps linearly increasing from zero to one over an interval longer than $(t_k^* - 5\mathcal{L}, t_k^* - \mathcal{L})$. But as Figure A.1 shows, Equation A.4 suitably captures the precursors to drop events.

Let D_i equal $D(t)$ sampled at the end of flight i . We define drop correlation for statistic χ_j as $C(D, \chi_j)$. As with the congestion correlation, we found that RTT, flight size and flight duration usually had the largest drop correlations, although the other statistics often had non-negligible drop correlations as well. Note that even if the drop correlation is negative for a particular χ_j , that statistic can still be of use in predicting drops, since $-\chi_j$ will be positively correlated.

Since the drop correlation for a statistic varies from flow to flow, the individual statistics, χ_j , when considered by themselves, are not reliable predictors of drops. Therefore, we propose to use a linear combination of χ_j as a more robust predictor

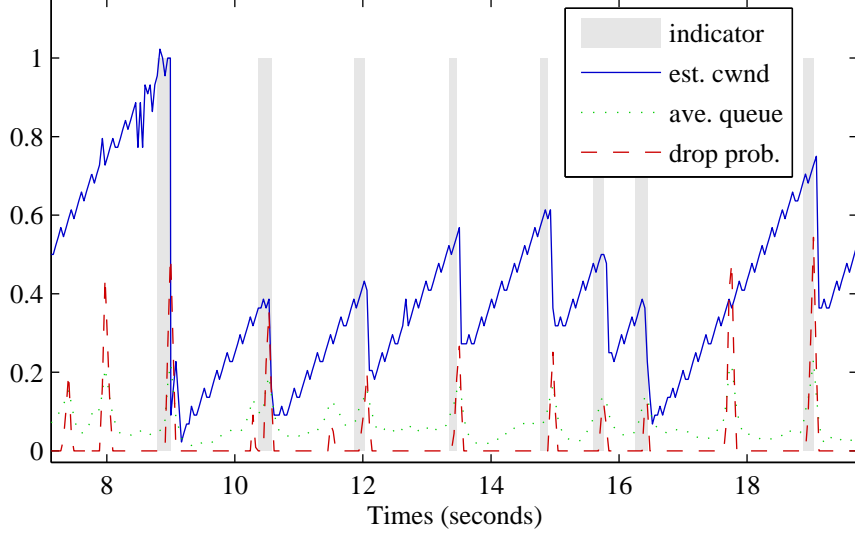


Figure A.1: This plot shows congestion variables for flow 3 in simulation S1. The gray bars mark where the drop indicator function equals 1. Note that these correspond to times where AQ is increasing. However, AQ and DP can increase without causing a drop in the flow. This happens at 8 and 18 seconds. At these times packets are dropped in cross traffic flows, but not this flow.

of drops. More formally, we find $c \in \mathbb{R}^8$ that maximizes

$$\gamma(c) = C(D, \sum_{j=1}^8 c_j \chi_j). \quad (\text{A.5})$$

We frame this as a Lagrange multiplier problem by introducing the constraint

$$\sum_{i=1}^n [c \cdot (\chi_i - \bar{\chi})]^2 = 1, \quad (\text{A.6})$$

where $\bar{\chi}$ is the mean of the n samples vectors, χ_i , $i = 1 \dots n$. We can impose this constraint without loss of generality, since $\gamma(c)$ is invariant under scalar dilations of c . Based on our objective function in Equation A.5 and the constraint in Equation

