MEASUREMENT AND ANALYSIS OF END-TO-END DELAY AND LOSS IN THE INTERNET

A Dissertation Presented

by

SUE B. MOON

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2000

Department of Computer Science

MEASUREMENT AND ANALYSIS OF END-TO-END DELAY AND LOSS IN THE INTERNET

A Dissertation Presented

by

SUE B. MOON

Approved as to style and content by:

James F. Kurose, Co-chair

Donald F. Towsley, Co-chair

Krithivasan Ramamritham, Member

Christopher Raphael, Member

Ramón Cáceres, Member

James F. Kurose, Department Chair
Department of Computer Science

Dedicated to my parents,

Pyung Wan Chung and Kuk Jin Moon

ACKNOWLEDGMENTS

The past seven years at UMass has been the most challenging and exciting moment in my life. I learned so much academically and also in personal life that I am happy that I made the decision to come to UMass. I have acquainted many people on the way and I have benefited tremendously from the interaction with them. I would like to use this opportunity to express my gratitude to them.

I would like to first thank my advisors, Professors Jim Kurose and Don Towsley for their guidance throughout my Ph.D. They have taught me not only the rigor and beauty of science, but also the importance of honesty, integrity, and hard work as a researcher by their own example. They have been a constant source of inspiration when I needed direction, and of encouragement when I was weary. It is with my deepest gratitude that I acknowledge them.

I have learned greatly from working with Professor Christopher Raphael. He has shared his insightful and nurturing ideas with me. I am grateful to have him on my thesis committee.

Professor Krithi Ramamritham has taught me in classroom and also as my thesis committee member. His comments always made me think about problems in many different perspectives, and enlarged my own view.

Ramón Cáceres was my mentor when I was a summer intern at AT&T Research Labs, and later on served on my thesis committee. I value very much how considerate he has been as my mentor. His erudition and expertise in research have helped and stimulated me in my own research.

Paul Skelly has always been a wonderful and engaging friend. We worked on one of the most interesting problems of my thesis together, and I enjoyed every single moment of it filled with his sagacity and humor.

I spent a summer at GTE Labs under Adrian Conway's mentorship, and was given an opportunity to continue the collaboration with GTE Labs back at UMass. I thank Adrian Conway for his guidance and support through it all.

I also had the opportunity and privilege to work with many professors and researchers at school and in the research community. Specifically, I would like to thank Nick Duffield, Professor Joseph Horowitz, and Vern Paxson for sharing their time and expertise with me.

Betty Hardy and Sharon Mallory of the UMass Computer Science Department have been the single source of reliability and support whenever I needed them. I know that I can never thank them enough.

The Advance Networking Group at UMass has been the bedrock of friendship and nurturing environment for a fledgling researcher. With their support, I survived those seven years in graduate school. Those who left UMass include: Henning Schulzrinne, Ramesh Nagarajan, Erich Nahum, Zhi-Li Zhang, Jim Salehi, David Yates, Ramachandran Ramjee, and Victor Firoiu. For all the jokes, arguments, discussions, and more, I cherish the last few years spent with lab members: Maya Yajnik, Supratik Bhattacharyya, Sneha Kasera, Sambit Sahu, Jitu Padhye, Dan Rubenstein Timur Friedman, and Tian Bu. Our group has had several visitors over the years, who became very good friends. I would like to thank two in particular, Francesco LoPresti and Olov Schelén, for sharing their worldly wisdom with me, and also enriching my research life with their insights.

I had the privilege to be an honorary member in two other groups at UMass. The theory group with Kousha Etessami, Sushant Patnaik, Lixin Gao, Rajesh Prabhu, Lewis McCarthy, Bill Hesse, and Matt Greene, served as good references when I needed their strength in theory and algorithms, and were good buddies when I simply needed to have fun. The robotics group of Jefferson Coelho, Elizeth Araujo, Manfred Huber, and I shared

many late nights at school, at the movie theaters, on the running trails, and at our homes. They made Amherst a second home for me.

I would like to thank Ann Desmond, who has been my housemate for six years for her friendship and help. She spent many evenings helping me with writing and sharing her ideas about almost everything. I leave Amherst with her friendship, and I will always be very grateful for that.

Many more friends helped and made my life in Amherst great fun. I cherish all those funny emails exchanged over the Atlantic with Klaus Schossmaier, sunny days on the soccer field with Ivon Arroyo and Agustin Schapira, the longest run of my life with Dana Nakano Laird, and goofy jests and horizon-expanding trips with Rodrigo Coelho. Keunwon Chung, Whuiyeon Jin, Eunjung Ana Chai, and Yoosoon Chang have been the same old good friends throughout my Ph.D., and I thank them for their enduring friendship.

I was luckier than most other foreign students to have close relatives nearby. My aunt and uncle, Ae-Jin Kang and Young Mo Kang, and their children have taken me under their wings, and gave me their constant support, love, and homemade Kimchi. It is with my deepest gratitude that I acknowledge them in my thesis.

My brother, Cho Woo Moon, and sister, Hyuna Irene Moon, have always been there when I needed the sibling chat through good and bad times.

My parents, Pyung Wan Chung and Kuk Jin Moon, have imbued me with their enthusiasm about life, and in their unending love, I live. With all my heart I dedicate this dissertation to them.

ABSTRACT

MEASUREMENT AND ANALYSIS OF END-TO-END DELAY AND LOSS IN THE INTERNET

FEBRUARY 2000

SUE B. MOON

B.S., SEOUL NATIONAL UNIVERSITY, SEOUL, KOREA

M.S., SEOUL NATIONAL UNIVERSITY, SEOUL, KOREA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor James F. Kurose and Professor Donald F. Towsley

Measurement and analysis of the network behavior are crucial to understanding the Internet performance and designing appropriate control mechanisms for better performance. End hosts and their applications, however, have a limited capability in accessing and acquiring information about the network behavior. To them, end-to-end measurement of the network behavior is usually the only available information. This thesis focuses on two fundamental measures of network performance: end-to-end packet delay and loss.

First, we address the issue of accuracy in end-to-end delay. Raw delay measurements in the Internet contain impairments due to unsynchronized clocks between two measuring hosts. We propose a linear programming (LP) based algorithm to estimate and remove clock skew in delay measurements. We compare the LP-based algorithm with three other

algorithms, and show that the LP-based algorithm is robust, and performs well over actual delay measurements and in simulation.

Next, we consider the problem of adaptively adjusting the playout delay at the receiver of packet audio applications. We first present efficient algorithms that compute a bound on the achievable performance of any playout delay adjustment algorithm, and a new adaptive playout delay adjustment algorithm that tracks the network delay of recently received packets and efficiently maintains delay percentile information.

We also look at the correlation between end-to-end delay and loss. We quantify the correlation as sample mean delay conditioned on loss and loss conditioned on delay, and analyze the measurements based on them. The results indicate that it is likely that the packet delay would increase in the near future if a packet loss is detected.

Recent developments from the MINC (Multicast-based Inference of Network-internal Characteristics) project have shown that we can attain asymptotically converging estimates of link loss using Maximum Likelihood Estimators (MLE) from end-to-end multicast measurements. We validate the effectiveness of the MLE algorithm by showing that the inferred link loss estimates are close to the actual link loss inside the network. Also we study the performance of MLE estimates given a limited number of packets and under a wide range of loss rates and tree topologies.

We conclude this dissertation with a discussion for future research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Overview

In the past few years we have witnessed a significant growth in the Internet in terms of the number of hosts, users, and applications. The success in coping with the fast growth of the Internet rests on the IP (Internet Protocol) architecture's robustness, flexibility, and ability to scale. The underlying end-to-end paradigm of the IP architecture [49, 9] is that the end hosts are responsible for the recovery of lost packets, reordering of out-of-order packets, and flow and congestion control. This architectural principle is embodied in the main transport protocol of the Internet, TCP (Transport Control Protocol) [25]. This has proven to be crucial the success of the Internet.

Backed by availability of high bandwidth, in the order of Gb/s on the backbone, of hundreds of Mb/s on local area networks (LANs), and of Mb/s on the last hop to the users using Asymmetric Digital Subscriber Line (ADSL) and cable modems, new applications, such as Internet telephony, audio and video streaming services, video-on-demand, and distributed interactive games, have proliferated. These new applications have diverse quality-of-service (QoS) requirements that are significantly different from traditional best-effort service. For example, high-quality video applications, such as remote medical diagnosis and video-on-demand applications, demand reliable and timely delivery of high bandwidth data, and require QoS guarantees from the network.

On the other hand, a majority of multimedia applications including Internet telephony, video conferencing, and web TV, do not need in-order, reliable delivery of packets, and can tolerate a small fraction of packets that are either lost or highly delayed, while still

maintaining reasonably good quality. These applications employ end-to-end control that adapts to the changing dynamics of the network and can often deliver the acceptable quality to users.

The rapid growth of the Internet and the proliferation of its new applications pose a serious challenge in network performance management and monitoring. The sheer volume of network traffic imposes a burden on network administrators, and demands a visual interface for easy grasp of the current status of the network. The ever-expanding topology renders debugging even a single misbehaving router a very complex task. As more end users are likely to be affected by changes on high-bandwidth backbones, it is all the more important to to monitor network dynamics, and to detect any anomalies or breakdowns inside the network as soon as they appear.

The Simple Network Management Protocol (SNMP) was developed to provide a coherent framework in network management of the Internet. Using SNMP, network management stations send a request to end hosts and routers for variables defined by Management Information Base (MIB), and gather information about interfaces, routing tables, and protocol states from the replies. HP's OpenView, Desktalk Systems' TRENDsnmp+, and Castle Rock Computing's SNMPc are all SNMP-based network monitoring tools.

There are other tools that are available to network administrators; these monitors include OC3MON and OC12MON developed by MCI. They collect traffic data from high-volume trunks of OC3 and OC12 speed, (which correspond to 155.52Mb/s and 622Mb/s, respectively), and provide flow-based analysis of the collected data. The detailed flow-based analysis allows us to identify the type of traffic and the source and destination of the traffic.

Cisco's Internetworking Operating System (IOS) has a similar passive monitoring feature called NetFlow. NetFlow allows the router to capture a number of traffic statistics, including a packet's source and destination IP addresses and port numbers, and protocol

type. The collected data is then exported to another platform where the data can be used in network analysis, planning, management, accounting, billing, and data mining.

Unlike network administrators and managers, most end users and their applications do not have access to the inside of the network. The current Internet has no mechanism for providing feedback on network congestion to the end-systems at the IP layer, and neither does IPv6. For applications and their end hosts, end-to-end measurements may be the only way of measuring network performance, especially when there is no provision inside the network to give information about the current status of the network to users without access to the routers. [11].

There are two modes of network performance measurement: active and passive. Active measurement tools generate packets and inject them to the network for the purpose of performance evaluation. All of the tools listed above are passive, since they do not generate traffic, but merely monitor the traffic on high-capacity trunks and routers.

Users at the edges of the network can learn about the conditions inside the network by actively generating packets at one end and observing the outcome at the other end of the network. Most end-to-end measurement-based tools fall into this category of active measurement.

Performance metrics that are of interest to end users are throughput, latencies, and packet loss rates. Consider a user accessing a web page of an Internet Service Provider (ISP). The user would want the web page to appear on one's own web browser as soon as possible. The user's foremost concern would be the latency, but, if we look closely, we see that packet losses would incur retransmission of packets, and thus increase the overall latency of the web page delivery. For a bulk-data transfer, the overall throughput which itself will be influenced by packet loss would be the most important consideration.

Packet delay and packet loss are two fundamental end-to-end network characteristics that represent quality of service. The magnitude and variation of delay affect the interactivity of applications; the loss rate and the distribution of losses affect the loss recovery

mechanism, and long-term throughput of applications. Measuring end-to-end delay and loss is the first step towards understanding the underlying delay and loss processes.

The magnitude of delay represents the time that a packet spends traversing the network from a sender to a receiver. For delay measurements, a sender adds timestamps to packets for a receiver to gather delay information [53]. If clocks at the sender and receiver are not synchronized, the delay calculated from the sender and receiver timestamps is not accurate. Since the clocks at both end-systems are involved in measuring delay, the synchronization of the two clocks becomes an issue in the accuracy of delay measurement. The Network Time Protocol (NTP) [35] is widely used in the Internet for clock synchronization. It provides an accuracy of the order of milliseconds under reasonable circumstances over time scales of hours to days. It reduces systematic errors in delay measurement, but does not eliminate them [47]. To obtain an accurate measurement of one-way delay, errors and uncertainties related to clocks need to be accounted for. We address the problems that arise in measuring delay later in this dissertation.

Packet loss is an indicator of network congestion. When a node inside the network receives more packets than it can serve (or buffer), it drops packets according to its queueing policy. An end host learns of the congestion when it detects a packet loss. From the end host's point of view, knowledge of the underlying loss processes, its correlation to delay, and the locations of lossy bottlenecks can help in devising control strategies to cope with, and adapt to, the changing conditions inside the network. It is in this respect we focus our work in this dissertation.

The correlation of delay and loss can be exploited by applications to predict future delay or loss, and incorporated into their control mechanisms. The knowledge of network dynamics inside the network can be used by a diverse set of applications: reliable multicast applications that need to place an active server close to a sender, but beyond a bottleneck point; network management tools that monitor and detect abnormal behaviors; multicast applications that join and leave multicast groups based on the performance of

the groups, and so on. Recent developments in MLE-based (Maximum Likelihood Estimator) inference techniques [5, 6] have shown that one can attain unbiased and asymptotically converging estimates of link losses within the network based only on measurements made at the edge of the network. Such inference techniques, once implemented, will allow end hosts to obtain timely and accurate knowledge of network-internal characteristics, and to enhance their control mechanisms accordingly.

In this dissertation, we address issues the concerning measurement and analysis of end-to-end performance, more specifically, delay and loss, using active measurement.

1.2 Problem Statement

In this dissertation we raise the following specific questions concerning end-to-end network performance.

- What are the clock synchronization problems that affect end-to-end delay measurements, and how can we remove them from measurements?
- Given end-to-end Internet delay measurements, what control mechanism can a continuous-media audio application employ to adapt well to the changing network conditions and to minimize the average playout delay?
- What is the correlation between packet delay and loss observed at end hosts? What is its implication to applications?
- How accurate is the inferred network-internal performance compared to actual measurements?
- What are the implementation issues and trade-offs one should consider when deploying the inference techniques?

We have outlined above the problems we address in the rest of the dissertation. In the next section, we briefly note the contributions of the dissertation. We and investigate the problems in depth in Chapters 2 through 5.

1.3 Contributions of the Dissertation

The contributions of this dissertation are the following.

- We present a framework for understanding the systematic errors introduced in one-way delay measurements by unsynchronized clocks, and develop a new linear--programming-based algorithm to estimate the clock skew (the difference in two clocks' frequencies in Internet delay measurements). We show that the new algorithm is robust in the sense that the error margin of the skew estimate is independent of the magnitude of the skew. In actual delay measurements, our algorithm performs accurately in the presence of different levels of network congestion. Furthermore, through simulation we show that the new algorithm is unbiased, and that the sample variance of its skew estimate is better (smaller) than existing algorithms.

- We investigate the problem of adaptive control mechanisms for continuous media applications, based on the use of on-line measurements. More specifically, we focus on the use of adaptive audio playout delay adjustment algorithms at end hosts to improve end-to-end performance. We present efficient algorithms to compute upper and lower bounds on the optimum (minimum) average playout delay for a given number of packet losses. These bounds indicate the performance achievable by any playout delay adjustment algorithm. We also present an algorithm that keeps a window of past history of packet delays to estimate the playout delay, and adapts quickly to sudden changes in delay. We show that this algorithm performs close to optimal for the range of parameters of interest.

- We examine the correlation between packet delay and loss in the Internet. Our goal is to study the extent to which one performance measure can be used to predict the future behavior of the other (e.g., whether observed increasing delay is a good predictor of future loss) so that an adaptive continuous media application might take *anticipatory* action based on observed performance. We present a quantitative study of the correlation between packet delay and loss, and discuss their implications for adaptive continuous media applications.

- We present MBone experiments that validate the effectiveness of an MLE-based inference technique we call MINC (Multicast Inference of Network Characteristics). MINC exploits the performance correlation experienced by multicast receivers to infer the loss rate and other attributes of internal links in a multicast tree. We validate MINC by comparing the inferred loss rate on internal MBone tunnels with the measured loss rate using the `mtrace` tool. The results show that the inferred values match measured values very closely.

- We study the performance of MINC when given a limited amount of information, and present simulation results for its performance. We vary the number of receivers, the range of loss rates, and the tree topology, and analyze the MINC performance under these different configurations. These results are of use in the design of any MINC-based applications.

1.4 Structure of the Dissertation

The rest of this dissertation is organized as follows. In Chapter 2 we look at the impact of clock synchronization on end-to-end delay, and introduce a linear programming based algorithm to estimate the clock skew in network delay measurements. In Chapter 3 we consider the problem of adaptively adjusting the playout delay of an audio application, present efficient algorithms for computing a bound on the achievable performance of any

playout delay adjustment algorithm, and describe a new adaptive delay adjustment algorithm that tracks the network delay of recently received packets and efficiently maintains delay percentile information. In Chapter 4 we quantify the correlation between delay and loss as sample mean delay conditioned on loss and loss conditioned on delay, and analyze it. In Chapter 5 we present an experimental study that validates the MLE-based inference analysis on the MBone, and extend the work to the study of the performance of MLE-based inference analysis given a limited number of probe packets.

Finally, in Chapter 6 we give a summary of the dissertation, and present extensions and ideas for future work.

CHAPTER 2

ESTIMATION AND REMOVAL OF CLOCK SKEW

2.1 Introduction and Motivation

End-to-end delay traces are frequently used in analyzing network performance. The accuracy of such measurements depends on whether the clocks involved in measurement are synchronized. To obtain an accurate measurement of one-way delay, errors and uncertainties related to clocks must be accounted for. When one of the clocks involved in the measurement resets its time, the measured delay using the timestamps from two clocks may be affected, depending on the comparative magnitudes of delay and the time adjustment. Let us first examine a sample of the measured one-way delay between two hosts. This sample illustrated in Figure 2.1 is taken from a trace described in Section 2.6.4 (Trace 2.1 in Table 2.2).

We know that the system where the sender resided adjusted its time every six hours scheduled as a `cron` task, and believe the large jump of 15 seconds in Figure 2.1 is due to such an adjustment.

Depending on the order of time adjustment, it is not always so easy to detect this change. Let us assume a time adjustment of 10ms, and consider a stream of packets from a sender and a receiver that do not take the same path. Due to a routing change in the middle of the stream, the first half of the packets use a faster path than that taken by the second half of packets. The end-to-end delays of the second half of the packets would be consistently larger than those of the first half. We should expect a similar scatter-plot of this case to Figure 2.1, except that the time adjustment would be less apparent. It would be hard to differentiate a true time adjustment from a routing behavior.

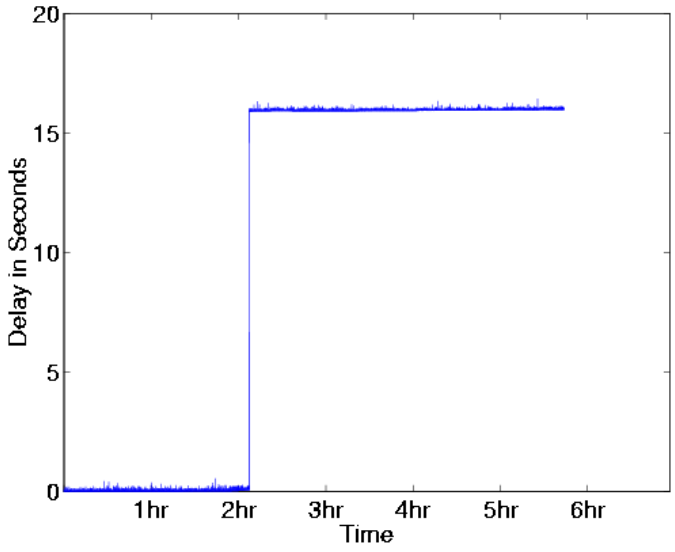


Figure 2.1. Scatter-plot of delay from Trace 2.1 with a time adjustment

When two clocks involved in the measurement run at different frequencies (that is, have a clock skew), inaccuracies are introduced in the measurement.

Figure 2.2 shows part of a trace that plots end-to-end delay as a function of packet send time taken after the time adjustment Trace 2.1 in Table 2.2 in Figure 2.1. The x-axis is the sender timestamp, and the y-axis is the delay calculated by subtracting the sender timestamp from the receiver timestamp. The measured delays lie in the range of 31.15 to 31.5 seconds. The measured delays are not the actual end-to-end delays, but include the clock offset between the two clocks plus the end-to-end delay. Clock offset is the difference in times of two clocks, and skew is the difference in clock speeds. We defer the formal definitions of offset and skew to Section 2.2.

The end-to-end delay consists of transmission and propagation delays plus variable queueing delay. When all of the packets go over the same route to the receiver, they incur the same propagation delay, and, if they are of the same size, the transmission delay also is the same. Even if the packets go over the same route, and are of the same size, the

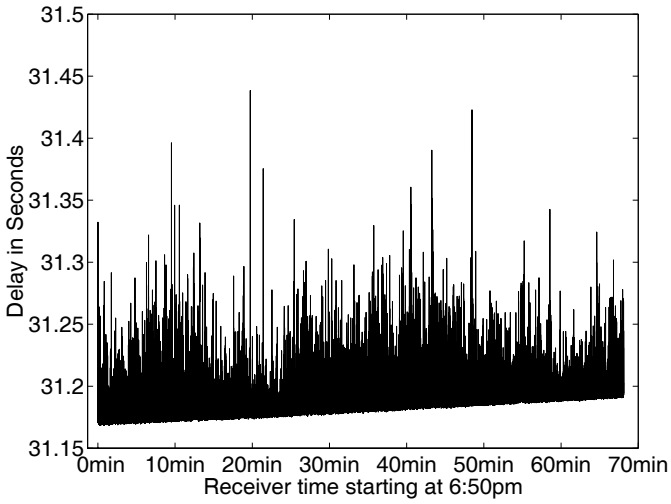


Figure 2.2. Scatter-plot of delay from Trace 2.1

packets experience different levels of congestion inside the network. This is what causes the variability in the end-to-end delay.

In Figure 2.2 the delay shows an increasing trend of approximately 100 milliseconds over the duration of 70 minutes at the receiver. This offset is significant enough to distort performance metrics such as the average and autocorrelation of end-to-end delay. While one might imagine that the ever increasing minimum delay is due to increasing congestion and queueing delay, this is unlikely as the minimum observed delay increases over time. Instead, the linear increase in delay attests to a constant speed difference between two clocks.

Previous work by Paxson [46, 47] addresses problems in delay measurements due to clock adjustments and rate mismatches. It uses forward and reverse path measurements of delay between a pair of hosts to deal with clock synchronization problems, such as relative offset and skew. Many applications, however, see only one-way delay (e.g., Internet telephony, video-on-demand applications, RealPlayer, web TV), and still have to deal with the clock synchronization problems in packet delay. One-way measurements alone

are not enough to infer the clock offset, and we cannot distinguish the clock offset from the fixed portion of end-to-end delay. For example, in Figure 2.2, it is difficult to tell how much of the 31.15 seconds is due to the time difference between clocks and the fixed transmission and propagation delay, without the availability of more information. Due to this lack of information in one-way delay measurements, we focus on the variable portion of one-way delay measurements.

The variable queueing delay serves a very important role in network and application design. Continuous-media applications such as audio and video need to absorb the delay jitter perceived at the receiver for smooth playout of the original stream [48, 39, 12]. Determining the correct amount of buffering, and reconstructing the original timing is crucial to the performance of continuous-media applications. The variable queueing delay is also useful in monitoring network performance at the edges of the network; the transmission and propagation delays are fixed per route, and do not convey any information about the dynamic changes inside the network when packets follow a fixed route.

In this chapter we focus on filtering out the effects of clock skew specifically in one-way delay measurements.

The rest of the chapter is organized as follows. In Section 2.2 we define the terms needed to describe clock behavior, and introduce the notation to be used in the remainder of the chapter. In Section 2.3 we formalize the clock synchronization problem between two hosts, and show how skew and offset affect the delay measurements. Then we list several desirable properties expected of a skew estimation algorithm. We introduce our skew estimation algorithm that is based on a linear programming technique in Section 2.4, and we survey three existing algorithms in Section 2.5. In Section 2.6 we compare the four algorithms presented in this chapter, and in Section 2.7 discuss how this work can be extended to the cases where the skew is not constant. Section 2.8 concludes this chapter.

2.2 Background

2.2.1 Clock terminology

In this section we introduce the terminology we use to describe clock behavior. A *clock* is a piecewise continuous function that is twice differentiable except on a finite set of points:

$$C : \mathcal{R} \longrightarrow \mathcal{R}$$

where $C'(t) \equiv dC(t)/dt$ and $C''(t) \equiv d^2C(t)/dt^2$ exist everywhere except for $t \in P \subset \mathcal{R}$ where $|P|$ is finite.

A "true" clock reports "true" time at any moment, and runs at a constant rate of one. Let C_T denote the "true" clock; it is the identity function given below,

$$C_T(t) = t \text{ and } P_T = \emptyset$$

In this chapter, we use the following nomenclature from [34, 35] to describe clock characteristics. Let C_a and C_b be two clocks:

- **offset**: the difference between the time reported by a clock and the "true" time; the offset of C_a is $(C_a(t) - t)$. The offset of the clock C_a relative to C_b at time $t \geq 0$ is $C_a(t) - C_b(t)$.
- **frequency**: the rate at which the clock progresses. The frequency of C_a at time t is $C'_a(t)$.
- **skew**: the difference in the frequencies of a clock and the "true" clock. The skew of C_a relative to C_b at time t is $(C'_a(t) - C'_b(t))$.
- **drift**: The drift of clock C_a is $C''_a(t)$. The drift of C_a relative to C_b at time $t \geq 0$ is $(C''_a(t) - C''_b(t))$.

Two clocks are said to be *synchronized* at a particular moment in time if both the relative offset and skew are zero. When it is clear that we refer to clocks of a sender and a receiver that are not the "true" clock in our discussion, we simply refer to relative offset and relative skew as offset and skew, respectively.

It is sometimes convenient to compare the ratio of frequencies between two clocks instead of the skew. This is captured by the following definition.

- **clock ratio**: the ratio of frequencies between a clock and the "true" clock; the ratio of C_a is $C'(a)$. The ratio of C_a relative to C_b at time t is $C'_a(t)/C'_b(t)$.

Let C_a and C_b have constant frequencies, and α and δ be the clock ratio and skew of C_b relative to C_a, respectively, i.e., $\alpha = C'_b/C'_a$ and $\delta = C'_b - C'_a$. The relation between the clock ratio and the skew is:

$$\delta = C'_b - C'_a = \alpha C'_a - C'_a = (\alpha - 1)C'_a \tag{2.1}$$

From now on, we assume that the sender and receiver clocks have constant frequencies, so that their skew and clock ratio are constant over time; we use the skew and clock ratio interchangeably, and use (2.1) whenever necessary to convert one from the other.

2.2.2 Time duration consistent with a clock

In the previous section, we defined a *clock* and introduced terminology relevant to its behavior. In this section we look at how a *time duration* is measured according to a clock. Let $\Delta(t_1, t_2, C_a)$ denote the time that has passed according to C_a between t_1 and t_2 of the "true" clock. Since a clock is a piecewise continuous function, we define the time duration as:

$$\begin{aligned}\Delta(t_1, t_2, C_a) &\equiv \int_{t_1}^{t_2} C'_a dt \\ &= \int_{t_1}^{p_1} C'_a dt + \int_{p_1}^{p_2} C'_a dt + \cdots + \int_{p_{n-1}}^{p_n} C'_a dt + \int_{p_n}^{t_2} C'_a dt\end{aligned}$$

$$\text{where } P_a \bigcap (t_1, t_2) = \{p_1, p_2, \ldots, p_n\}$$
$$\text{and } t_1 < p_1 < p_2 < \cdots < p_n < t_2.$$

If $P_a \bigcap (t_1, t_2) = \emptyset$, then

$$\Delta(t_1, t_2, C_a) = \int_{t_1}^{t_2} C_a' dt = C_a(t_2) - C_a(t_1) \tag{2.2}$$

When two clocks are not synchronized and, more specifically, have different frequencies, time duration measured with one clock will be different from the other. We say that a time duration measured with a clock is **consistent** with that by any other clock of the same frequency and any offset. If two clocks have a non-zero skew, time measured on one clock will not be consistent with that measured by any other clock.

We have modeled a clock as a piecewise continuous function in order to take into account the restrictions of real clocks. The resolution of a clock on a computer system is the smallest unit by which the clock's time is updated, and is greater than zero. At best, a clock in a computer is a step function with increments at every unit of its time resolution. We consider the time reports by a real clock with a fixed minimum resolution as samples of a continuous function at specific moments, and thus circumvent the discretization effect of the real clock. Another problem a real clock poses is the abrupt time adjustment possible through a time resetting system call. Some systems that do not run NTP [35] have a very coarse-grain (in the order of hours) synchronization mechanism in the `cron` table. The time adjustment in such a case can be several orders of magnitude larger than the usual increment of the clock resolution, and the time can even be set backward. The piecewise nature of a clock function accommodates the abrupt time adjustment, and computes the duration of elapsed time on a clock.

When a delay measurement involves more than one clock, the synchronization between those clocks has a tremendous impact on the accuracy of the measurement. Let

us consider the case of measuring a packet delay between two hosts. The sender adds a timestamp to a packet when it leaves the sender, and the receiver records the time the packet arrives at the receiver. When the two host clocks are perfectly synchronized, the difference between the two timestamps is the end-to-end network delay experienced by that packet. If the clocks on the two hosts have a non-zero offset, but no skew, the difference between two timestamps includes not only the end-to-end delay, but also the offset. Given only a one-way measurement, we cannot distinguish the offset from the measurement, unless we are given the network delay, which is what we intended to measure in the first place. If the clocks have a non-zero skew, not only is the end-to-end delay measurement off by an amount equal to the offset, but it also gradually increases or decreases over time depending on whether the sender clock runs slower or faster than the receiver clock.

In the following sections we formalize the clock synchronization problem outlined above, and show how to remove the clock skew in measurements.

2.3 Basics of a Skew Estimation Algorithm

In the previous section we defined a clock, and what is meant for a time duration to be consistent with a clock. In this section we discuss the problem of estimating and removing the effects of clock skew from delay measurements. We first derive how much the clock skew contributes to the measured end-to-end delay if the skew is non-zero and constant. This derivation provides a basis for the discussion of several desirable properties for skew estimation algorithms.

2.3.1 Delay measured between two clocks

From Section 2.3.1, if the clock ratio between the sender and receiver clocks is greater than or less than one, network delays will appear to become longer or shorter over the course of a measurement period. The purpose of removing the effect of skew on the

delay measurements is to transform the delay measurements so that they are consistent either with the sender or receiver clock. In our work, we have chosen to make the delay measurements consistent with the receiver clock. When the receiver has no means to access the "true" clock, the only clock the receiver has access to is its own clock. It is, thus, natural to measure one-way delays according to the receiver clock. For the simplicity of derivation, we assume the receiver clock to be the true clock, i.e. $C_r'(t) = 1$ and $\alpha = C_s'(t)$. However, this assumption does not lead to loss of generality; the same derivation leads to the delay consistent with the receiver clock whether it is the "true" clock or not.

For different size packets, the clock skew may not be distinguishable from the delay trend, if any. For example, if the packet size grows over time and the route from the sender to the receiver is fixed, then the transmission delay gradually increases, and it is hard to distinguish a skew from this delay trend. Thus we assume all the packets have the same size.

Let us now introduce the terminology for clocks, timestamps, and delays used in measurements.

- C_s: sender clock.
- C_r: receiver clock.
- N: number of packets that arrive at the receiver.
- s_i: timestamp of the i-th packet leaving the sender according to C_r, $i = 1, 2, \ldots, N$.
- t_i^s: timestamp of the i-th packet leaving the sender according to C_s, $i = 1, 2, \ldots, N$; $t_i^s = C_s(s_i)$.
- t_i^r: timestamp of the i-th packet arriving at the receiver according to C_r, $i = 1, 2, \ldots, N$.

- d_i: end-to-end delay measurement of the i-th packet, $i = 1, 2, \ldots, N$;

$$d_i = t_i^r - t_i^s \tag{2.3}$$

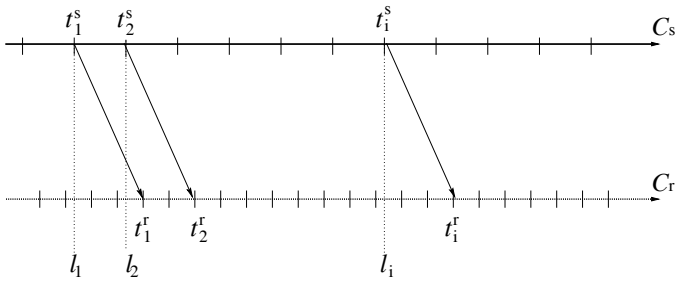


Figure 2.3. Timing chart showing constant delay

Figure 2.3 shows the timing between C_s and C_r when C_s runs at half the speed of C_r and all of the packets experience the same network delay. The end-to-end delay of the i-th packet consistent with C_r is $t_i^r - s_i$. s_i, however, is not known at the receiver, and we calculate d_i using t_i^s and t_i^r. Consequently, in this case, the end-to-end delay is consistent with neither C_s nor C_r. To make it consistent with C_r, we need to determine the skew of C_r relative to C_s, and remove it from the measurement d_i.

2.3.2 Clock Skew in Delay Measurements

When there is a constant clock skew between two clocks, the clock offset between them gradually increases or decreases over time, depending on the sign of the skew. The amount of increase or decrease in the clock offset is proportional to the time duration of observation. We use the change in offset to estimate the clock skew. Thus it is more convenient to use timestamps relative to a specific point in time, such as the departure or arrival time of the first packet, than to use absolute timestamps. Below we introduce relative timestamps at the sender and the receiver.

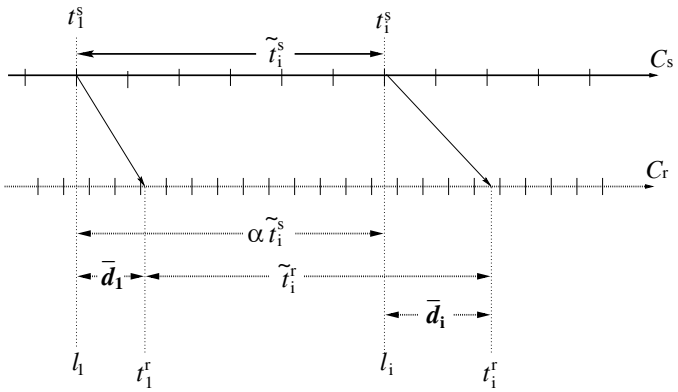


Figure 2.4. Timing chart showing variable delay

- $\tilde{t}_i^s$: time duration between the first and i-th packets' departure times from the sender consistent with C_s.

$$\begin{aligned} \tilde{t}_1^s &= 0 \\ \tilde{t}_i^s &= \Delta(s_1, s_i, C_s) = t_i^s - t_1^s \end{aligned}$$

- $\tilde{t}_i^r$: time duration between the first and i-th packets' arrival times at the receiver consistent with C_r.

$$\begin{aligned} \tilde{t}_1^r &= 0 \\ \tilde{t}_i^r &= t_i^r - t_1^r \end{aligned}$$

By (2.1) and (2.2),

$$\begin{aligned} \Delta(s_1, s_i, C_r) &= s_i - s_1 \\ &= \alpha\Delta(s_1, s_i, C_s) = \alpha\tilde{t}_i^s \end{aligned} \tag{2.4}$$

Figure 2.4 illustrates the relationship between $\Delta(s_1, s_i, C_r)$ and $\tilde{t}_i^s$ on a timing chart. The quantities $\bar{d}_1$ and $\bar{d}_i$ shown in the figure are defined below.

- $\bar{d}_i$: end-to-end delay consistent with C_r.

$$\begin{aligned}
\bar{d}_1 &= \Delta(s_1, t_1^r, C_r) = t_1^r - s_1 \\
\bar{d}_i &= \Delta(s_i, t_i^r, C_r) = t_i^r - s_i \\
&= t_i^r - s_1 - \alpha\tilde{t}_i^s = (t_i^r - t_1^r) + (t_1^r - s_1) - \alpha\tilde{t}_i^s \\
&= \tilde{t}_i^r + \bar{d}_1 - \alpha\tilde{t}_i^s
\end{aligned} \tag{2.5}$$

The quantity $\bar{d}_i$ is not obtainable directly from measured timestamps, due to the skew between the sender and receiver clocks. Instead, following is the quantity obtainable from actual timestamps:

- $\tilde{d}_i$: delay calculated from $\tilde{t}_i^s$ and $\tilde{t}_i^r$.

$$\begin{aligned}
\tilde{d}_i &= \tilde{t}_i^r - \tilde{t}_i^s = \tilde{t}_i^r + \bar{d}_1 - \alpha\tilde{t}_i^s + (\alpha - 1)\tilde{t}_i^s - \bar{d}_1 \\
&= \bar{d}_i + (\alpha - 1)\tilde{t}_i^s - \bar{d}_1
\end{aligned} \tag{2.6}$$

The goal of estimating and removing the clock skew is to obtain $\bar{d}_i$ from the actual delay measurement $\tilde{d}_i$. From (2.3) and (2.6), we note that the difference between d_i and $\tilde{d}_i$ is:

$$d_i - \tilde{d}_i = t_1^r - t_1^s.$$

Also note in (2.6) that $\tilde{d}_i$ differs from $\bar{d}_i$ by $(\alpha - 1)\tilde{t}_i^s$ minus a constant $\bar{d}_1$. If $\alpha > 1$, $(\alpha - 1)\tilde{t}_i^s$ grows linearly with $\tilde{t}_i^s$, and, thus, $\tilde{d}_i$ gets larger. This explains the increasing trend observed in Figure 2.2. Finally, from (2.6) it is obvious that the measured network delays can be made consistent with C_r given α and $\bar{d}_i$ according to:

$$\bar{d}_i = \tilde{d}_i - (\alpha - 1)\tilde{t}_i^s + \bar{d}_1. \tag{2.7}$$

Let $\hat{\alpha}$ and $\hat{\beta}$ be the estimates of α and $\bar{d}_1$. Then the delay after the skew removal, $\hat{d}_i$, is:

$$\hat{d}_i = \tilde{d}_i - (\hat{\alpha} - 1)\tilde{t}_i^s + \hat{\beta} \tag{2.8}$$

2.3.3 Desirable Properties for Skew Estimation Algorithms

Before we delve into the details of the skew estimation algorithm, we first state some desirable properties that any such algorithm should exhibit. These properties are used later as a basis for comparing different estimation algorithms.

We begin by introducing the notations for an estimation algorithm and estimates parametrized by the estimation algorithm. Let $\mathcal{A}$ be a skew estimation algorithm. We make the same assumption as in Section 2.3.1 that the skew between the sender and the receiver clocks is constant, and the receiver clock is the "true" clock. Given a set of end-to-end delays, $\mathcal{D} = \{\bar{d}_i\}_{i=1}^N$, which are predetermined and fixed, and also consistent with the "true" clock, we know that delay measurements, $\tilde{d}_i = \bar{d}_i, i = 1, \ldots, N$, if there is no clock skew between the two hosts ($\alpha = 0$); $\tilde{d}_i$ differ from $\bar{d}_i$ if the clock skew is not zero ($\alpha \geq 0$). In that sense $\tilde{d}_i$ depends on the clock skew, α, and $\bar{d}_i$, and is noted $\tilde{d}_i(\alpha, \mathcal{D}), i = 1, \ldots, N$.

We define $\hat{\alpha}_{\mathcal{A}}(\alpha, \mathcal{D})$ and $\hat{\beta}_{\mathcal{A}}(\alpha, \mathcal{D})$ to be the estimates of α and $\bar{d}_1$, respectively, delivered by algorithm $\mathcal{A}$, when given $\bar{d}_i, 1 \leq i \leq N$ and α. Below is a list of desirable properties that should be exhibited by algorithm $\mathcal{A}$.

- **Property 1**: The time and space complexity of algorithm $\mathcal{A}$ should be linear in N. The computational complexity of an algorithm in terms of time and space is an important metric in assessing the performance and applicability of the algorithm.

- **Property 2**: Since the purpose of skew estimation is to remove the skew from delay measurements, it is desirable that *the delays be non-negative after the skew is removed*.

$$\hat{d}_i = \tilde{d}_i(\alpha, \mathcal{D}) - (\hat{\alpha}_{\mathcal{A}}(\alpha, \mathcal{D}) - 1)\tilde{t}_i^s + \hat{\beta}_{\mathcal{A}}(\alpha, \mathcal{D}) \geq 0$$

- **Property 3**: The skew estimation algorithm should be robust in the sense that it is not affected by the magnitude of the actual skew. That is, the difference between the estimate and the actual skew should be independent of the actual skew. Under the same network condition, the skew estimate for different skews should exhibit the same margin of error from the actual skew, no matter how small or large the skew is. We state this property as follows:

$$\hat{\alpha}_{\mathcal{A}}(\alpha, \mathcal{D}) - \alpha = \hat{\alpha}_{\mathcal{A}}(1, \mathcal{D}) - 1, \ \ \forall \alpha > 0. \tag{2.9}$$

In the following section, we introduce a new algorithm based on linear programming to estimate α in delay measurements, and use the skew estimate of the algorithm to remove the skew from one-way delay measurements to make them consistent with the receiver clock. In the next two sections, we focus on a simple case where the clock skew is constant, and defer the discussion of a time-varying skew case to Section 2.7.

2.4 Linear Programming Algorithm

Figure 2.2 illustrates a trace where the skew between two clocks was nearly constant over the measurement duration. Looking at the figure, one is tempted to pick up a ruler, draw a line that skims through the bottom of the mass of the scatter-plot, measure the angle between the line and the x-axis, and calculate the skew using simple trigonometry. This approach is difficult to automate, and invites human errors that are untraceable. A second thought would be to pick the first and last data points, and draw a line between them. The accuracy of this approach, however, can be easily thrown off, since end-to-end delays exhibit high variability on the order of magnitude larger than the skew all through the measurement duration. Our approach is to fit a line that lies under all the data points, but as closely to them as possible.

We have formulated the above idea as a linear programming problem. The condition that the line should lie under all the data points forms the first part of our linear programming problem, and defines the feasible region for a solution; the objective function of the linear programming problem is to minimize the sum of the distances between the line and all the data points on the y-axis.

2.4.1 Algorithm

Having presented the intuition behind the algorithm, we now formally introduce the algorithm. The goal is to estimate the clock ratio α given $\tilde{t}_i^s$ and $\tilde{d}_i$. The output of the skew estimation algorithm is: $\hat{\alpha}$ and $\hat{\beta}$, where $\hat{\alpha}$ is the estimate of α, and $\hat{\beta}$ is the estimate of $\bar{d}_1$. We return to the interpretation of $\hat{\beta}$ at the end of this section. If we estimate both α and $\bar{d}_1$ correctly, then we can subtract $(\alpha - 1)\tilde{t}_i^s - \bar{d}_1$ from $\tilde{d}_i$, and obtain $\bar{d}_i$, which is the end-to-end delay consistent with C_r and free of clock skew. Even when the estimates $\hat{\alpha}$ and $\hat{\beta}$ are not exactly the same as α and $\bar{d}_1$, we still want the resulting end-to-end delay to be non-negative, after the skew is removed. When we formulate our skew estimation as a linear programming problem, this condition defines the feasible region within which a solution should lie,

$$\tilde{d}_i - (\alpha - 1)\tilde{t}_i^s + \beta \geq 0, \quad 1 \leq i \leq N \tag{2.10}$$

There are infinitely many pairs of α and β that satisfy the condition above, if the feasible region from (2.10) is not trivial. Our objective function to minimize the distance between the line and all the delay measurements is stated as:

$$(\hat{\alpha}, \hat{\beta}) = \arg_{\alpha,\beta} \min\{\sum_{i=1}^{N} \Big(\tilde{d}_i - (\alpha - 1)\tilde{t}_i^s + \beta\Big)\} \tag{2.11}$$

subject to (2.10).

One important point to note in (2.10) is that the estimated end-to-end delay of $\bar{d}_i$, calculated as $(\tilde{d}_i - (\hat{\alpha} - 1)\tilde{t}_i^s + \hat{\beta})$ once $\hat{\alpha}$ and $\hat{\beta}$ are obtained, will be greater than zero, instead of being greater than $\min_i \bar{d}_i$. Thus $\hat{\beta}$ is actually an estimate of ($\bar{d}_1$ - $\min_i \bar{d}_i$). The resulting delay of $\tilde{d}_i - (\hat{\alpha} - 1)\tilde{t}_i^s + \hat{\beta}$ is not the end-to-end delay, but rather the variable portion of the end-to-end delay.