A.6, the Lagrange multiplier λ satisfies

$$\sum_{i=1}^n (D_i - \bar{D})(\chi_i - \bar{\chi}) = \lambda \sum_{i=1}^n 2(\chi_i - \bar{\chi}) [c \cdot (\chi_i - \bar{\chi})], \quad (\text{A.7})$$

where \bar{D} is the mean of the scalar time series D_i . To solve for c let

$$A = \sum_{i=1}^n (\chi_i - \bar{\chi})^T (\chi_i - \bar{\chi}) \quad (\text{A.8})$$

and let

$$b = \sum_{i=1}^n (D_i - \bar{D})(\chi_i - \bar{\chi}), \quad (\text{A.9})$$

so that Equation A.7 is equivalent to

$$b = 2\lambda A c. \quad (\text{A.10})$$

Since the normalization of c does not matter let

$$c^* = A^{-1} b \quad (\text{A.11})$$

be the solution to the Lagrange multipliers problem. The optimal drop correlation, $\gamma(c^*)$, is compared to the maximum individual drop correlation in Table A.1.

flow	1	2	3	4
$\max_j C(D, \chi_j)$	0.25	0.49	0.26	0.21
$\operatorname{argmax}_j C(D, \chi_j)$	2	2	1	1
$\gamma(c^*)$	0.30	0.52	0.30	0.26

Table A.1: Drop correlations for four simulated flows. The third row is the index j for which the maximum individual drop correlation is attained. Flight size is denoted by 1, RTT is denoted by 2.

To test the utility of the drop correlation for real traffic, we selected nearly 400 flows from the UFL monitor that were clearly experiencing congestion. Figure A.2(a) shows the distribution of $\gamma(c^*)$ for these flows. Figure A.2(b) shows the number of times $\max_j C(D, \chi_j)$ is attained by statistic j . On average $\gamma(c^*)$ is 1.5 times greater than $\max_j C(D, \chi_j)$. For 92% of flows $\gamma(c^*)$ is statistically significant, that is, it is greater than the correlation obtained by replacing the statistics χ_j with random variables having the same distribution.

Unfortunately, this is not a feasible method for predicting drops. The coefficients of $\gamma(c^*)$ vary from flow to flow, and while the correlation coefficients are statistically significant, they are not large enough to make accurate predictions.

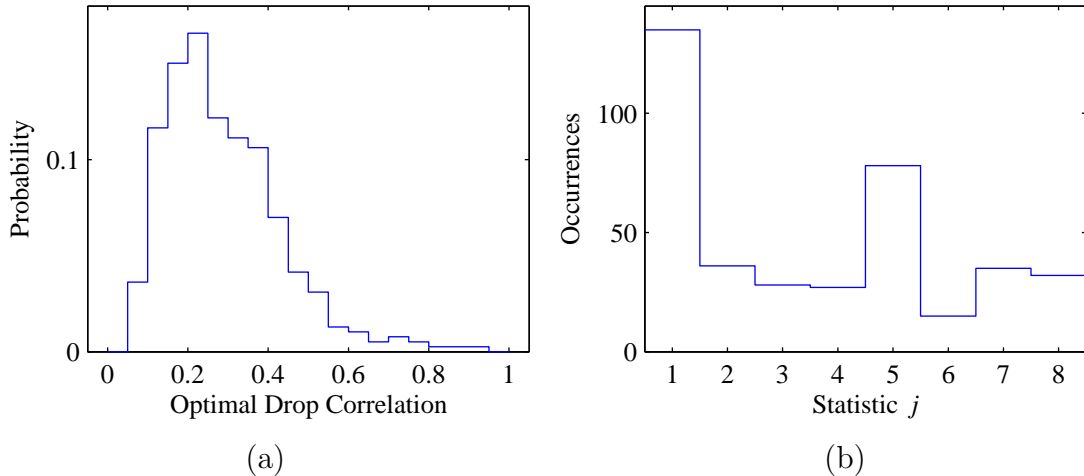


Figure A.2: (a) Distribution of the optimal drop correlation, $\gamma(c^*)$. The mean correlation coefficient is 0.29. (b) Histogram of $\operatorname{argmax}_j C(D, \chi_j)$. The most common value is $j = 1$, which corresponds to the flight size. Next is $j = 5$, corresponding to the median interpacket time. This deviates from the simulated flows where the RTT or flight duration often had the largest drop correlation. The least common value is $j = 6$, corresponding to the standard deviation of the interpacket times.