In the following sections, we look into other algorithms that can be used in skew estimation, and compare them with our linear programming algorithm in terms of the properties listed in Section 2.3, and their performance in actual and synthetic measurements.

2.5 Other Algorithms

2.5.1 Paxson's algorithm

In [46, 47], Paxson introduced an algorithm for removing a clock skew from a set of forward and reverse path delay measurements. In this section we briefly describe how his algorithm can be used when given delays in only one direction. Assume that the input to the algorithm is the same as in the previous LP algorithm; $\tilde{d}_i$ and $\tilde{t}_i^s$, for $1 \leq i \leq N$; Readers are referred to [46, 47] for more detail.

Paxson's algorithm is as follows:

- **Step 1.** Partition $\tilde{d}_i, i = 1, \ldots, N$ into $\sqrt{N}$ segments, and pick the minimum delay measurement from each segment. The selected measurements are called the "de-noised" one-way transit times (OTTs).

- **Step 2.** Pick the median of the slopes of all possible pairs of the "de-noised" OTTs. If the median slope is negative, assume that the OTTs have a decreasing trend (here we assume a decreasing trend is detected).

- **Step 3.** Select the cumulative minima test from the "de-noised" OTTs (see [46]), and test if the number of cumulative minima is sufficiently large to show that the decreasing trend found in Step 2 is probabilistically not likely, if there is no trend.

- **Step 4.** If the cumulative minima test declares a trend in Step 3, pick the median from the slopes of all possible pairs of the cumulative minima: output it as the estimate of $\alpha - 1$. Otherwise, the algorithm concludes that there is no skew and outputs zero.

The core of Paxson's algorithm is a robust line fitting technique based on robust statistics [18]. It uses the median as a robust estimate for the slope. As mentioned in [46, 47], robust line fitting alone fails in estimating the slope of the trend due to the high variability in OTTs, and that is why the "de-noised" OTTs and cumulative minima are used in his algorithm.

2.5.2 Linear regression algorithm

Linear regression is a standard technique for fitting a line to a set of data points. It is optimal in the mean square sense if the network delays are normally distributed, but is not robust in the presence of outliers. As pointed out in [46, 47], it is not a good choice for a skew estimation, even when applied to the "de-noised" OTTs above. Here we use it only as a reference algorithm that has no knowledge of the underlying behavior of delay measurements.

The linear regression algorithm in a skew estimation provides estimates of α and β that minimize the mean square error e in:

$$e = \sum_{i=1}^{N} \{\tilde{d}_i - (\hat{\alpha} - 1)\tilde{t}_i^s + \hat{\beta}\}^2. \tag{2.12}$$

2.5.3 Piecewise minimum algorithm

In this section we introduce another simple algorithm called piecewise minimum algorithm. It does not output one single estimate for the skew, but a set of estimates. The algorithm is as follows. It partitions the delay measurements into segments of the same length, picks a minimum from each segment, and connects them to obtain a concatenation of line segments. The minima are the same as the "de-noised" OTTs in Section 2.5.1. The resulting concatenation of line segments are the estimates of the skew, and is very unlikely to be a straight line.

When the skew is as obvious as in Figure 2.2, the resulting concatenation of line segments is close to a straight line. It, however, does not perform as well in other situations. We look at such cases later in Section 2.6.4

2.6 Comparison of the Four Algorithms

In this section we compare the four algorithms described in Sections 2.4 and 2.5. In particular we assess how well they perform based on the desirable properties from Section 2.3. We also apply the algorithms to actual delay measurements, and compare their performance by looking at the adjusted delays. Lastly, we use delay measurements from simulation to compare our approach to Paxson's algorithm.

2.6.1 Computational complexity

The time complexity of a two-variable linear programming problem is proven to be $O(N)$ [13, 33]. We have implemented a simple and efficient $O(N)$ algorithm that exploits the fact that $\tilde{t}_i^s$'s are sorted in our specific problem [40]. The other three algorithms also have complexity of $O(N)$.

2.6.2 Non-negative delay after the skew removal

In order to guarantee that the delay remains positive after the skew is removed, a skew estimation algorithm must estimate $\bar{d}_1$ correctly. The LP algorithm, however, is the only one that estimates $\bar{d}_1$ (or $\bar{d}_1 + \min_i \bar{d}_i$), as explained in Section 2.4. Paxson's original algorithm for skew estimation is for two-way measurements *after* the clock offset has been removed. The linear regression algorithm provides an estimate of $\hat{\beta}$. However this is just a y-intercept of the regression line which bears no relevance to the correct estimation of $\bar{d}_1$. The piecewise minimum algorithm outputs a concatenation of line segments, and the slopes of those line segments are skew estimates. The algorithm does not have any provision to guarantee that all the data points lie above the concatenation of line segments.

For the three algorithms that do not provide an estimate for $\bar{d}_1$ that ensures that delays are non-negative after the skew removal, we choose $\hat{\beta}$ that satisfies the following condition for all $\hat{\alpha}$'s in each algorithm:

$$\hat{\beta} = \max_{1 \le i \le N} \{\hat{\beta}_i : \tilde{d}_i - (\hat{\alpha}_i - 1)\tilde{t}_i^s + \hat{\beta}_i > 0\} \tag{2.13}$$

where $\hat{\alpha}_i = \hat{\alpha}$ and $\hat{\beta}_i = \hat{\beta}$ for $1 \le i \le N$ in Paxson's and linear regression algorithms; in the piecewise minimum algorithm $\hat{\alpha}_i$ and $\hat{\beta}_i$ are determined by the line segment to which $\tilde{d}_i$ and $\tilde{t}_i^s$ belong to.

2.6.3 Robustness

We focus on the performance of an algorithm, as measured by the difference between the estimate and the actual skew, and whether the difference depends on the variability of the network delays alone, and not on the magnitude of the clock skew. This property guarantees that the estimation algorithm performs in a robust manner, in the sense that the margin of error remains the same, no matter how large the skew is.

We first show that the LP algorithm satisfies this property, and then discuss how well the other algorithms satisfy the property.

2.6.3.1 Linear Programming Algorithm

We use the same assumptions and notations for the skew estimation algorithm and estimates as in Section 2.3.3 when considering two different clock skews: a set of delays, $\mathcal{D} = \{\bar{d}_i\}_{i=1}^{N}$, where $\bar{d}_i$'s are fixed, and the clock ratio varies from one to some constant. It can be restated as follows. From one set of measurements to the other, nothing changes except for the frequency of the sender clock relative to the receiver clock. The receiver observes that the delay measurements, $\tilde{d}_i$'s, are different between two sets, but the end-to-end delays consistent with the receiver clock remain the same in both sets. We also note that $\tilde{t}_i^s$'s remain the same in both sets.

Let us consider a case where the sender and receiver clocks are "true" clocks, and a set of packet delays, $\mathcal{D} = \{\bar{d}_i\}_{i=1}^{N}$, is consistent with the "true clock." Suppose that we measure those delays when the frequency of the sender clock changes so that the skew is $\alpha \neq 1$. We have

$$\tilde{d}_i(1, \mathcal{D}) = \bar{d}_i - \bar{d}_1 \tag{2.14}$$

$$\tilde{d}_i(\alpha, \mathcal{D}) = \bar{d}_i + (\alpha - 1)\tilde{t}_i^s - \bar{d}_1 \tag{2.15}$$

from (2.6).

Let $\mathcal{A}$ be the LP algorithm, and consider the problem of determining $\hat{\alpha}$ and $\hat{\beta}$ when both clocks are "true" clocks. By (2.10), (2.11), and (2.14), the problem becomes minimizing

$$\sum_{i=1}^{N} \{\bar{d}_i - \bar{d}_1 - (\hat{\alpha} - 1)\tilde{t}_i^s + \hat{\beta}\}$$

such that

$$\hat{\beta} \geq (\hat{\alpha} - 1)\tilde{t}_i^s - \bar{d}_i + \bar{d}_1, \quad \text{for} \quad 1 \leq i \leq N.$$

Let $\hat{\alpha}_{\mathcal{A}}(1, \mathcal{D})$ and $\hat{\beta}_{\mathcal{A}}(1, \mathcal{D})$ be the values that solve this problem.

Now define $\alpha^* = (\alpha + \hat{\alpha}_{\mathcal{A}}(1, \mathcal{D}) - 1)$, and substitute $\alpha^* - \alpha$ with $\hat{\alpha}_{\mathcal{A}}(1, \mathcal{D}) - 1$ above; the above problem is now equivalent to choosing α^* and $\hat{\beta}$ that minimize

$$\sum_{i=1}^{N} \{\bar{d}_i - \bar{d}_1 - (\alpha^* - \alpha)\tilde{t}_i^s - \hat{\beta}\}$$

such that

$$\hat{\beta} \geq (\alpha^* - \alpha)\tilde{t}_i^s - \bar{d}_i + \bar{d}_1, \quad \text{for} \quad 1 \leq i \leq N.$$

By (2.15), it is equivalent to choosing α^* and $\hat{\beta}$ to minimize

$$\sum_{i=1}^{N} \{\tilde{d}_i(\alpha, \mathcal{D}) - (\alpha^* - 1)\tilde{t}_i^s + \hat{\beta}\}$$

such that

$$\hat{\beta} \geq (\alpha^* - 1)\tilde{t}_i^s - \tilde{d}_i(\alpha, \mathcal{D}), \quad \text{for} \quad 1 \leq i \leq N,$$

which solves the case when the skew is $\alpha \neq 1$. Let $\hat{\alpha}_{\mathcal{A}}(\alpha, \mathcal{D})$ and $\hat{\beta}_{\mathcal{A}}(\alpha, \mathcal{D})$ be the values that solve the above problem. Then we can conclude:

$$\hat{\alpha}_{\mathcal{A}}(\alpha, \mathcal{D}) - \alpha = \hat{\alpha}_{\mathcal{A}}(1, \mathcal{D}) - 1 \quad \text{and} \quad \hat{\beta}_{\mathcal{A}}(\alpha, \mathcal{D}) = \hat{\beta}_{\mathcal{A}}(1, \mathcal{D}).$$

2.6.3.2 Other algorithms

It is clear that the linear regression algorithm satisfies Property 3. In the piecewise minimum algorithm, the increase in measured delay due to a clock skew is a function of the sender timestamp, but not of the end-to-end delay as stated in (2.6). As the skew gets larger, the increase due to the clock skew becomes the dominant part of a measured

delay, and the minimum of a segment is more likely to be found near the beginning of the segment. Depending on the magnitude of the skew, a minimum-based algorithm uses different minima, and clearly the differences between the estimate and the actual skew are not the same.

Since Paxson's algorithm employs a robust line fitting technique after local minima are obtained, we choose to simulate Paxson's algorithm to examine its robustness. In the simulation, the number of packets is 600, and α changes from 0.01 to 0.1 and 4. The end-to-end delay consistent with C_r, $\bar{d}_i$, is assumed to have an exponential distribution with the means, $\mu = 10msec$ and $\mu = 100msec$. The purpose of the simulation is to show the variability of the difference between the estimate and the actual skew over a range of clock skews. Thus we use the same set of $\bar{d}_i$ for all three values of α, and calculate the sample mean and the variance of the estimates. As shown in Section 2.6.3.1, the difference between the actual skew and the estimate does not change in the LP algorithm case, and thus the sample variance of the estimates from the algorithm remains the same for the three values of α in the simulation. The sample variance of Paxson's algorithm, however, increases as the skew increases. We list only the sample variances in Table 2.1. This illustrates that the difference between the actual skew and the estimate of Paxson's algorithm grows as the skew grows, and thus the algorithm does not satisfy Property 3.

2.6.4 Measurement

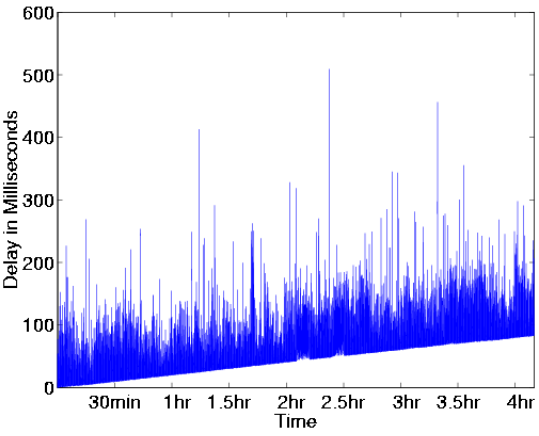
In this section we use Internet delay measurements to see how each algorithm performs when applied to actual measurements. We collected several traces of delay and loss measurements over the Internet and MBone [14] between November 14, 1997, and December 21, 1997. Table 2.2 provides a brief description of the traces. Detailed information about hosts are in Appendix B. The clocks at the end-hosts were not synchronized in any of the traces. We used constant-length UDP packets whose payloads consisted of a sequence number and a timestamp, and they were sent out at periodic intervals.

α -1	$\mu = 10msec$			
	LP		Paxson	
	mean	var.	mean	var.
0.01	0.01	0.9666e-5	0.01	0.0373e-3
0.1	0.1	0.9666e-5	0.1	0.1073e-3
4	4	0.9666e-5	4.0001	0.3945e-3
α -1	$\mu = 100msec$			
	LP		Paxson	
	mean	var.	mean	var.
0.01	0.01	0.8405e-4	0.01	0.0002
0.1	0.1	0.8405e-4	0.1001	0.0004
4	4	0.8405e-4	4.0001	0.002

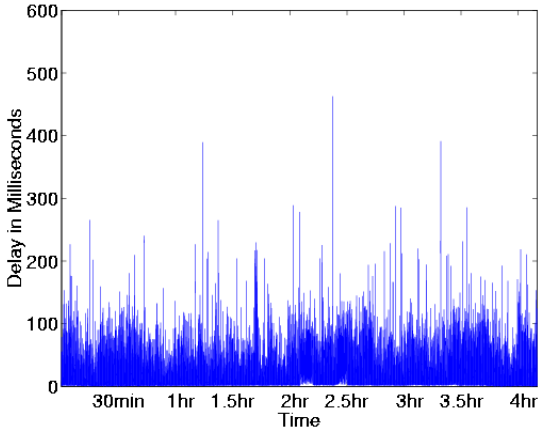
Table 2.1. Sample variance of estimates from simulations to test Property 3

Trace	Sender	Receiver	Start Time at Sender	Duration	Type	Interval
2.1	UMass	SICS	12:50pm, Fr, 11/14/97	4hr 10min	Unicast	80ms
2.2	UMass	WashU	3:54pm, Th, 11/20/97	10hr	Multicast	160ms
2.3	UMass	SICS	12:03am, Fr, 11/21/97	5hr 54min	Unicast	160ms

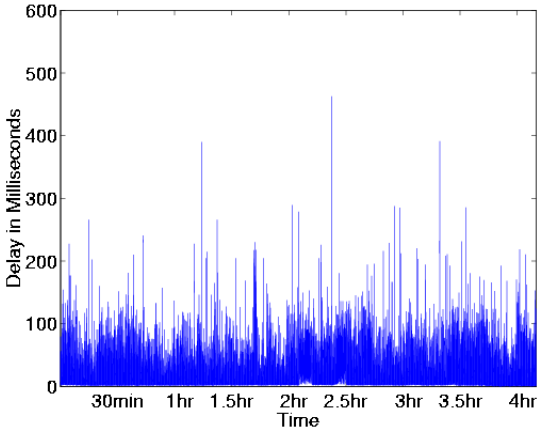
Table 2.2. Traces Used in Chapter 2



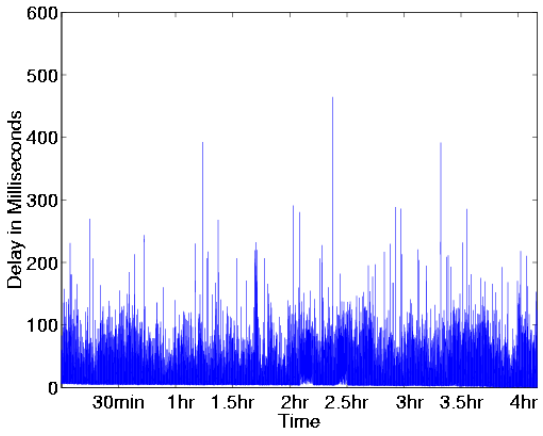
(a) Original Trace



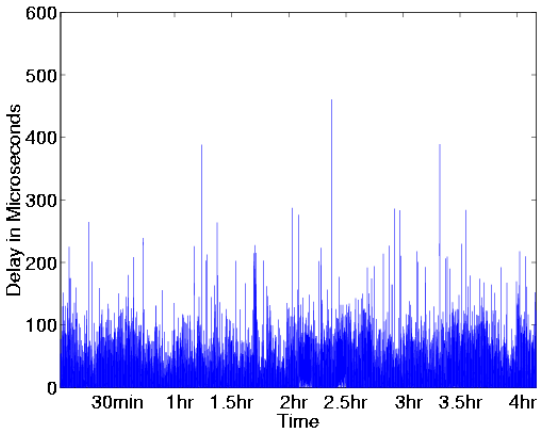
(b) LP algorithm



(c) Paxson's algorithm

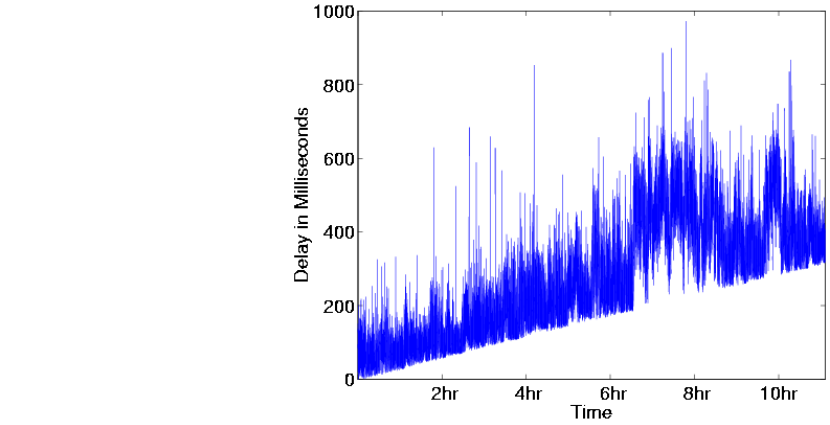


(d) Linear regression algorithm

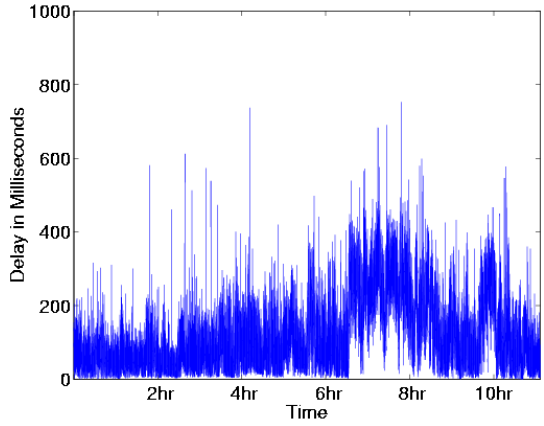


(e) Piece-wise minimum algorithm

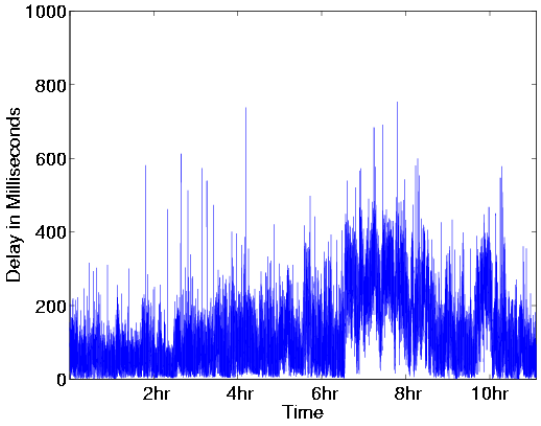
Figure 2.5. Scatter-Plots of Delay from Trace 2.1 Before and After the Skew Removal.



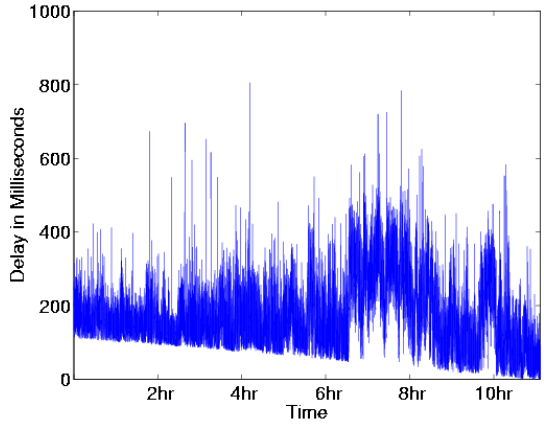
(a) Original Trace



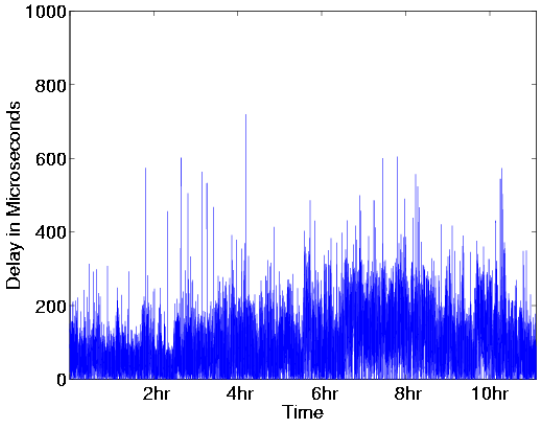
(b) LP algorithm



(c) Paxson's algorithm



(d) Linear regression algorithm



(e) Piece-wise minimum algorithm

Figure 2.6. Scatter-Plots of Delay from Trace 2.2 Before and After the Skew Removal.

Trace 2.1 in Table 2.2 exhibits an increasing trend in delay measurements, which translates to a constant skew. Figure 2.5(a) plots the original trace before the skew is removed. All four algorithms estimate the skew well, and the delays after the skew is removed are plotted in Figures 2.5(a) to (d). On the x-axis the sender timestamp, $\tilde{t}_i^s$ is plotted, and on the y-axis, the delay before and after the skew estimation and removal, $\tilde{d}_i$ and $\tilde{d}_i - (\hat{\alpha} - 1)\hat{t}_i^s$, are plotted.[1] The gray foreground is the delay before the skew is estimated and removed, and the black background is the delay after the skew is removed.

Figure 2.6 is from Trace 2.3 in Table 2.2. The same is on the x and y axes, as in Figure 2.5. The linear skew trend is apparent as in Trace 2.1, but the delay behavior changes significantly in the later half of the trace, and more losses are detected. This is a multicast packet delay measurement, and the overall loss rate of the trace is very high: 42%. The losses are more pronounced as the jagged bottom of the gray plot in the second half of the trace. Considering the extraordinary high loss rate of the trace, we think that the clock skew was constant, but due to heavy congestion inside the network, queueing delays increased significantly over an extended period of time, and is shown in measurements.

In Figure 2.6(d) the linear regression algorithm fails miserably in estimating the skew. The large delays between 6 and 8 hours on the x-axis produce outliers which have a significant impact on the linear regression. After the skew is filtered out, the delay has a decreasing trend, which is the opposite of the original increasing trend. Since the LP and Paxson's algorithms come up with one estimate for the constant skew, the resulting delays from both algorithms are close to the x-axis, while keeping the increased delay trend intact. The piecewise minimum algorithm calculates too high a minimum over the increased delay period, and ends up interpreting the increased delay trend as a skew. The result is that the effect of network congestion on delay is removed along with the skew.

[1]Here we actually plot $\tilde{d}_i - \min_i \tilde{d}_i$ and $\tilde{d}_i - (\hat{\alpha} - 1)\hat{t}_i^s - \min_i \tilde{d}_i$ to plot the delays before and after the skew estimation and removal in the same range of values on the y-axis.

Figures 2.5 and 2.6 visually demonstrate the relative performance of each algorithm. The LP and Paxson's algorithms estimate the skew accurately in the presence of different levels of network congestion. In contrast, the linear regression and piecewise minimum algorithms perform poorly when the network is heavily congested, and the delay fluctuates significantly. In order to investigate the relative performance more precisely, we simulate the LP and Paxson's algorithms with synthetic delay, and compare the sample mean and variance of the estimate in the next section.

2.6.5 Simulation

The purpose of the simulation is to examine the average performance of a skew estimation algorithm in terms of the sample mean and variance of the estimate. We have performed two sets of simulations for each of the LP and Paxson's algorithms. The first set of simulations (referred to as Set 2.1) uses a periodic inter-packet departure time of 20 milliseconds; the second set (Set 2.2) uses a periodic inter-packet departure time of 1 second. The other parameters of the simulation are the number of packets, the mean delay (μ), the skew. The number of packets used is either 600 or 3000. In Set 2.1, 600 packets correspond to 12 seconds and 3000 to 1 minute; in Set 2.2, to 10 minutes and 1 hour, respectively. We assume an exponential distribution for end-to-end delays in our simulation and vary the mean (μ) of the exponential distribution from 1 to 10, 100, and $1000msec$. The clock ratio (α) varies from 1.0001 to 1.1 and 5.

For a set of fixed values of the number of packets, mean delay, and skew, 1000 runs were executed, and the histograms of the skew estimates from the 1000 runs are plotted in Figures 2.7 to 2.10. We plot $\hat{\alpha} - 1$ on the x-axis, and the frequency on the y-axis in Figures 2.7 to 2.10. The histograms have a fixed bin size of 30; the greater distance between the minimum and maximum of the estimates is, the wider the bin is.

Figures 2.7 and 2.8 plot the results of the LP and Paxson's algorithms from Set 2.1. Figures 2.9 and 2.10 from Set 2.2. The first rows of Figures 2.7 to 2.10 is for $\alpha - 1 =$

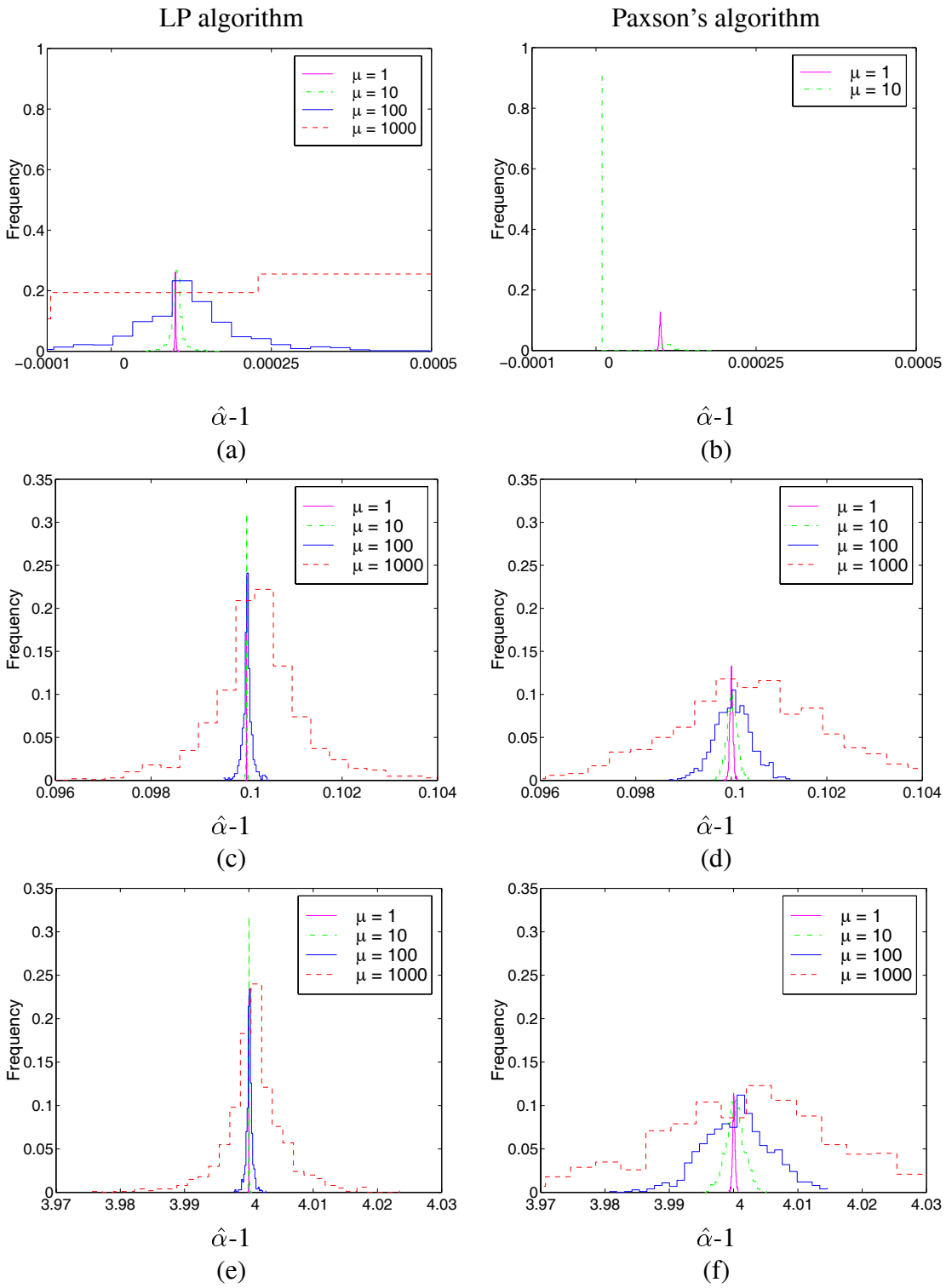


Figure 2.7. Set 2.1 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 600

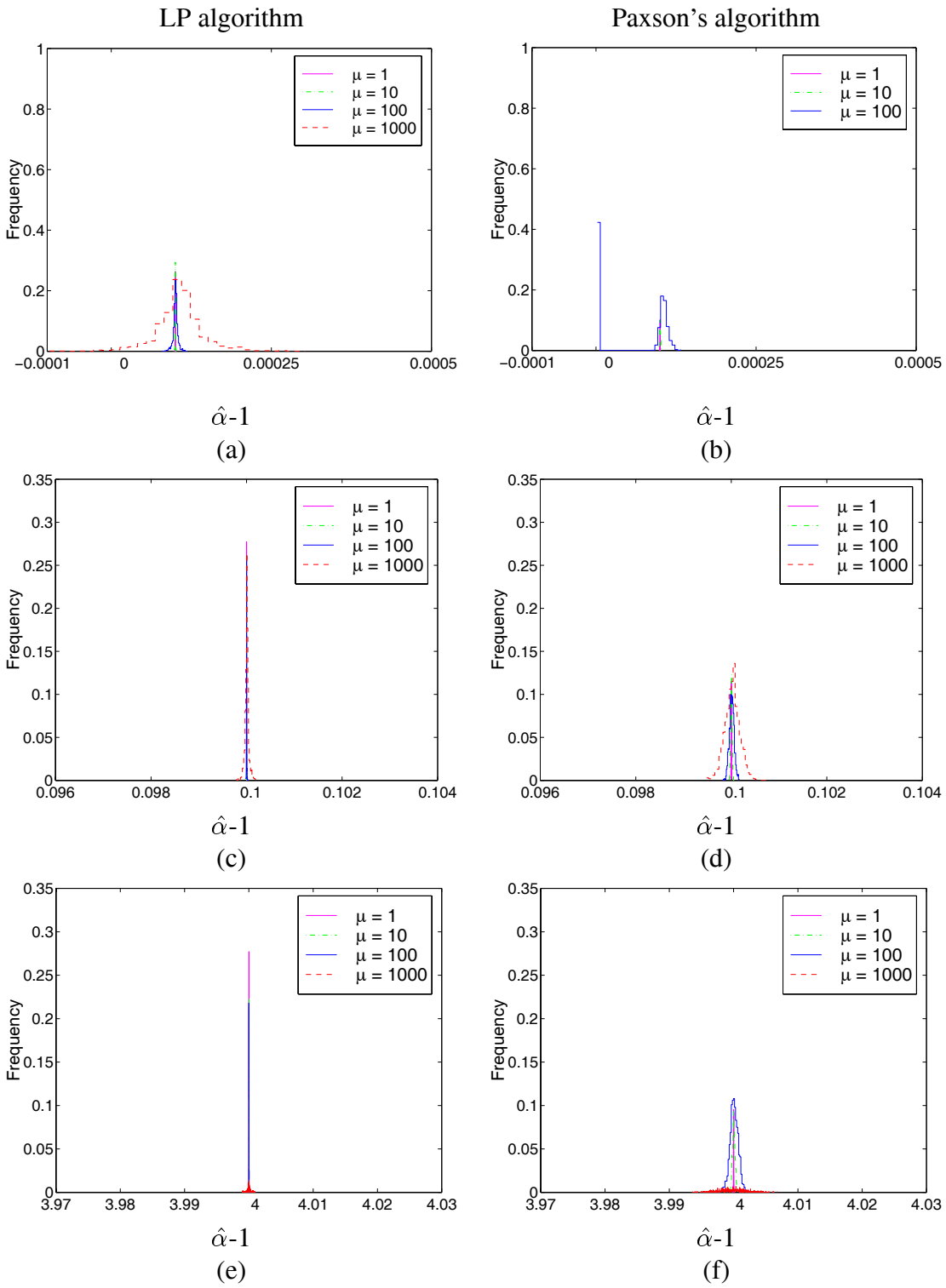


Figure 2.8. Set 2.1 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 3000

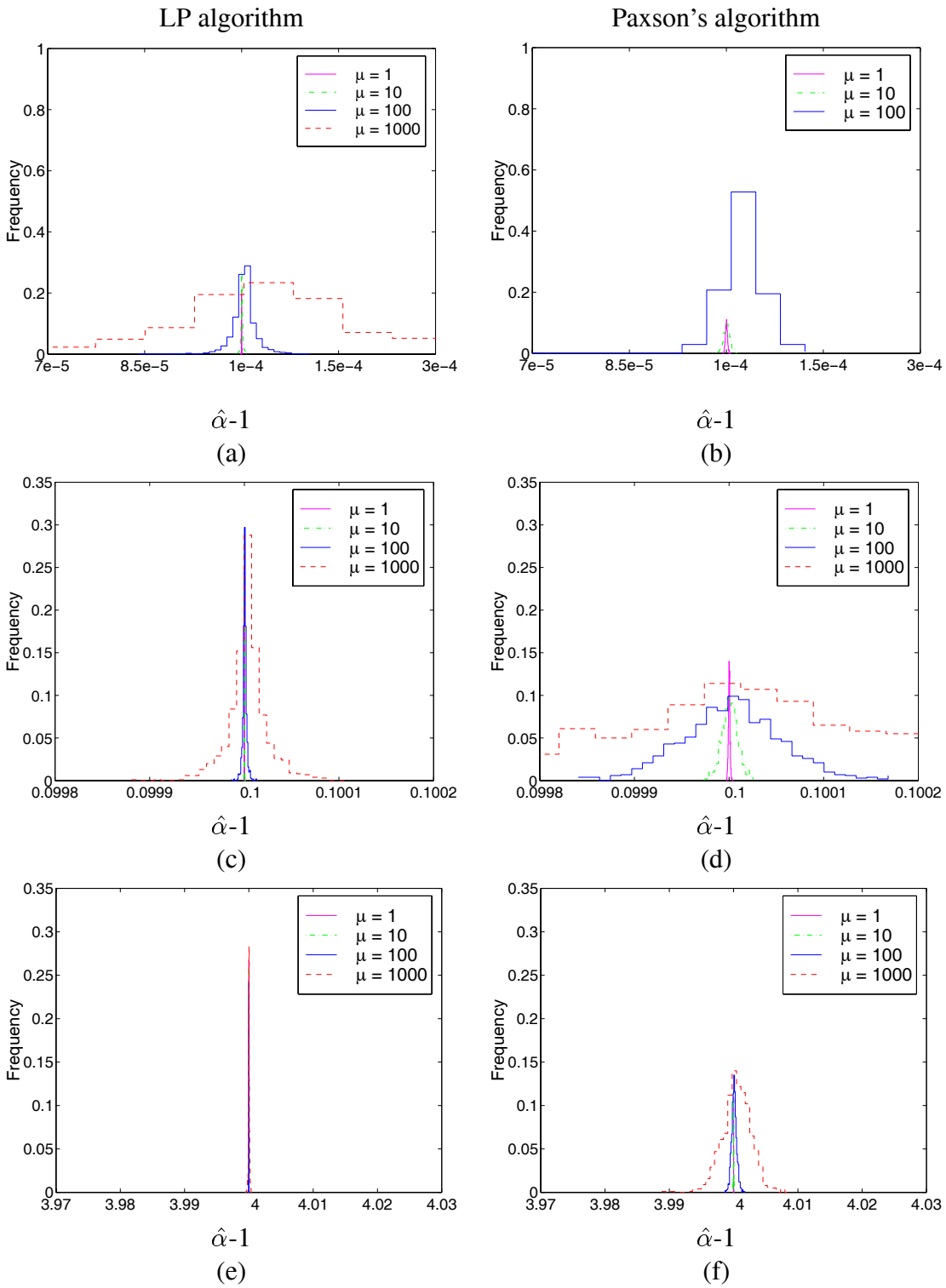


Figure 2.9. Set 2.2 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 600

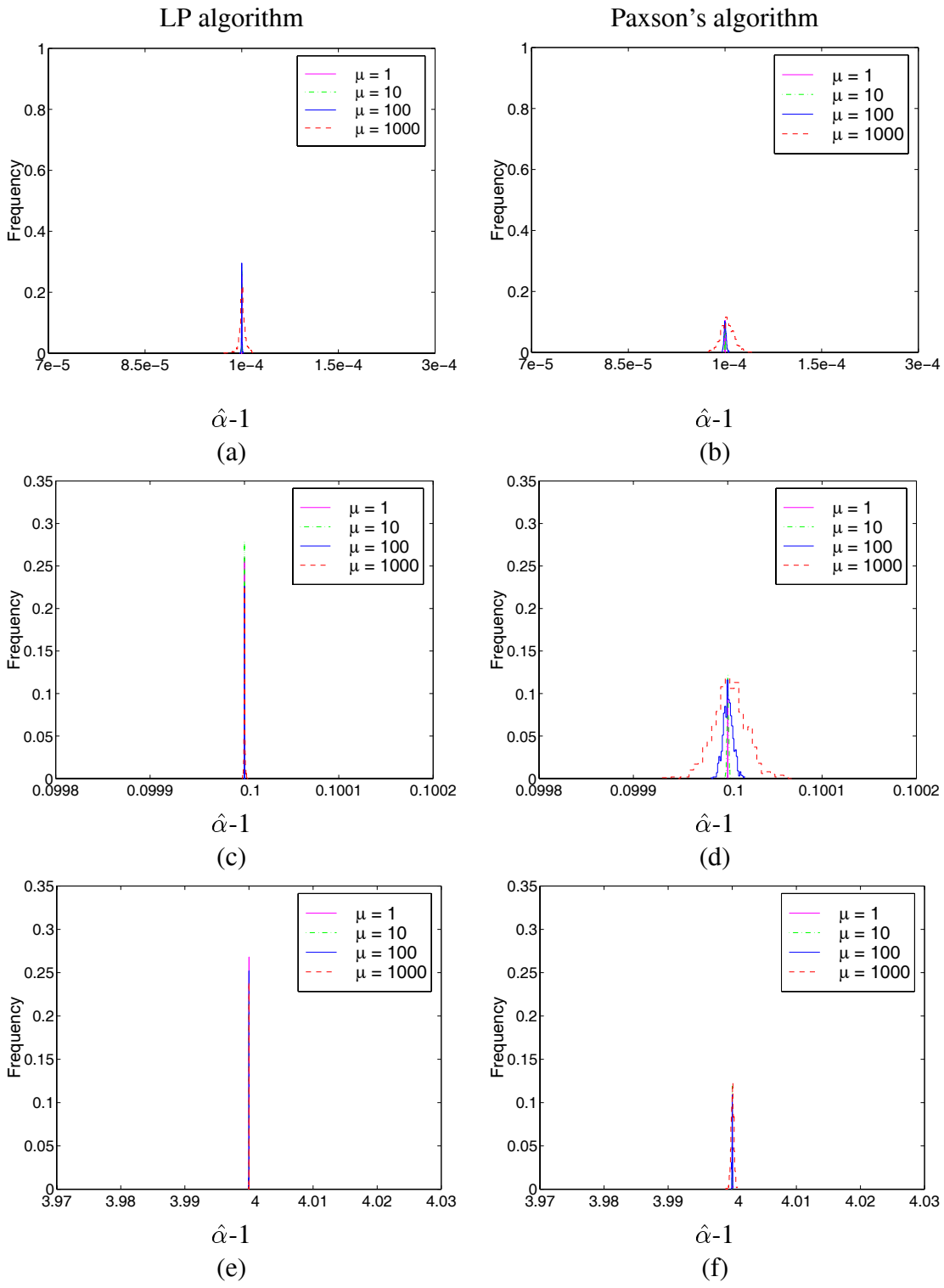


Figure 2.10. Set 2.2 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 3000

0.0001, the second rows for $\alpha - 1 = 0.1$, and the last rows for $\alpha - 1 = 4$. The ranges on the x-axis and y-axis are fixed for a given α in a set to make it easy to compare how the histograms spread horizontally between the LP and Paxson's algorithms.

In both Sets 2.1 and 2.2, the histograms when the number of packets is 600 are more widespread than when the number of packets is 3000. This is because the estimate gets more accurate as more delay measurements are available.

Most histograms are symmetric centered at the mean delay, and their estimates are very close to the true α values with a sample variance less than $\pm 4\%$ of α. However, as α gets closer to 1, the distribution of estimates starts to diverge from a symmetrical shape in Paxson's algorithm. Figure 2.7(b) where the number of packet is 600 and $\alpha - 1 = 0.0001$, the histograms for $\mu = 100$ and $\mu = 1000$ are not plotted because their frequencies are 1 at $\hat{\alpha} - 1 = 0$. The same is true for $\mu = 1000$ when the number of packets is 3000 in Set 2.1 (Figure 2.8(b)), and for $\mu = 1000$ when the number of packets is 600 in Set 2.2 (Figure 2.9(b)). This is because of Step 3 in Paxson's algorithm. If the number of cumulative minima is not significant enough, Paxson's algorithm assumes no skew, and outputs $\hat{\alpha} = 1$. The simulations results show that it is hard to pass Step 3 and the algorithm is biased against a small skew, when the average delay is relatively large, and the measurements span over a short period of time.

The histograms of the LP algorithm are consistently less spread out than those of Paxson's algorithm for the given range of parameter values. We have shown through simulation that the estimate of our LP algorithm is unbiased, and exhibits less sample variance than that of Paxson's.

2.6.6 Performance Given a Small Number of Packets

To examine how the algorithms perform in a very small time scale, we partition Trace 2.1 into 10-minute-long intervals, and apply the LP and Paxson's algorithms to the segments of the trace. When the number of packets used in clock skew estimation is small,

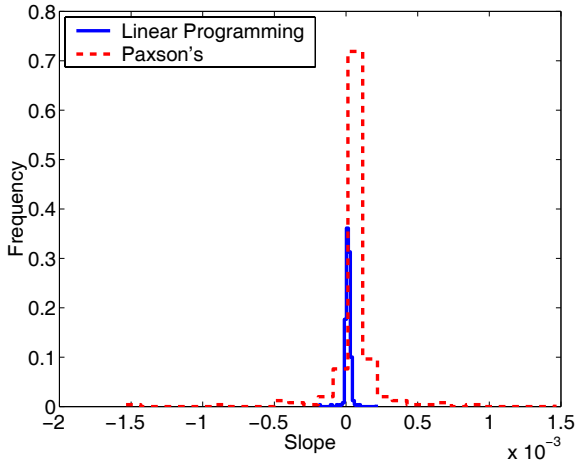


Figure 2.11. Histograms of $\hat{\alpha} - 1$ from Trace 2.1 broken into 10-minute-long intervals.

Paxson's algorithm is very likely to declare that there is no trend, and thus no clock skew. To compare the estimates without the trend check in Paxson's algorithm, we do not execute the checks in lines 24-25 in Appendix A.2. Figure 2.11 shows the histograms of the clock skew estimates from the segments by the LP and Paxson's algorithms. Consistent with previous results in longer measurements and simulations, the LP algorithm exhibits a lower sample variance than Paxson's algorithm.