However, this method does indicate how certain statistics relate to congestion and drops. One could improve this method by finding a single linear combination that is optimal for a wide variety of flows. One could also include other statistics, such as time since the last drop and quantities similar to interpacket time statistics computed instead for interack times. A more promising approach is to derive a nonlinear function using more sophisticated statistical techniques.

BIBLIOGRAPHY

- [1] Abilene Backbone Network. <http://abilene.internet2.edu>, 2005.
- [2] Active Measurement Project. National Laboratory for Applied Network Research. <http://amp.nlanr.net>, 2005.
- [3] J. Aikat, J. Kaur, F. D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proc. of the ACM SIGCOMM Internet Measurement Conference*. ACM, October 2003.
- [4] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *Proc. of the ACM SIGMETRICS*, pages 316–317. ACM Press, 2003.
- [5] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control. <http://www.rfc-editor.org/rfc/rfc2581.txt> , 1999.
- [6] C. Barakat. TCP/IP modeling and validation. *IEEE Network*, pages 38–47, May/June 2001.
- [7] N. Brownlee and K. Claffy. Understanding internet traffic streams: Dragonflies and tortoises. *IEEE Communications Magazine*, 40(10):110–117, October 2002.
- [8] CAIDA measurement tools. <http://www.caida.org/tools>, 2005.
- [9] J. Cavanaugh. Protocol overhead in IP/ATM networks. Technical report, Minnesota Supercomputer Center, 1994.

- [10] C. R. Simpson, Jr. and G. F. Riley. Neti@home: A distributed approach to collecting end-to-end network performance measurements. In *Proc. of the Passive and Active Measurement Workshop*, April 2004.
- [11] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking*, 12(6):963–977, 2004.
- [12] P. Erdős, R. Faudree, J. Pach, and J. Spencer. How to make a graph bipartite. *J. Comb. Theory Ser. A*, 45(1):86–98, 1988.
- [13] P. Erdős, A. Gyárfás, and Y. Kohayakawa. The size of the largest bipartite subgraphs. *Discrete Math.*, 177(1-3):267–271, 1997.
- [14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.
- [15] I. Frommer, B. Hunt, E. Harder, R. Lance, E. Ott, and J. Yorke. Modeling congested internet connections. Technical report, UMD Institute for Physical Science and Technology, 2003.
- [16] M. Goyal, R. Geurin, and R. Rajan. Predicting TCP throughput from non-invasive network sampling. In *Proc. of the IEEE INFOCOM*. IEEE, 2002.
- [17] J. P. Hespanha, S. Bohacek, K. Obraczka, and J. Lee. Hybrid modeling of TCP congestion control. *Lecture Notes in Computer Science*, 2034, 2001.

- [18] P. Huang, A. Feldmann, and W. Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *Proc. of the ACM SIGCOMM Internet Measurement Workshop*. ACM, 2001.
- [19] V. Jacobson, R. Braden, and D. Dorman. RFC 1323: TCP extensions for high performance. <http://www.rfc-editor.org/rfc/rfc1323.txt>, 1992.
- [20] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Inferring TCP connection characteristics through passive measurements. In *Proc. of the IEEE INFOCOM*. IEEE, 2004.
- [21] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *Computer Communications Review*, July 2002.
- [22] H. Jiang and C. Dovrolis. Source-level IP packet bursts: Causes and effects. In *Proc. of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 301–306. ACM Press, 2003.
- [23] H. Jiang and C. Dovrolis. The effect of flow capacities on the burstiness of aggregated traffic. In *Proc. of the Passive and Active Measurement Workshop*, April 2004.
- [24] H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, 1997.
- [25] D. Katabi, I. Bazzi, and X. Yang. A passive approach for detecting shared bottlenecks. In *Proc. of the IEEE International Conference on Computer Communications and Networks*. IEEE, 2001.