In this set of simulations, we number Set 2.3, we have used the same values for the inter-packet departure time, the mean delay, and the skew, as in Set 2.1, but with fewer packets. Instead of 600 and 3000 packets, we use 20 and 100 packets, which correspond to 400 milliseconds and 2 seconds. In Figures 2.12 and 2.13 we present the results from the simulations using 20 and 100 packets, respectively. The results are consistent with what is presented above using segments of actual measurements; the histograms of the LP algortihms are less spread out than those of Paxson's algorithm.

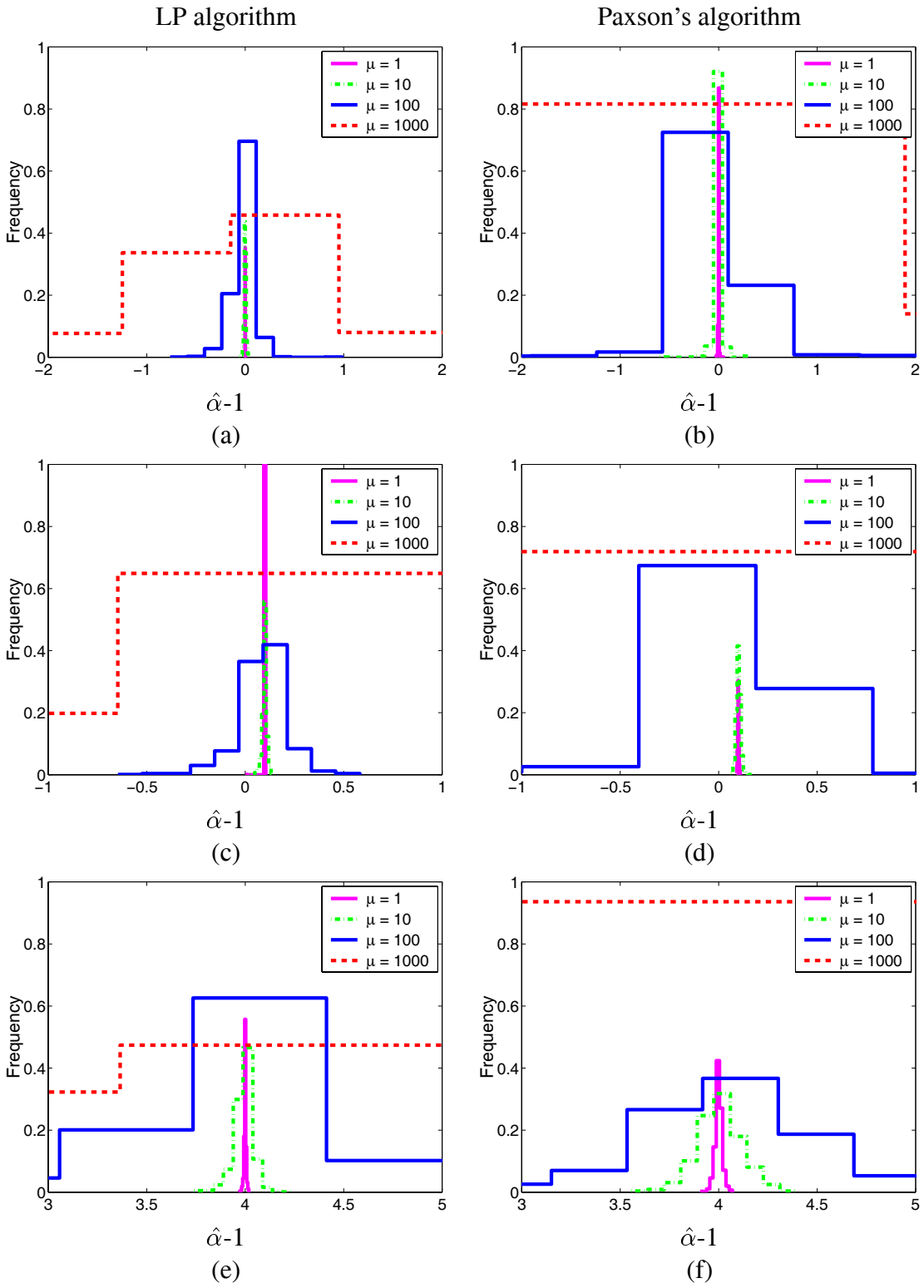


Figure 2.12. Set 2.3 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 20

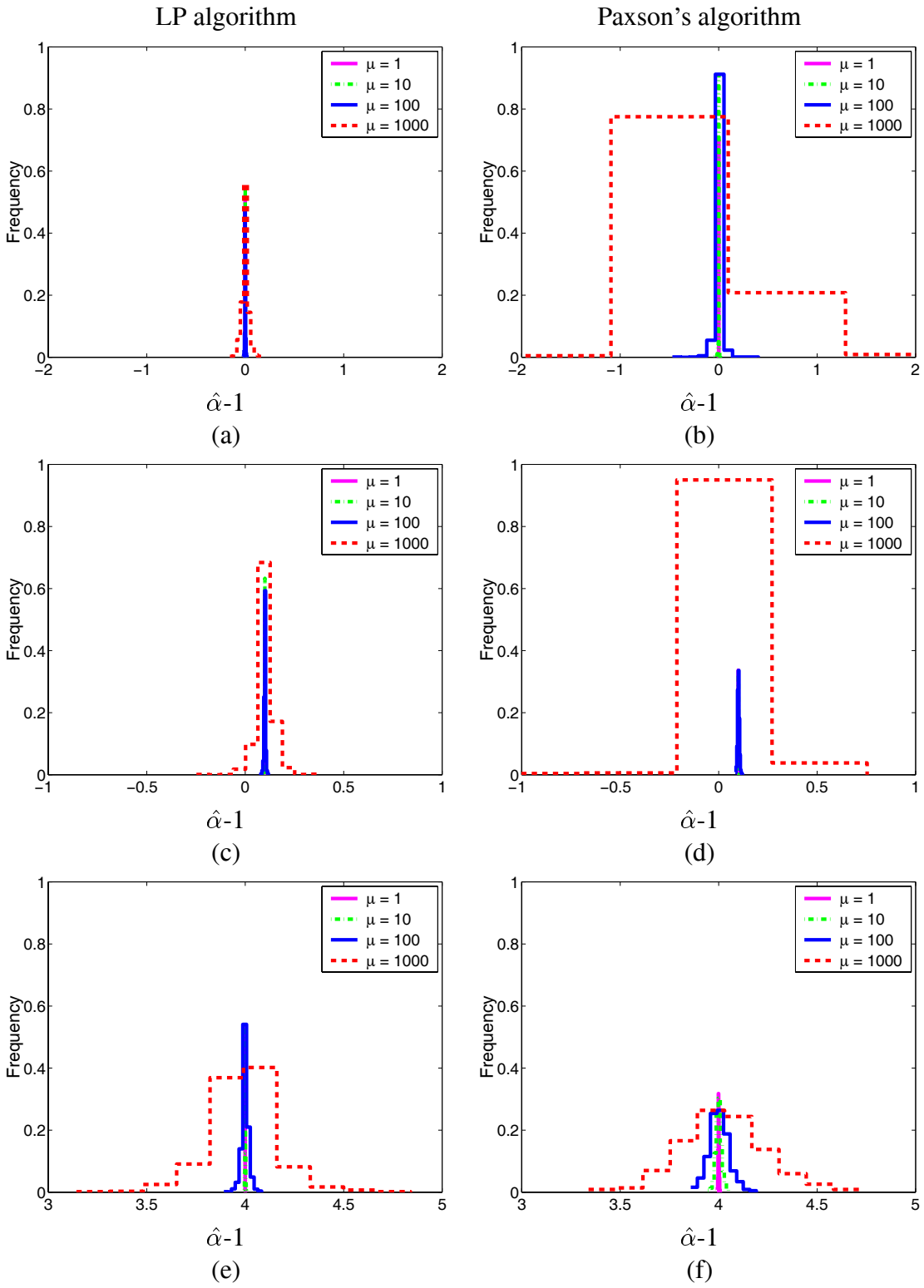


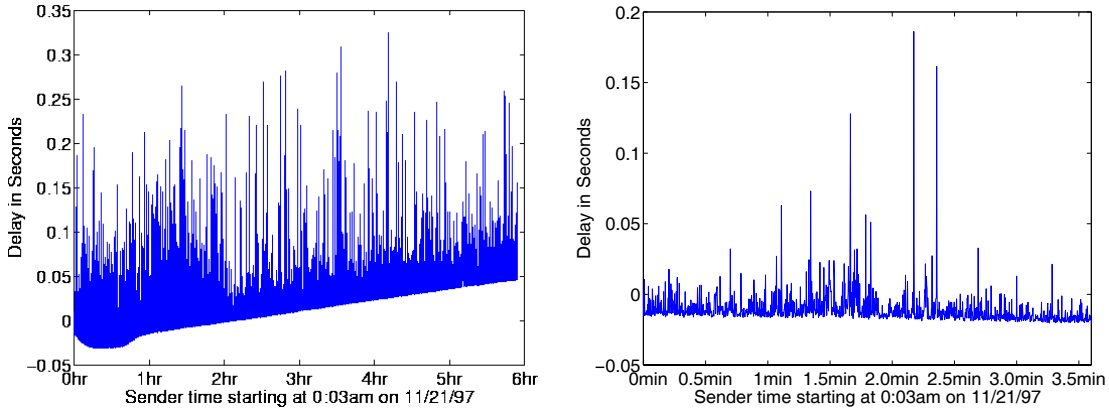
Figure 2.13. Set 2.3 - Histograms of $\hat{\alpha} - 1$ when the number of packets is 100

2.7 Further Discussion

2.7.1 Non-zero Clock Drift

In Traces 2.1 and 2.2 we have observed mostly linear skew trends. Trace 2.3 in Table 2.2 exhibits a non-linear linear trend. Figure 2.14(a) displays $\tilde{t}_i^s$ on the x-axis and $\tilde{d}_i$ on the y-axis from Trace 2.3. The slope of the delays varies over time in the first hour of the trace. The drift, as defined in Section 2.2.1, is non-zero. It attests to a slowly varying skew present in the first hour of the trace; and then a linear trend sets in and lasts until the end of the trace. Figure 2.14(b) is a 100 times magnified view of the first 3.5 minutes in Figure 2.14(a). The slowly varying drift in a large time scale of hours is not noticeable in the magnified graph. Instead we only see a constant skew trend.

This suggests that we can apply the constant skew estimation algorithms in a piece-wise manner on a smaller time scale. How to detect a non-zero drift, and to determine the proper size to partition a long trace are important issues, and require further study.



(a) Non-zero drift

(b) Non-zero-drift: magnified view

Figure 2.14. Scatter-plot of Trace 2.3 that exhibits a non-linear trend.

2.7.2 Detection of clock adjustments

When a clock adjusts its time, the adjustment is likely to show up in delay measurements. Paxson uses two-way measurements to detect such a clock adjustment [46, 47]. It

is harder to distinguish a genuine clock time adjustment from the delay fluctuation due to network congestion in one-way delay measurements, and needs further study.

2.8 Conclusion

In this chapter, we have presented a framework for understanding the systematic errors introduced in one-way network delay measurements by unsynchronized clocks, and discuss several properties desirable of a skew estimation algorithm. We have developed a linear-programming-based algorithm, and compared it with three other existing algorithms. The linear regression and piecewise minimum algorithms demonstrated poor performance when applied to traces of Internet delay measurements. We generated synthetic delay measurements, and analyzed the sample mean and variance of the estimates of our and Paxson's algorithms. The results show that the estimate of the LP algorithm is unbiased and exhibits less variance. In conclusion, the LP algorithm addresses all of the desirable properties, and is simple, fast, and robust.

CHAPTER 3

ADAPTIVE PLAYOUT DELAY ADJUSTMENT

3.1 Introduction

In the 20 years that have passed since the early Arpanet experiments with packetized voice [10], packetized audio has blossomed into an application that many Internet users now use regularly. For example, the audio (and video and whiteboard) segments of many technical conferences and workshops are now carried over the MBone multicast network [8, 22, 31]. Smaller, more interactive, group meetings are also frequently conducted over the Internet using these multimedia tools.

Packet audio tools such as NeVoT [54], vat [23], and RAT [17, 28] operate by periodically gathering audio samples generated at the sending host, packetizing them, and transmitting the resulting packet (via UDP unicast/multicast) to the receiving site(s). For efficiency, a source's audio is typically divided into "talkspurts" (periods of audio activity) and "silence periods" (periods of audio inactivity, during which no audio packets are generated). In order to faithfully reconstruct the audio at a receiving site, data in packets within a talkspurt must be played out in the same periodic manner in which they were generated.

If the underlying network is free of variations (jitter) in packet delays, a receiving site can simply play out an audio packet as soon as it is received. However, jitter-free, in-order, on-time packet delivery rarely, if ever, occurs in today's packet-switched networks. In order to compensate for these variable delays, a smoothing buffer is thus typically used at a receiver. Received packets are first queued into the smoothing buffer and the periodic playout of packets within a talkspurt is delayed for some amount of time beyond the

reception of the first packet in the talkspurt. Informally, we refer to this delay as the *playout delay* of the talkspurt. Clearly, the longer the playout delay, the more likely it is that a packet will have arrived before its scheduled playout time. Excessively long playout delays, however, can significantly impair human conversations. There is thus a critical tradeoff between the length of playout delay and the amount of loss (due to late packet arrival) that is incurred. Generally, delays between talkspurt generation and receiver playout of less than 400ms [20] and a loss percentage of up to 5% [24] are considered to be quite tolerable in human conversations. The talkspurt playout delays themselves can be either fixed for the duration of the audio session (an approach examined in [37, 10]), or adaptively adjusted from one talkspurt to the next, with intervening silence periods artificially elongated or compressed accordingly – the approach taken in the NeVoT and vat audio tools.

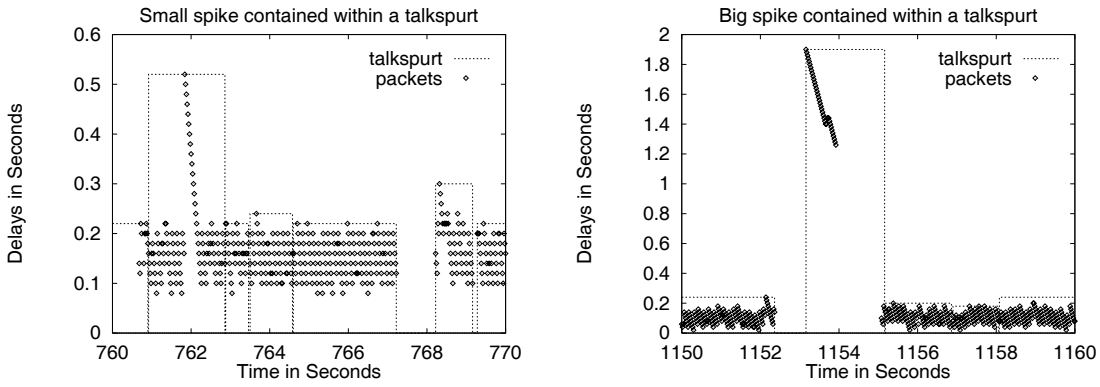
In this chapter we focus on this tradeoff between packet playout delay and packet loss. The main contributions of this chapter are twofold. First, given a trace of packet audio receptions at a receiver, we present efficient algorithms for computing upper and lower bounds on the optimum (minimum) average playout delay for a given number of packet losses (due to late arrivals) at the receiver for that trace. These bounds, which we show to be tight for a range of loss and delay values of interest, are of particular importance as they provide a bound on the achievable performance of *any* adaptive playout delay adjustment algorithm. Our second significant contribution is the development of a new adaptive delay adjustment algorithm that tracks the network delay of recently received packets and efficiently maintains delay percentile information. This information, together with a "delay spike" detection algorithm based on (but extending) earlier work [48], is used by the new algorithm to dynamically adjust talkspurt playout delay. We show that this new algorithm generally outperforms existing delay adjustment algorithms over a number of measured audio delay traces and performs close to the theoretical optimum over a range of parameter values of interest.

The remainder of this chapter is structured as follows. Section 3.2 provides additional background for our work, including an extended discussion of the observed delay spikes in the packet audio traces reported earlier in [48] as well as in new, more recent experimental traces reported here. In Section 3.3 we describe the algorithms used to compute bounds on the optimum average playout delay for a given loss. In Section 3.4 we present our new adaptive playout delay adjustment algorithm and examine its performance. We conclude this chapter in Section 3.5.

3.2 Background

As discussed above, a receiving site in an audio application typically buffers packets and delays their playout [1, 37] in order to compensate for variable network delays. The playout delay can be constant throughout the entire audio session or can be adaptively adjusted during the session from one talkspurt to the next. In the Internet, end-to-end delays fluctuate significantly [4, 50] and a constant, non-adaptive, playout delay would likely yield unsatisfactory audio quality for interactive audio applications. There are two approaches to adaptive playout adjustment: per-talkspurt and per-packet adjustment. The former approach uses the same playout delay throughout a talkspurt (and, as a result, faithfully reconstructs the original periodic nature of the received audio data from the sender), but allows different playout delays from one talkspurt to another. While this may result in artificially elongated or compressed silence periods, this is not noticeable in played out speech if the change is reasonably small [37]. In the latter approach, the playout delay varies from packet to packet. A per-packet adaptive adjustment introduces gaps inside talkspurts and is cited as being damaging to the audio quality [1, 10].

Because of the variable nature of network delays that gives rise to the need for playout delay adjustment algorithms, an understanding of network delays and their effects on packet audio at both the individual packet and talkspurt level is important. It will thus be instructive to first informally examine a few traces of actual audio traffic and identify



(a) Small Spike (b) Big Spike

Figure 3.1. Delay spikes in scatter-plot of delay measurements over time.

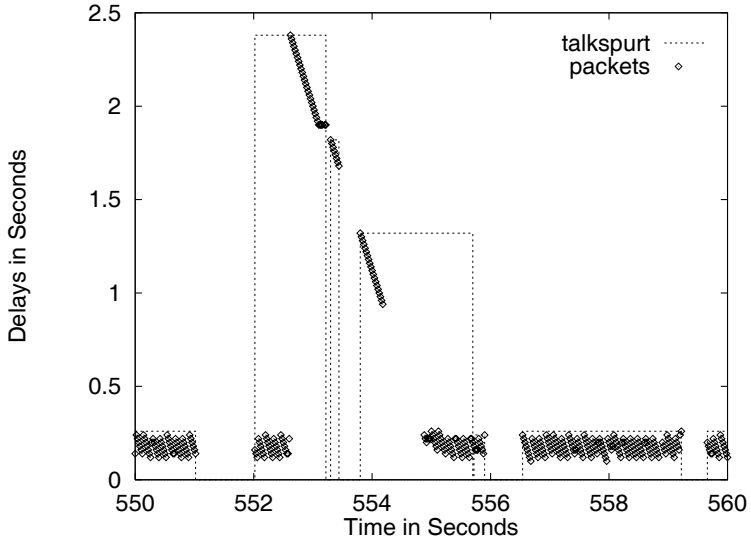


Figure 3.2. Delay Spike spanning several talkspurts

a number of characteristic aspects of the interaction between network delay and packet audio playout.

Fig. 3.1 and 3.2 plot the variable portion of the delay between a packet's transmission at a sender and its reception at a receiver as a function of the time at which the packet was transmitted at the sender. The propagation component of the end-to-end delay has been removed by subtracting out the minimum of the measured end-to-end delays in the entire delay trace (presumably the case in which there is little or no queueing of the packet in intermediate routers). Note that by considering only the variable delay component, the

Trace	Sender	Receiver	Start time at Sender	Duration	Multicast
3.1	UMass	GMD	08:41pm, Tu, 6/27/95	1348 secs	No
3.2	UMass	GMD	09:58am, Fr, 7/21/95	1323 secs	Yes
3.3	UMass	GMD	11:05am, Fr, 7/21/95	1040 secs	No
3.4	INRIA	UMass	09:20pm, Th, 8/26/93	580 secs	No
3.5	UCI	INRIA	09:00pm, Sa, 9/18/93	1091 secs	No
3.6	UMass	Osaka	00:35am, Fr, 9/24/93	649 secs	No

Table 3.1. Traces Used in Chapter 3

issue of sender and receiver clock synchronization can be avoided. The variable delay component of each packet is plotted as a diamond on the graph. Dotted-line rectangles are used to distinguish talkspurts from each other showing which packets belong to which talkspurt. The width of a rectangle in the figures represents the length of a talkspurt and its height represents the largest variable portion of network delay over all packets within that talkspurt. A packet is generated every 20ms during a talkspurt, and hence a missing dot at a 20ms interval within a talkspurt indicates a lost packet within the network.

The delay traces shown in Figure 1, as well as all other traces reported in this chapter, were collected using NeVoT [54], an audio conferencing tool that allows both point-to-point or multicast connections. NeVoT has a tracing mechanism that can collect timestamps of packets sent and received, RTP sequence numbers [53], and vat virtual timestamps of packets. In our experiments and simulations we used vat virtual timestamps. Packet audio was encoded in 8KHz PCM mode and the packetization unit time was 20ms. The sending and receiving hosts, the start time and date of the trace, the trace length, and an indication of whether packets were sent as unicast or multicast packets are indicated in Table 3.1. Traces 3.4, 3.5, and 3.6 are from earlier work, and are described in details in [48]. Traces 3.1 through 3.3 are new traces consisting of the transmission of the audio component of a recording with both female and male voices. Fig. 3.1 and 3.2 are all taken from Trace 3.1 in Table 3.1.

Delay spikes are evident in Figures 3.1 and 3.2. Figure 3.1(a) shows a spike whose delay is less than an order of magnitude greater than other "baseline" delays and whose duration is short enough to be contained in a single talkspurt. Figures 3.1(b) and 3.2 show larger spikes with delay peaks that are almost an order of magnitude larger than the "baseline" delays. A large spike can either be contained in one talkspurt, as in Figure 3.1(b), or can span several talkspurts, as in Figure 3.2. In Trace 3.1 of Table 3.1, there are 23 such conspicuously large spikes; 10 of these are contained in one talkspurt, 9 span two talkspurts, and the remaining 4 span three talkspurts.

Previous studies [4, 50, 48] have indicated the presence of "spikes" in end-to-end Internet delays. Bolot [4] conjectures that with periodically generated packets (as is the case with our audio packets and as was the case in [4, 50, 48]), the initial steep rise in the delay spike and the linear, monotonic decrease spike after the initial rise, is due to "probe compression" – the accumulation of a number of packets from the connection under consideration (the audio session, in our case) in a router queue behind a large number of packets from other sources. We note that probe compression is a plausible *conjecture* about the cause(s) of delay spikes. Validation of this conjecture would require careful measurements of packet traffic and its delay at the routers where congestion occurs. Kasera et al. [26] discuss the many difficulties involved in making such measurements without privileged access to the routers.

Note that when a delay spike is properly contained within a talkspurt, the next opportunity to change the playout delay (i.e., at the beginning of the next talkspurt) occurs *after* the delay spike terminates. In such a case, it is not possible to adaptively react to the delay spike, since the delay spike is already over (i.e., the delay has returned to its baseline value) by the next talkspurt and any packets that were so excessively delayed during the delay spike that they missed their playout time have already been lost. In cases where a delay spike spans multiple talkspurts, however, it *is* advantageous to quickly react to the delay spike, as discussed in [48]. Note also that the fluctuations in "baseline" delays

are less in comparison to spikes and, as a result, their delay distribution does not change significantly over time.

These two observations form the basis for the new delay adaptation algorithm to be presented in Section 4. First, however, we address the question of determining the playout delays incurred under a theoretically optimum playout delay adjustment algorithm. We do this in the following section.

3.3 Optimum Average Playout Delay

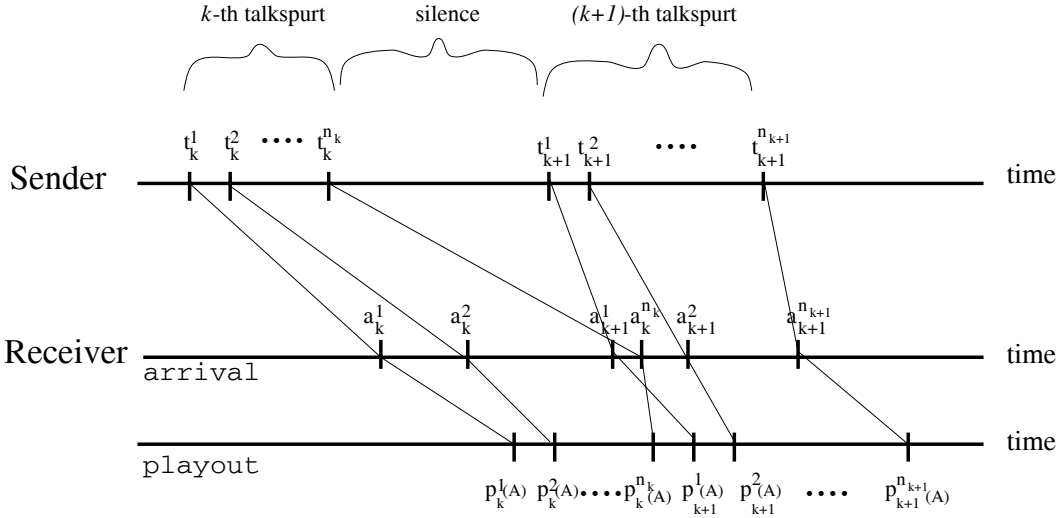


Figure 3.3. Timings associated with the i-th packet in the k-th talkspurt

In the previous works of [10, 37, 58, 48], the tradeoff between the average playout delay and loss due to late packet arrival is used as the performance measure in comparing one adaptive playout delay adjustment algorithm with another – a tradeoff which we also use in this chapter. We have chosen to consider loss and delay on a per-packet rather than per-talkspurt basis for two reasons. First we note that the lengths of talkspurts depend on silence detection algorithms and their parameters. Per-talkspurt results are thus closely tied to the silence detection algorithm used. More importantly, different talkspurts have different lengths. One might argue that in determining an overall performance measure,

per-talkspurt measures could be weighted by the length of the talkspurt. In a sense, we are already doing so by considering individual per-packet delay and loss measures, and requiring that all packets within the same talkspurt be played out periodically.

Here a playout delay (or, more accurately, end-to-end application-to-application delay) is defined to be the difference between the playout time at the receiver and the generation time at the sender. We refer to Figure 3.3 to show the timing information of audio packets and formally define the average playout delay.

Consider a trace consisting of M talkspurts. We define the following quantities:

- t_k^i: sender timestamp of the i-th packet in the k-th talkspurt.
- a_k^i: receiver timestamp of the i-th packet in the k-th talkspurt.
- n_k: number of packets in the k-th talkspurt. Here we only consider those packets actually received at the receiver.
- N: total number of packets in a trace, $N = \sum_{k=1}^{M} n_k$.

The playout time of a packet depends on which algorithm is used at the receiver to estimate the playout delay of the packet. Consider a playout algorithm A. Then $p_k^i(A)$ is the playout timestamp of the i-th packet in the k-th talkspurt under A. When it is obvious which algorithm is used, we omit the parameter A. If the i-th packet of the k-th talkspurt arrives later than $p_k^i(A)$ (i.e., $p_k^i(A) < a_k^i$), it is considered lost. Otherwise, it is played out with the playout delay of $(p_k^i(A) - t_k^i)$. Let $r_k^i(A)$ be an indicator variable for whether the i-th packet of the k-th talkspurt arrives before its playout time, as computed by playout algorithm A:

$$r_k^i(A) = \begin{cases} 0, & \text{if } p_k^i(A) < a_k^i \\ 1, & \text{otherwise.} \end{cases}$$

The total number of packets played out under Algorithm A is denoted as $N(A)$ and computed using $r_k^i(A)$:

$$N(A) = \sum_{k=1}^{M} \sum_{i=1}^{n_k} r_k^i(A).$$

Then the average playout delay of those played-out packets is defined as:

$$\frac{1}{N(A)} \sum_{k=1}^{M} \sum_{i=1}^{n_k} r_k^i(A)(p_k^i(A) - t_k^i).$$

If there are N packets in a trace and, among them, $N(A)$ packets are played out under Algorithm A, the loss percentage l is:

$$l = \frac{N - N(A)}{N} \times 100.$$

Our goal in this section is to present a bound on the optimum (minimum) average playout delay for a given number of packet losses (due to late arrivals) at the receiver for a given packet delay trace. To illustrate this problem, suppose we are given a trace of sender and receiver timestamps of audio packets in an audio session. Suppose now that we are free to set the playout delays of the various talkspurts to whatever values we choose such that only one packet (in the entire trace) will be lost. That is, we want to lose a packet so that the average playout delay over all played-out packets in the trace is minimized. This provides a bound on the average delay achievable by *any* delay adjustment algorithm, given that only one out of all the received packets in the trace is lost. We then repeat this procedure for two packet losses, and so on.

The obvious way to calculate the exact minimum bound is as follows: for a given number of dropped packets, say i, determine all possible configurations of i lost packets, compute the average playout delay for each configuration, and compute the minimum of these average playout delays. The number of computations of i lost packets grows exponentially in N. We reduce the computational cost by instead deriving upper and

lower bounds on the optimum (minimum) playout delay for a given loss percentage, that requires an amount of computation that is polynomial in the number of packets in the trace.

Section 3.3.1 provides the background needed for presenting these algorithms and defines our notation. We provide a detailed description of our approach in Sections 3.3.2 and 3.3.3.

3.3.1 General Overview

Below we introduce additional terminology to be used in the following sections.

- $\hat{d}_k^i$: delay between the generation of the i-th packet of the k-th talkspurt at the sender and its reception at the receiver, namely $\hat{d}_k^i = a_k^i - t_k^i$. We do not assume that the sender and receiver clocks are synchronized, but do assume that they do not drift.
- $\hat{d}$: $\hat{d} = \min_{1 \le k \le M, 1 \le i \le n_k} \{\hat{d}_k^i\}$.
- d_k^i: normalized delay of the i-th packet of the k-th talkspurt. This accounts only for the variable portion of the end-to-end delay. We will use this normalized delay (rather than $\hat{d}_k^i$) in calculating the bounds of the optimum average playout delay,

$$d_k^i = \hat{d}_k^i - \hat{d}.$$

- $d_k^{(i)}$: i-th smallest normalized delay in the k-th talkspurt.

Recall that the playout delay of all packets in the k-th talkspurt should be the *same* due to the periodic nature of packet generation within a talkspurt at the sender and periodic

playout at the receiver. Given an algorithm A, we denote the playout delay of the k-th talkspurt as $\hat{p}_k(A)$. The playout time of the i-th packet in the k-th talkspurt is then:

$$p_k^i(A) = t_k^i + \hat{p}_k(A). \tag{3.1}$$

In later sections where there is no confusion about which algorithm is used, we will denote $\hat{p}_k(A)$ simply as $\hat{p}_k$.

To successfully play out i packets from the k-th talkspurt, at least i packets during the k-th talkspurt must arrive before their playout time calculated by (3.1). To achieve this goal, the playout delay of an algorithm must be set larger than or equal to $d_k^{(i)}$. In practice (i.e., in an actual on-line implementation), $d_k^{(i)}$ cannot be known in advance before all the packets belonging to the k-th talkspurt arrive, but $p_k^i(A)$ must often be determined before these packets arrive. Since our bounding algorithms are off-line algorithms, we assume that t_k^i and a_k^i are available at the start of their executions.

The packet arrival times in a talkspurt are not the only quantities that determine the playout delay. The long playout delay of one talkspurt may force the playout of packets of the subsequent talkspurt to be further delayed. For example, consider a playout delay algorithm A. Assume that, in order to play out all packets contained in the k-th and $(k+1)$-th talkspurts, algorithm A sets the playout delays $\hat{p}_k(A)$ and $\hat{p}_{k+1}(A)$ to $d_k^{(n_k)}$ and $d_{k+1}^{(n_{k+1})}$, respectively. The playout time of the first packet in the $(k+1)$-th talkspurt, $p_{k+1}^1(A)$, becomes $t_{k+1}^1 + \hat{p}_{k+1}(A)$. If the playout time of the first packet of the $(k+1)$-th talkspurt comes before the playout time of the last packet of the k-th talkspurt, i.e., $t_k^{n_k} + \hat{p}_k(A) > t_{k+1}^1 + \hat{p}_{k+1}(A)$, then the beginning of the $(k+1)$-th talkspurt overlaps the end of the k-th talkspurt at the receiver. We refer to this as a *collision* of the k-th and $(k+1)$-th talkspurts. The condition for a collision can be summarized as:

$$t_k^{n_k} + \hat{p}_k(A) \quad > \quad t_{k+1}^1 + \hat{p}_{k+1}(A), \quad \text{or}$$

$$\hat{p}_k(A) \quad > \quad (t_{k+1}^1 - t_k^{n_k}) + \hat{p}_{k+1}(A). \tag{3.2}$$

In order to avoid such collisions, the playout delay of the subsequent talkspurt must be increased. Note that collisions can occur in a cascade when the above collision condition persists over several talkspurts in a row. We call such a sequence of collisions a *collision train*.

In order to provide a lower bound of the optimum average playout delay, we first simplify the problem by ignoring the effect of collisions and assume that packets of talkspurts in a collision are allowed to overlap. Note that this underestimates the optimum average playout delay (since, in practice, some talkspurts would be further delayed in order to avoid overlapping packets from different talkspurts), and thus represents a potentially unachievable lower bound on the minimum playout delay for a given loss. Later we will account for collisions. Our algorithms are based on dynamic programming [3].

3.3.2 Off-line algorithm without collisions

Our first algorithm provides a lower bound of the optimum average playout delay for a given loss percentage. Recall that it is obtained by ignoring additional delays due to collisions, i.e., the effect of one talkspurt's long playout delay on that of the subsequent talkspurt is not considered. We define $D(k, i)$ to be the minimum average playout delay possible when choosing i packets to be played out from the k-th to M-th talkspurts. Using dynamic programming, calculating $D(1, i)$ for i from 0 to N generates the lower bounds on the optimum average playout delay for loss percentages of 100% down to 0%. It is described by the following equation.

$$D(k,i) = \begin{cases} 0, & \text{if} \quad i = 0 \\ d_k^{(i)}, & \text{if} \quad k = M \quad \text{and} \quad i \leq n_M \\ \infty, & \text{if} \quad k = M \quad \text{and} \quad i > n_M \\ \min\limits_{0 \leq j \leq i} \left(((i-j)D(k+1, i-j) + jd_k^{(j)})/i \right), & \text{otherwise.} \end{cases} \tag{3.3}$$

In the following theorem, we prove that $D(k,i)$ in (3.3) is the minimum average playout delay when choosing i packets from k-th to M-th talkspurts to be played out, for the case that collisions are ignored.

Theorem 1 *$D(k,i)$ is the minimum average playout delay of choosing i packets to be played out from k-th to M-th talkspurts.*

Proof Assume that $D(x,y)$ is minimal for $k+1 \leq x \leq M, 0 \leq y \leq i-1$, but that $D(k,i)$ obtained via (3.3) is not. Here, j packets are chosen from the k-th talkspurt with the playout delay to be $d_k^{(j)}$, and $(i-j)$ packets from $(k+1)$-th to M-th talkspurts. Those $(i-j)$ packets have a playout delay of $D(k+1, i-j)$. Thus:

$$iD(k,i) > (jd_k^{(i)} + (i-j)D(k+1, i-j))$$

which contradicts the definition of $D(k,i)$ in (3.3), namely that:

$$jd_k^{(i)} + (i-j)D(k+1, i-j) \geq iD(k,i).$$

Thus $D(k,i)$ is minimal. ∎

3.3.3 Off-line algorithm with collisions

The second algorithm computes an upper bound on the optimum average playout delay. It relies on dynamic programming as in Section 3.3.2, but is more complicated

due to the manner in which it accounts for collisions. In the first algorithm, the playout delay of a talkspurt is simply $d_k^{(i)}$, given k and i at each step of computation in (3.3). To take collisions into account, condition (3.2) is checked for every k and i in the second algorithm. If there is a collision, the playout delay of the latter of the two colliding talkspurts is adjusted to a larger value to avoid a collision. Checking condition (3.2) for collisions requires not only the playout delays of two adjacent talkspurts but also the sender timestamps of the last and first packets of each talkspurt. The identity of the first and last packets played out in a given talkspurt vary, depending on which packets are chosen by the bounding algorithm to be played out. To track those packets played out at every step of the computation, we introduce the vector $C(k, i)$ whose components are the sets of packets belonging to talkspurts $k, \ldots, M$ that are played out. Here i denotes the total number of packets contained within these sets.

An informal description of the second algorithm is as follows. Define $D(k, i)$ as before. Given k and i, assume that $D(x, y)$ is known for $k + 1 \leq x \leq M, 0 \leq y \leq i - 1$. The calculation of $D(k, i)$ consists of choosing j packets from the k-th talkspurt, and $(i - j)$ packets from the $(k + 1)$-th to M-th talkspurts to be played out so as to minimize the average playout delay of those packets. If playing out j packets from the k-th talkspurt results in a collision between k-th and $(k+1)$-th talkspurts, then the playout delay of the $(k + 1)$-th talkspurt becomes larger and is accounted for when calculating the average playout delay. Choosing j packets from the k-th talkspurt may cause a cascade of collisions, in which case the playout delays are increased for all talkspurts involved in the collision, and if so, more delays are added to calculate the average playout delay. $C(k, i)$ records which packets are chosen for the minimum total sum at this step of computation.

Let us now introduce the additional notation used in the second algorithm. Throughout, $\mathbf{S} = (S_1, \ldots, S_k, \ldots, S_M)$ is an M-dimensional vector where $S_k \subseteq \{1, \ldots, n_k\}$ is a set of packets from the k-th talkspurt. If $|S_k| = l$, then S_k contains the identities (indices) of the l packets with the l smallest normalized delays in the k-th talkspurt. Henceforth,

$\mathbf{S}$ will be referred to as a playout vector. Let $\mathbf{e}_k(i,j)$ be an M-dimensional vector whose components are each an empty set, $\emptyset$, except for the k-th component which is the set of packets with the $(i+1)$-th through $(i+j)$-th smallest normalized delays in the k-th talkspurt. Last, if $\mathbf{S}$ and $\mathbf{X}$ are playout vectors whose components are sets, then $\mathbf{S} \cup \mathbf{X}$ is understood to be the vector whose components are the unions of the components in $\mathbf{S}$ and $\mathbf{X}$.

- $s_k(\mathbf{S})$: difference in the sender timestamps of the last packet played out from the k-th talkspurt and of the first packet from the $(k+1)$-th talkspurt, given $\mathbf{S}$. This value is used in adjusting the playout delay in the case of a collision. It is given as:

$$s_k(\mathbf{S}) = \begin{cases} 0, & \text{if } k = M, S_k = \emptyset, \text{ or } S_{k+1} = \emptyset \\ \min\{t_{k+1}^i : i \in S_{k+1}\} - \max\{t_k^i : i \in S_k\}, & \text{otherwise.} \end{cases}$$

  $s_k(\mathbf{S})$ can be interpreted as the length of the silence period at the sender between the k-th and $(k+1)$-th talkspurts consisting of S_k and S_{k+1}, respectively.

- $\hat{p}_k(\mathbf{S})$: playout delay of the k-th talkspurt when the playout vector is $\mathbf{S}$. It differs from $d_k^{(|S_k|)}$ in the case of a collision. It is given by the following recursion:

$$\hat{p}_k(\mathbf{S}) = \begin{cases} 0, & \text{if } S_k = \emptyset, \\ d_1^{(|S_1|)}, & \text{if } k = 1, \\ \max\{d_k^{(|S_k|)}, \hat{p}_{k-1}(\mathbf{S}) - s_{k-1}(\mathbf{S})\}, & \text{otherwise.} \end{cases} \tag{3.4}$$

  If a collision has occurred, the playout delay $\hat{p}_k(\mathbf{S})$ becomes $\hat{p}_{k-1}(\mathbf{S}) - s_{k-1}(\mathbf{S})$. It is $d_k^{(|S_k|)}$ otherwise.

- $\Delta(\mathbf{S}, k, j)$: sum of the increases in the playout delays incurred in the $(k+1)$-th to M-th talkspurts from collisions due to the introduction of j additional packets to be

played out in the k-th talkspurt given that the playout vector was originally $\mathbf{S}$. It is given by the following expression.

$$\Delta(\mathbf{S}, k, j) = \sum_{x=k+1}^{M} |S_x|(\hat{p}_x(\mathbf{S} \cup \mathbf{e}_k(|S_k|, j)) - \hat{p}_x(\mathbf{S})). \tag{3.5}$$

If the introduction of j additional packets to the k-th talkspurt incurs collisions, the playout delays of the $(k+1)$-th to the last talkspurts in a collision train become larger. The difference between the new larger playout delay and the original playout delay of the talkspurt is multiplied by the number of packets chosen from that talkspurt. These values are summed to obtain the total amount of additional playout delay.

In the second algorithm, not only $D(k,i)$ but also $C(k,i)$ is calculated at each step of computation. The equations for $D(k,i)$ and $C(k,i)$ are as follows:

$$D(k,i) = \begin{cases} 0, & \text{if } i = 0 \\ d_k^{(i)}, & \text{if } k = M, i \le n_M \\ \infty, & \text{if } k = M, i > n_M \\ \min\limits_{0 \le j \le i} \left\{ \frac{(i-j)D(k+1,i-j) + j d_k^{(j)} + \Delta(C(k+1,i-j),k,j)}{i} \right\}, & \text{otherwise.} \end{cases} \tag{3.6}$$

$$C(k,i) = \begin{cases} (\emptyset, \ldots, \emptyset), & \text{if } i = 0 \\ \mathbf{e}_k(0,i), & \text{if } k = M, i \le n_M \\ \mathbf{e}_k(0,n_M), & \text{if } k = M, i > n_M \\ C(k+1, i-j) \cup \mathbf{e}_k(0,j) & \\ \quad \text{where } j = \arg\min\limits_{0 \le j \le i} \left\{ \frac{(i-j)D(k+1,i-j) + j d_k^{(j)} + \Delta(C(k+1,i-j),k,j)}{i} \right\}, & \\ \quad \text{otherwise.} & \end{cases} \tag{3.7}$$

The only difference between (3.3) and (3.6) of the two bounding algorithms is the extra term $\Delta()$ in (3.6), which accounts for the extra delays incurred by collisions.

When j out of i packets are chosen from the k-th talkspurt for some $D(k,i)$, the indices of those packets are in $\mathbf{e}_k(0,j)$. The union of $\mathbf{e}_k(0,j)$ and $C(k+1,i-j)$ is assigned to $C(k,i)$.

The second algorithm accounts for the collisions in its calculation, but does not generate the exact optimum average playout delay. A close look at (3.6) reveals why. Consider a case, where for some $D(k,i)$ resulting from the algorithm, j packets are played out from the k-th talkspurt and cause a collision between the k-th and $(k+1)$-th talkspurt. The j packets from the k-th talkspurt chosen to be played out are those with the j smallest normalized playout delays. The algorithm thus does *not* consider the case where playing out some other j packets from the k-th talkspurt would not incur a collision (or as severe a collision) to the $(k+1)$-th talkspurt.

It is worth noting that it is not necessary to keep track of the identities of every packet to be played out in $C(k,i)$. It suffices to keep track of the number of packets in each talkspurt, the normalized delays, and the sender timestamps of the earliest and latest of the packets being played out. We found it simpler to present the algorithm as described above.

3.3.4 Computational complexity

The time complexity of the first algorithm is $O(M \cdot N^2)$ and the space complexity is $O(M \cdot N)$. The total number of packets in a trace, N, easily exceeds 30,000 for a trace lasting longer than 10 minutes. A closer look at (3.3) reveals that j can vary only from 0 to $\max_{1 \le k \le M}\{n_k\}$ in a real trace. If all talkspurts but one, in a trace, contain only one packet, then $\max_{1 \le k \le M}\{n_k\}$ is approximately N, and the time and space complexities are as indicated above. On the other hand, if all talkspurts in a trace have the same number

of packets, then $\max_{1 \leq k \leq M}\{n_k\}$ is N/M. This decreases the time complexity to $O(N^2)$. The space complexity also reduces to $O(N)$.

The second algorithm has higher time and space complexities; the time complexity is $O(M^2 \cdot N^2)$, and the space complexity also $O(M^2 \cdot N^2)$.

In some cases, it is possible to partition talkspurts into groups such that no two talkspurts from different groups collide. In this case, we can apply the second bounding algorithm to groups, and treat groups as talkspurts under the first bounding algorithm to obtain the upper bounds. This two-step computation can be used to reduce the algorithm's running time. If such a grouping eventually yields groups with only one talkspurt, the exact optimum average playout delay is calculated using the first bounding algorithm for the trace.

In this section, we have introduced two algorithms that provide the lower and upper bounds on the optimum average playout delay. The results from these two algorithms will be used later in Section 3.4.4 to compare the performance of various on-line adaptive algorithms.