- [26] D. Katabi and C. Blake. Inferring congestion sharing and path characteristics from packet interarrival times. Technical report, MIT Laboratory for Computer Science, 2001.
- [27] W. Leland, M. S. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of Ethernet traffic. In *Proc. of the ACM SIGCOMM*. ACM, 1993.
- [28] N. R. Lomb. Least-squares frequency analysis of unequally spaced data. *Astrophysics and Space Science*, 39:447–462, 1976.
- [29] Measurement and Analysis on the WIDE Internet (MAWI). <http://tracer.csl.sony.co.jp/mawi>, 2004.
- [30] V. Misra, W. Gong, and D. Towsley. Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In *Proc. of the ACM SIGCOMM*. ACM, 2000.
- [31] Network Reseach Group, Lawrence Berkeley National Laboratory. <http://ee.lbl.gov>, 2005.
- [32] S. Northcutt and J. Novak. *Network Intrusion Detection*. New Riders Publishing, 2003.
- [33] ns2 Network Simulator. <http://www.isi.edu/nsnam/ns>, 2000.
- [34] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of the ACM SIGCOMM*, pages 303–314, Vancouver, CA, 1998.

- [35] V. Padmanabhan, S. Ramabhadran, and J. Padhye. Netprofiler: Profiling wide-area networks using peer cooperation. Technical report, Microsoft Research, 2005.
- [36] C. Partridge, P. Barford, D. Clark, S. Donelan, V. Paxson, J. Rexford, and M. Vernon. *The Internet Under Crisis Conditions: Learning from September 11*. The National Academies Press, 2003.
- [37] Passive Measurement and Analysis archive. National Laboratory for Applied Network Research. <http://pma.nlanr.net>, 2005.
- [38] Pathrate measurement tool. <http://www.pathrate.org>, 2005.
- [39] V. Paxson and S. Floyd. The failure of Poisson modeling. In *Proc. of the ACM SIGCOMM*. ACM, 1994.
- [40] R. S. Prasad, M. Jain, and C. Dovrolis. Socket buffer auto-sizing for high-performance data transfers. Technical report, Georgia Tech, 2003.
- [41] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.
- [42] D. Rubenstein, J. Kurose, and D. Towsley. Detecting shared congestion of flows via end-to-end measurement. In *Proc. of the ACM SIGMETRICS*, pages 145–155. ACM Press, 2000.

- [43] S. Sarvotham, R. Riedi, and R. Baraniuk. Connection-level analysis and modeling of network traffic. In *Proc. of the ACM SIGCOMM Internet Measurement Workshop*. ACM, 2001.
- [44] J. D. Scargle. Studies in astronomical time series, II. Statistical aspects of spectral analysis of unevenly spaced data. *Astrophysical Journal*, 263:835–853, 1982.
- [45] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. In *Proc. of the ACM SIGCOMM*, pages 315–323. ACM Press, 1998.
- [46] S. Shakkottai, N. Brownlee, and K. Claffy. A study of burstiness in TCP flows. In *Proc. of the Passive and Active Measurement Workshop*, April 2005.
- [47] Y. Tsang, M. Coates, and R. Nowak. Passive unicast network tomography based on TCP monitoring. Technical report, Rice University, ECE Department, 2000.
- [48] Unidata-LDM. University Corporation for Atmospheric Research <http://www.unidata.ucar.edu/packages/lDM>, 2005.
- [49] WAND Network Research Group. University of Waikato. <http://wand.cs.waikato.ac.nz>, 2005.
- [50] E. Weigle and W. Feng. Dynamic right-sizing: A simulation study. In *Proc. of the ICCCN*. IEEE, 2001.

- [51] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proc. of the ACM SIGCOMM*, pages 309–322. ACM Press, 2002.