3.4 On-line Adaptive Algorithm Based on Past History

In this section we present a new adaptive, on-line playout delay algorithm and discuss its motivation, design, and implementation. In Section 3.4.1 our observations regarding existing playout delay algorithms, and how they motivated the design of a new algorithm, are discussed. In Section 3.4.2 the new algorithm is presented in pseudo-code. In Section 3.4.3 we look into the implementation issues of the algorithm. In Section 3.4.4 we compare the new algorithm with others and with the bounds presented in Section 3.3.

3.4.1 Motivation

Let us first consider the playout delay adjustment algorithms Algorithm 1 and Algorithm 4 introduced in [48]. We relabel these as Algorithms 3.1 and 3.2, and present them

for completeness in this chapter. These two algorithms are based on stochastic gradient algorithms used in estimation and control theory [30], and operate by estimating two statistics characterizing the network delay incurred by audio packets: the delay itself, and a variational measure of the observed delays. Each of these estimates is recomputed each time a new packet arrives.

$$\alpha = 0.998002$$
$$\hat{u}_k^i = \alpha \hat{u}_k^{i-1} + (1-\alpha)\hat{d}_k^i$$
$$\hat{v}_k^i = \alpha \hat{v}_k^{i-1} + (1-\alpha)\left|\hat{u}_k^i - \hat{d}_k^i\right|$$

Figure 3.4. Pseudo code of Algorithm 3.1

Let $\hat{u}_k^i$ and $\hat{v}_k^i$ be estimates of the packet delay and variational measure of the i-th packet of the k-th talkspurt, respectively. At the beginning of a new talkspurt, the playout delay $\hat{p}_k$ is estimated as follows:

$$\hat{p}_k = \hat{u}_{k-1}^{n_{k-1}} + \beta \hat{v}_{k-1}^{n_{k-1}} \tag{3.8}$$

Here β is a variation coefficient and provides some slack in playout delay for arriving packets. The larger the coefficient, the more packets that are played out at the expense of longer playout delays. It is thus a parameter which can be used to control the delay/loss tradeoff incurred under Algorithms 3.1 and 3.2. It is used as such later in our simulations.

Algorithms 3.1 and 3.2 both use (3.8) to determine the playout delay for a talkspurt; they only differ in how they calculate $\hat{u}_k^i$ and $\hat{v}_k^i$. The algorithms themselves are given in Figures 3.4 and 3.5.

Algorithm 3.1 is a linear filter that is slow in catching up with a change in delays, but is good at maintaining a steady value, when $(1-\alpha)$, the gain of the estimator, is set

IF (`mode` == NORMAL)
  IF $(\left|\hat{d}_k^i - \hat{d}_k^{i-1}\right| > \left|\hat{v}_k^{i-1}\right| * 2 + 800)$
    `var` = 0;
    `mode` = SPIKE;
  ELSE
    `var` = `var`/2 + $\left|(\hat{d}_k^i - \hat{d}_k^{i-1})/8 + (\hat{d}_k^i - \hat{d}_k^{i-2})/8\right|$;
  IF (`var` ≤ 63)
    `mode` = NORMAL;
    $\hat{d}_k^{i-2} = \hat{d}_k^{i-1}$
    $\hat{d}_k^{i-1} = \hat{d}_k^i$
    return;
IF (`mode` == NORMAL)
  $\hat{u}_k^i = 0.125 * \hat{d}_k^i + 0.875 * \hat{u}_k^{i-1}$;
ELSE
  $\hat{u}_k^i = \hat{u}_k^{i-1} + \hat{d}_k^i - \hat{d}_k^{i-1}$;
$\hat{v}_k^i = 0.125 * \left|\hat{d}_k^i - \hat{u}_k^i\right| + 0.875 * \hat{v}_k^{i-1}$;
$\hat{d}_k^{i-2} = \hat{d}_k^{i-1}$
$\hat{d}_k^{i-1} = \hat{d}_k^i$
return;

Figure 3.5. Pseudo code of Algorithm 3.2

to be very low. We use a specific value of $\alpha = 0.998002$ chosen for NeVoT1.4 in our simulations. The choice of α is further discussed in Section 3.4.4

Algorithm 3.2 shown in Figure 3.5 has two modes of operation, depending on whether a spike has been detected. In normal mode, it operates like Algorithm 3.1 with a different gain, but in spike-detection mode, u_k^i is updated differently.

Figure 3.6 plots the playout delay of Algorithms 3.1 and 3.2 as well as that of our new algorithm, Algorithm 3.3 (to be described shortly) for a given delay trace. In Figure 3.6 the x-axis indicates the elapsed time since the beginning of a session. A diamond plots the end-to-end queueing delay of a packet received at that point in time. Solid rectangles delineate talkspurt boundaries. The playout delay computed by each of the

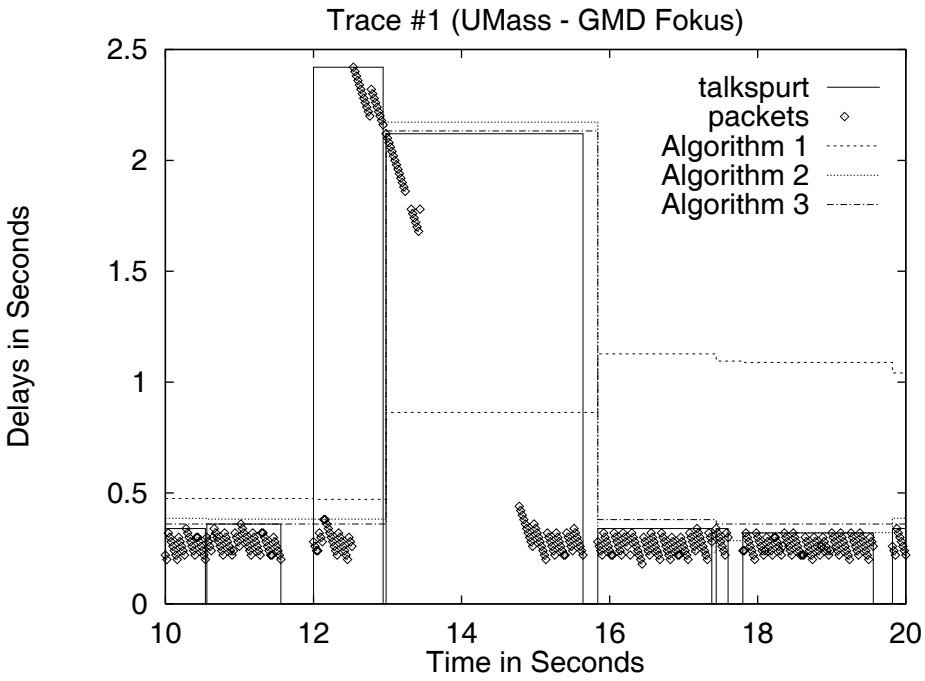


Figure 3.6. Delay estimates of three algorithms

three algorithms is indicated by a horizontal line that is as long (in the x-dimension) as the talkspurt.

From the figure, we see that Algorithm 3.1 computes the playout delay in such a way that the playout delay begins to increase only well after the delay spike has occurred. Note that under Algorithm 3.1, the packets at the beginning of the talkspurt beginning at approximately 13 seconds, are lost. This is because Algorithm 3.1 uses a delay estimator that reacts too slowly to delay spikes. Algorithm 3.2, on the other hand, computes a playout delay that reacts quickly to the delay spike. For example, the playout delay computed via Algorithm 3.2 for this high-delay talkspurt is such that no packets are lost. In the subsequent talkspurt, however, the playout delay is underestimated by Algorithm 3.2 and many packets are lost. The problem here is that Algorithm 3.2 attempts to track the network delays too closely and loses packets whenever its delay estimate is small, and the following talkspurt begins with packets that have suffered an even slightly higher

delay (i.e, the talkspurt beginning near time 16 and beyond). In the following section, we discuss the design of a new algorithm based on these observations.

3.4.2 Design

```
( 1) IF (mode == SPIKE)
( 2)   IF (d̂_i^k ≤ tail * old_d) /* the end of a spike */
( 3)     mode == NORMAL;
( 4) ELSE
( 5)   IF (d̂_k^i > head * p̂_k)    /* the beginning of a spike */
( 6)     mode = SPIKE;
( 7)     old_d = p̂_k;       /* save p̂_k to detect the end of a spike later */
( 8)   ELSE
( 9)     IF (delays[curr_pos] ≤ curr_delay)
(10)       count -= 1;
(11)     distr_fcn[delays[curr_pos]] -= 1;
(12)     delays[curr_pos] = d̂_k^i;
(13)     curr_pos = (curr_pos+1) % w;
(14)     distr_fcn[d̂_k^i] += 1;
(15)     IF (delays[curr_pos ] < curr_delay)
(16)       count += 1;
(17)     WHILE (count < w × q)
(18)       curr_delay += unit;
(19)       count += distr_fcn[curr_pos];
(20)     WHILE (count > w × q)
(21)       curr_delay -= unit;
(22)       count -= distr_fcn[curr_pos];
```

Figure 3.7. Pseudo code of Algorithm 3.3

Let us first informally describe Algorithm 3.3. The key idea behind our new algorithm is to collect statistics on packets that have already arrived and to use them to estimate the playout delay. Instead of using the linear filter mechanism, each packet's delay is logged and the distribution of packet delays is updated at every packet arrival. When a new talkspurt starts, our algorithm calculates a given percentile point q in the distribution

function of the packet delays for the last w packets, and uses it as the playout delay for the new talkspurt. As in Algorithm 3.2, Algorithm 3.3 detects spikes and behaves accordingly: once a spike is detected, it stops collecting packet delays and follows the spike until it detects the end of a spike. Upon detecting the end of a delay spike, it resumes its normal operation. As shown in Section 3.2, the delays of packets in a spike decrease in a linear fashion. Thus it is reasonable to use the delay of the first packet of a talkspurt as the playout delay for the talkspurt, if a new talkspurt begins during a spike. In the next paragraph we give a high-level description of the algorithm. The algorithm is also presented in C-language-like pseudo code in Figure 3.7, and is referred to during the design description below.

Algorithm 3.3 operates in two modes. For every packet that arrives at the receiver, the algorithm checks the current mode and, if necessary, switches its mode in lines 1 - 7 of Figure 3.7. Lines 9 - 22 update the delay distribution in normal mode. If a packet arrives with a delay that is larger than some multiple of the current playout delay, the algorithm switches to spike-detection mode. The end of a spike is detected in a similar way: if the delay of a newly arrived packet is less than some multiple of the playout delay before the current spike, the mode is set back to normal. Two parameters *head* and *tail* are used in lines 5 and 2 of Figure 3.7 in detecting the beginning and end of a spike. To determine the sensitivity of the algorithm to these parameters, we varied *head* from 2 to 20 and *tail* from 1 to 4 and evaluated Algorithm 3.3 using our delay traces. We found the algorithm to be relatively insensitive to values of *head* between 2 and 10, and of *tail* between 1 to 3. We chose 4 and 2 for *head* and *tail* in our simulations, as multiplication by powers of 2 can be implemented as shift operations.

Depending on the current mode, the playout delay for the next talkspurt is estimated differently in each mode as shown in Figure 3.8. In spike-detection mode, the delay of the first packet of a talkspurt becomes the estimated playout delay for the talkspurt.

```
(1)    IF (mode == SPIKE)
(2)       p̂_k = d̂_k^1;
(3)    ELSE (mode == NORMAL)
(4)       p̂_k = curr_delay;
```

Figure 3.8. Pseudo code of Playout Delay Estimation in Algorithm 3.3

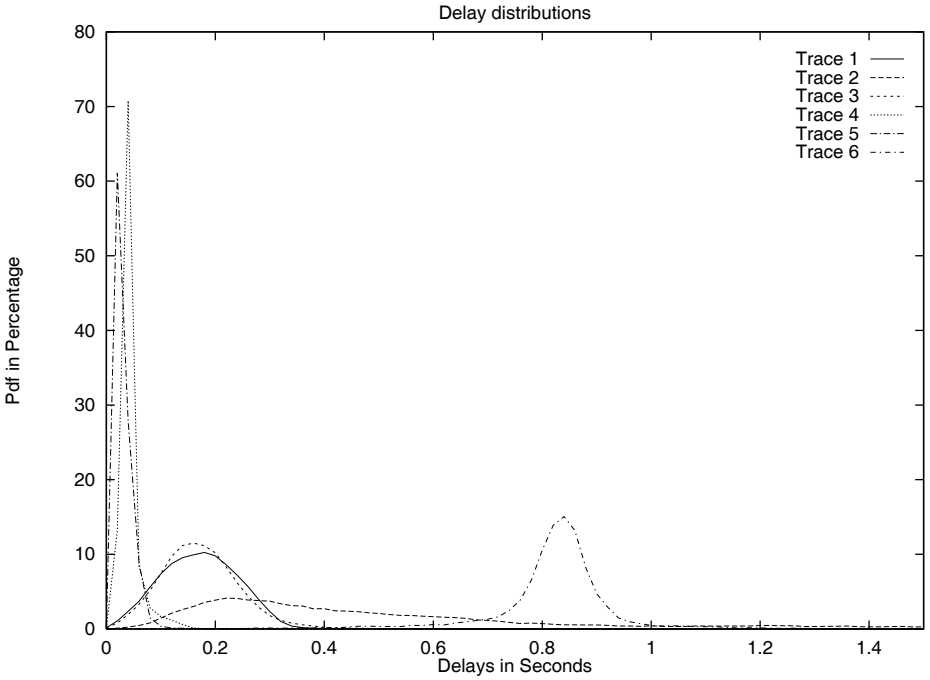


Figure 3.9. Delay Distribution of Traces

Otherwise, `curr_delay`, which is the given percentile point of delay based on previous statistics of packet delays, is used.

3.4.3 Implementation

All of the algorithms are executed every time a packet arrives at the receiver. Since the packetization interval of audio packets varies from 16ms to 32ms [24], the algorithms should be efficient enough to run 30 to 60 times a second, while leaving enough processing power for other activities. For Algorithm 3.3, lines 9-22 consist primarily of updating

counters and are integer operations. Theoretically its time complexity is proportional to the number of packet delays, w, that are stored. However, since most delay distributions are bell-shaped (Figure 3.9 plots these distributions from our six traces), it is expected to execute only a few loops, and thus in practice, we expect that the time complexity per packet for this algorithm to be constant.

```
(1)   WHILE (there is a packet in a trace file)
(2)     fetch a packet;
(3)     first checkpoint;
(4)     packet processing of the algorithm;
(5)     second checkpoint;
```

Figure 3.10. Playout Delay Estimation of Algorithm 3.3

To verify our hypothesis, a simple experiment was devised to measure the running time of our algorithm as well as that of Algorithm 3.1. Our measurements were performed as shown in Figure 3.10. The difference in time between the first and second checkpoints is accumulated over the entire trace of packets and the sum is divided by the number of packets in a trace. We ran the simulator on an SGI Indy R4600(134MHz) IRIX 5.2 and used the `gettimeofday()` system call for checkpointing. If the simulator is interrupted by other processes between two checkpoints, the time difference between two checkpoints includes not only the running time of our simulator, but also that of other processes. This impedes the exact measurement of algorithm running time, and affects both algorithms. To minimize the effect of this extraneous measurement, our experiments were run under light-loads, and checkpointed every 10000 packets. In addition we performed the same measurement for the case that the time to execute the algorithm 10,000 times was measured. All experiments on six traces gave the same order of magnitude value of less than 200μs for the per-packet processing of the algorithm – a relatively small amount of time given that packets are generated every 20ms. Algorithm 3.1 was also simulated

and run through the same set of experiments. We found no significant difference in the running times of Algorithms 3.1 and 3.3.

The space complexity of Algorithm 3.3 is linear in w, the window size of past history in number of packets, because delays of the previous w packets must be stored. The length of the history determines how sensitive the algorithm is in adapting to the change. If it is too short, the algorithm will have a myopic view of the past and is likely to produce a poor estimate of the playout delay. If it is too long, the algorithm will keep track of an unnecessarily large amount of past history. One potential weakness of our algorithm is that it may be slow to adapt to a steady increase or decrease in the "baseline" delays in the case of clock drifts. The decision on the length of history was made after evaluating the algorithm with different lengths of history. For lengths of history below 10,000 packets the performance degraded as the length became shorter. Above 10,000 packets, any performance enhancement was marginal. Thus in the results reported in the following section, the length of history w is set to 10,000 packets, corresponding to 200sec of time in the absence of silence periods. This results in a memory requirement of 40,000 bytes with 4-byte integers – a negligible amount of memory in today's workstations.

3.4.4 Comparison of Delay Adaptation Algorithms with Bounds

As mentioned in the introduction to this chapter, we compare the average playout delay vs. loss percentage produced by the different playout delay adaptation algorithms. To evaluate Algorithms 3.1 to 3.3, we designed and implemented a simulator that reads in the sender and receiver timestamps of each packet from a trace, determines if it has arrived before the playout time that is computed by a specific algorithm, and executes the algorithm. The simulator calculates the average playout delay and loss percentage for the given trace and outputs them. This allows us to compare the algorithms under the same conditions.

In Figure 3.11 the average playout delay is plotted as a function of the loss percentage for each algorithm. In the absence of any specific reference, all figures mentioned in this section are from Figure 3.11.

For Algorithms 3.1 and 3.2, instead of using the buffer size as the control parameter to be varied to achieve different loss percentages (as was done in [48]), here we varied β in (3.8). The range of values for β varies from 1 to 20 in our simulations. In Figure 3.11 a diamond for Algorithm 3.1 and a plus for Algorithm 3.2 are used explicitly to mark the β value of 4, which was used in [48].

For Algorithm 3.1, we ran a set of simulations to determine the sensitivity of the algorithm to the value of α. For $0.90 \leq \alpha \leq 0.999$, the algorithm's performance did not change dramatically. For $\alpha < 0.90$, however, the performance degraded. The specific value of 0.998002 was chosen for NeVoT1.4, and we used this value in our simulations.

Since Algorithm 3.3 does not use the mean or variational measure, it is parameterized by the percentile point of 50% to 100% on the figures. On the graphs of Algorithm 3.3, 99% and 97% points are marked with a square and a cross: the former represents the 99% point and the latter 97%.

As the algorithms for bounding performance use normalized delays in their calculation of average playout delay, the average playout delay of Algorithms 3.1 to 3.3 is also normalized by subtracting $\hat{d}$ from it; these normalized average playout delays are plotted in the figures. The lower and upper bounds of the optimum average playout delay always lie below the graphs of Algorithms 3.1 to 3.3, since they are the theoretically optimum (minimum) bounds of the average playout delay for any given loss percentage. Algorithms 3.1 to 3.3 yield a single playout delay vs. loss percentage for each value of the control parameter (variation coefficient or percentile point). The graphs of Algorithms 3.1 to 3.3 are generated by varying these control parameters and connecting these pairs.

Our figures illustrate several interesting points. First, note that the upper and lower bounds on the optimum playout delay versus loss tradeoff are quite close, as long as the

loss percentage is 1% or more. (3.3) and (3.6) thus provide very tight lower and upper bounds on the optimum average playout delay for loss percentages in the range of interest.

Trace 3.2 in Figure 3.11(b) was collected between two multicast sites on the MBone during busy hours. The network loss percentage between the sender and the receiver is a horrendously high 58%. It also has a long blackout period of almost 2min, when no packets arrived at the sender. This blackout phenomenon on the MBone is reported in [59].

Trace 3.3 in Figure 3.11(c) was also collected during busy hours, but using unicast connections over the Internet. It suffered a network loss percentage of 17%, which is far lower than that of Trace 3.2, but still far from the desirable range of 2 to 5%. For Trace 3.3, all three algorithms show similar average playout delays near 350msec on the y-axis for marked points.

In (d), (e), and (f) of Figure 3.11, Algorithm 3.3 performs best in all points, nearly touching the optimum delay in Figure 3.11(e). All four marked points on the graphs are close in their y-coordinates, but their x-coordinates are somewhat dispersed. The marked points of Algorithm 3.2 are consistently positioned to the right of other marked points on the x-axis, which means it drops more packets due to late arrival for a given playout delay. This verifies our previous observation that Algorithm 3.2 underestimates the playout delay after spikes.

3.5 Conclusion

In this chapter we have focused on the tradeoff between packet playout delay and packet playout loss. We presented algorithms for computing upper and lower bounds on the optimum (minimum) average playout delay for a given number of packet losses (due to late arrivals) at the receiver for a given trace of packet delays. These bounds were shown to be tight for a range of loss and delay values of interest, and are important as they provide a bound on the achievable performance of *any* adaptive playout delay adjustment

algorithm. We also presented a new adaptive delay adjustment algorithm that tracks the network delay of recently received packets and efficiently maintains delay percentile information. Our new algorithm was shown to outperform existing delay adjustment algorithms over a number of measured audio delay traces and performs close to the theoretical optimum over a range of parameter values of interest.

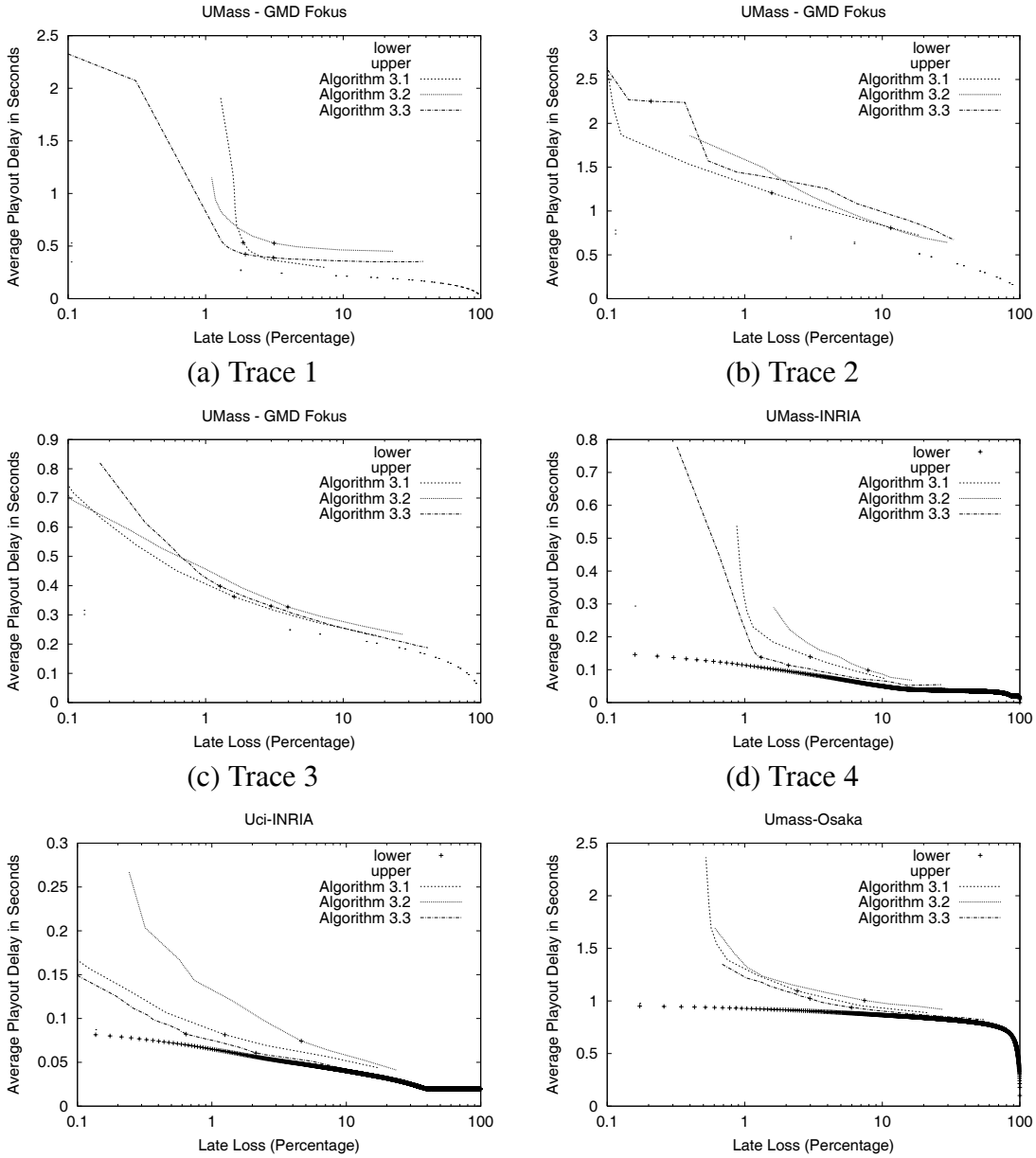


Figure 3.11. Comparison of three playout delay adaptive algorithms

CHAPTER 4

CORRELATION BETWEEN DELAY AND LOSS

4.1 Introduction and Motivation

Traditional Internet applications such as *telnet* and *ftp* use TCP as their transport layer protocol, and depend on TCP congestion control when there is congestion in the network. In TCP, a sender increases its transmission rate additively until it experiences a packet loss, which is taken as an indication of congestion. The sender then decreases its transmission rate multiplicatively, thus reacting quickly to the inferred congestion. As a result of this behavior, the transmission rate of the TCP sender is determined by the level of congestion in the network.

Continuous media applications, on the other hand, usually react to network congestion with less flexibility (if at all), due to their more stringent timing constraints. Continuous media applications can be not only loss-adaptive, but delay-adaptive as well. That is, they can have either a fixed playout time, or adapt to changes in packet delay and set the playout time accordingly as discussed in Chapter 3 and in [48, 39]. Such adaptive applications keep track of packet delays, and reflect any change in packet delays in their calculation of "playout time."

Given the above discussion, it is clear that packet loss and delay have tremendous impact on continuous media applications. Both loss and delay result from buffering within the network. As packets traverse the network, they are queued in buffers (thus adding to their end-end delay) and from time to time are dropped due to buffer overflow. Consider then a continuous media packet stream at a buffer that is filling up fast with packets from other traffic sources as well. The packets from the continuous media application continue

to be queued up in the growing packet queue together with the packets from other sources. Continuous media packets arriving at the receiver experience progressively higher end-end delays than earlier packets. When the buffer reaches its capacity, packet losses begin to occur. The receiver of the continuous media application thus sees increased delay, and eventually losses.

Consider next a scenario in which packets from a continuous media application arrive at a buffer that is already full. In this case, they are dropped. As other sources (e.g, TCP connections) detect congestion and decrease their transmission rate, the queue length at the buffer will decrease, and packets from the continuous media application will start to be queued, rather than dropped. In this scenario, the receiver sees losses followed by high, but possibly decreasing, packet delays.

The two examples above are plausible scenarios. In the first example, the receiver could have taken the increased delay as an indication of likely future packet loss; in the second scenario, the decreasing delay following loss indicates a future uncongested period of time. In both cases, the application could adapt its behavior accordingly. But do we expect such scenarios to occur (or be detectable) often in practice? Note that there are a number of implicit assumptions in the discussion above, including a single bottleneck link and presumed behavior by other competing applications. If delay is indeed affected by loss, or vice versa, how temporally close are they? What is causing such correlated behavior in the network? How can we exploit such information at the end-system? These are some of the issues that we touch on in the following sections.

In this chapter we report the correlation between delay and loss observed by a continuous media traffic source on the Internet. Our goal in undertaking this study is to determine the extent to which one performance measure could be used as a predictor of the future behavior of the other (e.g., whether observed increasing delay is a good predictor of future loss) so that an adaptive continuous media application might take *anticipatory* action based on observed performance. In this study, we ran numerous hour-long experiments in

which packets were periodically sent from a source to a destination. We measured the per-packet delay and packet loss and then analyzed our measurements off-line. Our results provide a quantitative study of the extent to which such correlation exists. Interestingly, we observe periodic phenomena in the correlation that we had initially not expected. We discuss our results, speculate on the observed behaviors, and discuss their implications for adaptive continuous media applications.

The rest of the chapter is organized as follows. In Section 4.2, we describe the tool used to collect measurements, and discuss the technical issues surrounding the measurement process and the evaluation of the empirical results. We introduce a quantitative measure for correlation between delay and loss in Section 4.3, and apply it to analyze the measurements in Section 4.4. Section 4.5 concludes this chapter with a summary.

4.2 Measurement

To collect end-end delay and loss data for a continuous media source, we built a tool [38] that generated packets with RTP headers [53, 52] at a fixed, periodic rate. The default RTP header has a fixed length of 12 bytes, and includes the version number, sequence number, media-specific timestamp, and source identifier. The sequence number is increased by one per packet, while the increment of a timestamp is dependent on the payload type. We maintained a log at both the source and the destination of real timestamps from the system clock along with the RTP sequence number and media-dependent timestamps; these are used to calculate end-end delay. To minimize the load inside the network, each packet had only the 12-byte RTP header, and did not carry any payload data.

To generate packets at a periodic interval of less than 100ms, we use the workstation's audio device for timing. Most workstations provide a timer in their operating systems, but its granularity is usually in hundreds of milliseconds, and not fine enough for our measurement purpose. Audio devices on most workstations interrupt the operating system

Trace	Receiver	Start time at Sender	Duration	Loss Prob.
4.1	UV	11:12pm, Tu, 2/11/97	1800sec	0.027
4.2	WashU	7:31pm, Mo, 3/24/97	2348sec	0.010
4.3	GaTech	7:10pm, Tu, 3/26/97	1864sec	0.041
4.4	UWash	7:58pm, Fr, 3/28/97	2343sec	0.013
4.5	UTexas	7:23pm, Tu, 2/18/97	1808sec	0.091
4.6	SICS	9:53pm, Mo, 3/24/97	2228sec	0.23

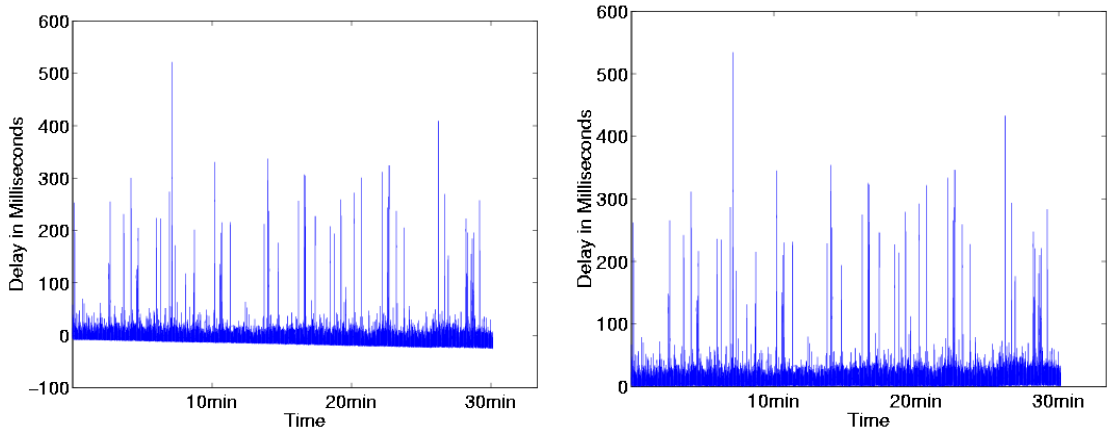
Table 4.1. Traces Used in Chapter 4

when the audio buffer is full, with the interval between two consecutive interrupts being in the tens of milliseconds. The application programming interface to the audio device on most workstations allows the fine-tuning of the interrupt interval by changing the audio buffer size. We used the audio device interrupt to generate packets periodically at 20 ms intervals.

Because timing and delay were so critical to our measurements, it was important for us to verify that packets were indeed being sent at 20 ms intervals, and not being inordinately delayed in the operating system or network interface. We thus first ran a 30-minute-long experiment between two SGI Indy machines on the same branch of a LAN. The results showed that there was no loss between the two machines, and that sender packet inter-departure times were kept almost constant at 20ms.

We collected six sets of measurements between February and March of 1997. Table 4.1 describes when and where the measurements took place. All the traces originated from two machines at the University of Massachusetts. Traces 4.1 - 4.3 were to sites on the east coast, Traces 4.4 and 4.5 to sites on the west coast, and Trace 4.6 to an European site.

As discussed in Chapter 2, impairments in measured delays due to unsynchronized clocks should be addressed before the measure delays could be analyzed. The first impairment we consider is the clock skew. If the clocks at the source and destination are not synchronized, and have a frequency drift, the delay measurements show a gradual



(a) Before the skew is removed. (b) After the skew is removed.

Figure 4.1. Scatter-plots of delay from Trace 4.1 before and after the skew removal

increase or decrease over time depending on which clock is faster. Figure 4.1(a) plots the packet timestamp at the sender versus per-packet delay from Trace 2.1, showing a gradual decline of delay over time from Trace 4.1. In this case, the sender clock is faster than the receiver clock. We used the Linear Programming algorithm presented in Chapter 2 to remove the skew in delay measurements. Figure 4.1(b) presents the delay after the skew is removed.

Another issue to be considered is routing. If packets are routed through different paths to the destination, the resulting delays and losses at the destination come from different queues in the network. This would complicate our analysis of correlation between losses and delays. A recent study by Paxson [45] suggests that route changes can be detected by out-of-order deliveries of packets (although out-of-order packets do not necessarily imply a route change). In his traces, less that 1% of packets were delivered out of order. In all our traces, out-of-order packets constitute less than 0.1% of the total number of packets. We do not consider out-of-order packets in our analysis below.

4.3 Correlation between Delay and Loss

Let us now introduce the notation to be used in our data analysis. We define the following:

- N: length of a trace.
- i: packet sequence number, $1 \leq i \leq N$
- l_i: indicates whether a packet is lost or delivered, $i = 1, 2, \ldots, N$;

$$l_i = \begin{cases} 0, & \text{if the packet is lost.} \\ 1, & \text{if the packet is delivered to the destination.} \end{cases}$$

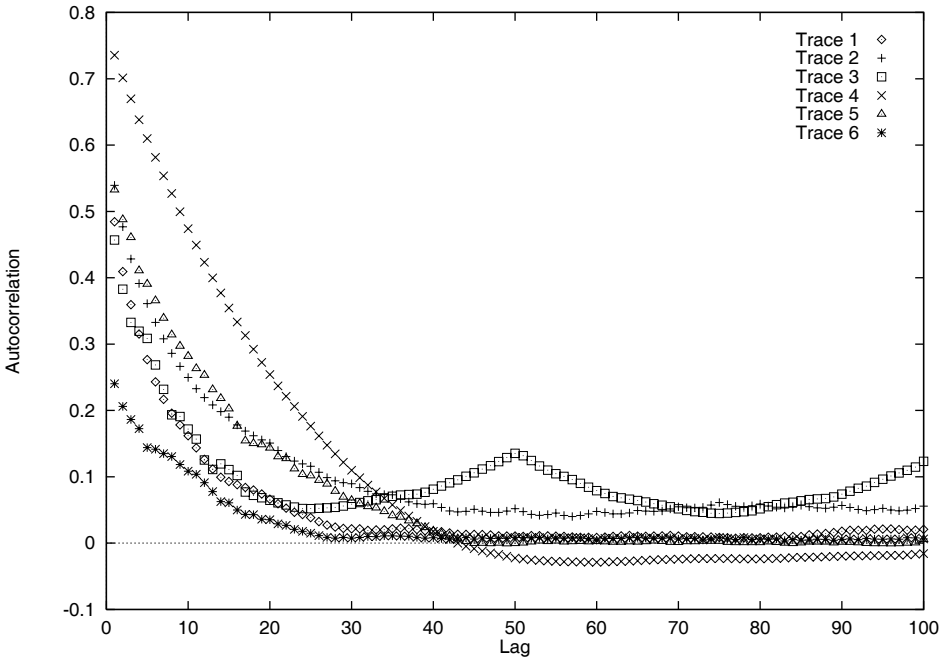
- d_i: delay of the i-th packet after the skew is removed. $d_i \geq 0$. If $l_i = 0$, i.e., if the i-th packet is lost, arbitrarily take d_i to be 0.
- M: number of packets that are delivered to the destination. Note that $M \leq N$ and $M + \sum_{i=1}^{N}(1 - l_i) = N$.
- $\bar{d} \equiv \sum_{i=1}^{N} d_i/M$: sample mean delay. We call it *unconditional mean delay*.
- $\bar{l} \equiv \sum_{i=1}^{N}(1 - l_i)/N$: loss rate.

Before proceeding to quantify the correlation between delay and loss, we first look at the autocorrelation of delay. The autocorrelation of delay at lag j is:

$$r_j = \frac{\sum_{i=1}^{N}(d_i - \bar{d})(d_{i+j} - \bar{d})}{\sum_{i=1}^{N}(d_i - \bar{d})^2}$$

As shown in Figure 4.2, the autocorrelation of delay eventually dies out over time as the lag increases. Trace 4.3, however, stands out from others. Since the autocorrelation of Trace 4.3 shows that delays with a lag of approximately 50 are correlated, Trace 4.3

Figure 4.2. Autocorrelation of Delay



shows periodicity in other measures of performance, as we discuss it in the next section. The delay autocorrelation delay is another performance measure with which to assess the temporal correlation of data.

In order to quantify the correlation between delay and loss, we consider the sample mean delay conditioned on loss, as discussed below. We cannot use the correlation between delay and loss, because they represent different aspects of a phenomenon, and their values are not compatible. Furthermore, since a packet either arrives at a receiver with a finite delay, or does not arrive at all, we cannot calculate the sample mean delay of packets conditioned on their own loss.

We introduce a *lag* in calculating the sample mean delay conditioned on loss. Specifically, the sample mean delay, conditioned on a loss occurring at a time lag j packets in the past, is the sample mean of delay of all packets in the trace that have a loss j packets before them in the trace. That is,

$$\hat{E}[d_i|l_{i-j}=0] = \sum_{k\in P} d_k/|P|, \quad \text{where} \quad P=\{k : l_{k-j}=0 \text{ and } l_k=1\} \tag{4.1}$$

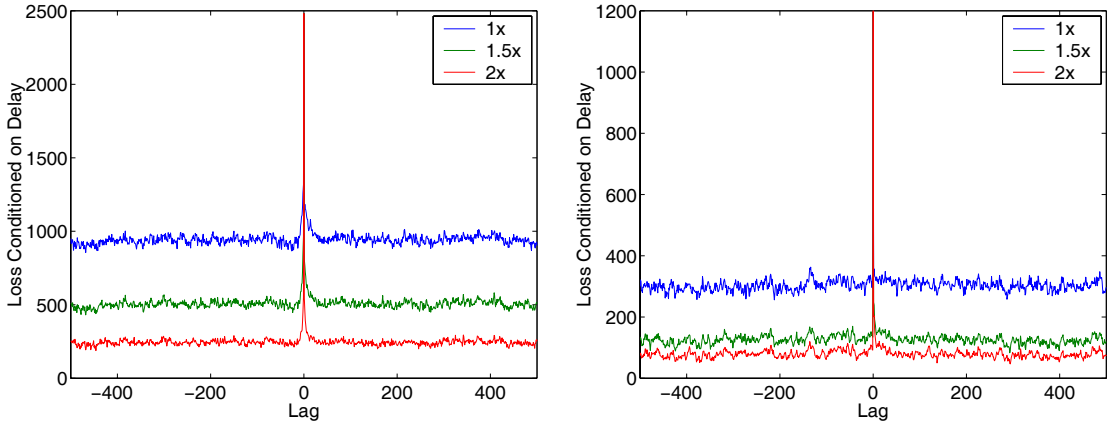
$$\bar{E}[d_i|l_{i-j}=0] = \hat{E}[d_i|l_{i-j}=0]/\bar{d} \tag{4.2}$$

We will refer to the above quantity of (4.1) as the *sample mean delay conditioned on loss,* and (4.2) as the *normalized sample mean delay conditioned on loss.* If the sample mean delay conditioned on loss at a positive lag of j is higher than the sample mean delay (i.e., the delay averaged over all received packets), it means that packets that arrive j packets after a loss have a higher average delay than a randomly chosen packet. That is, a loss occurring j packets in the past can be taken as a precursor to a higher delay later. A literal interpretation of a higher conditional average delay at a negative lag is less intuitive. In such a case, a loss can be thought of as an indicator of higher delay in the past. We revisit this issue later in this section.

The sample mean delay conditioned on loss at a positive lag allows us to look at delays of future packets following a loss event. To look at future loss conditioned on current packet delay, we can count the number of packets lost at a specific lag from a packet experiencing a given delay. This is expressed by the following:

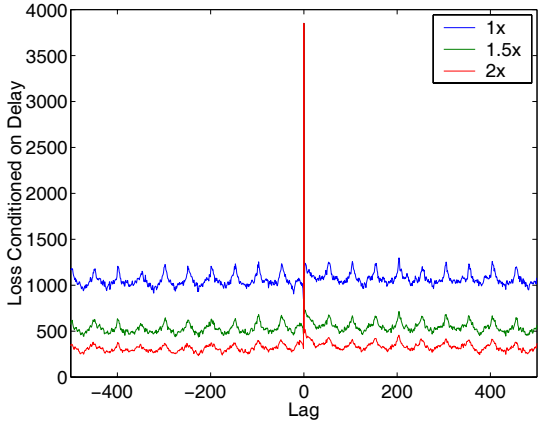
$$C_j = |\{i : l_i = 0 \text{ and } d_{i-j} > T\}| \tag{4.3}$$

Here, we are counting the losses conditioned on a packet at time lag j having a delay above a given threshold, T. By varying T within the range of $\bar{d}$ and $2\bar{d}$, we can observe how sensitive the losses are to the delays. Figure 4.3 plots the loss conditioned on delay as in (4.3). The threshold values used in calculation are 1, 1.5, and 2 times the unconditional mean delay. Figure 4.3 indicates that the loss conditioned on delay as a function of lag, computed via (4.3), is not particularly sensitive to the threshold value. Trace 4.3 stands out from other traces due to its definite periodicity.

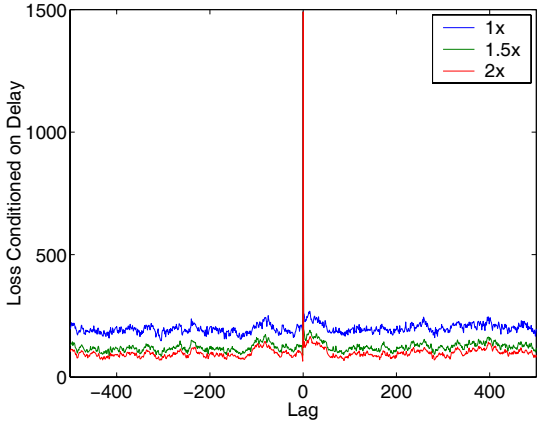


(a) Trace 4.1

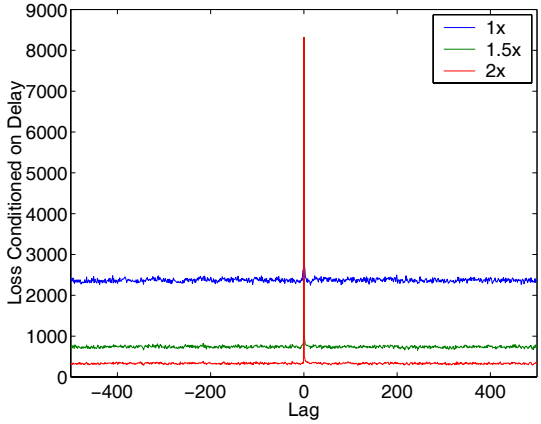
(b) Trace 4.2



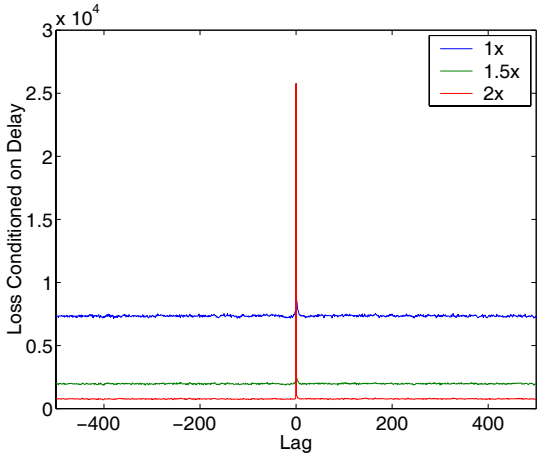
(c) Trace 4.3



(d) Trace 4.4



(e) Trace 4.5



(f) Trace 4.6

Figure 4.3. Loss Conditioned on Delay

4.4 Analysis of Measurements

Figure 4.4 plots the normalized sample mean delay of all six traces listed in Table 4.1, using (4.1). The range of the lag is between -500 and 500, which corresponds to 10 seconds in both positive and negative directions. All traces show higher loss-conditioned average delay near lag 0. This can be explained by an FCFS buffering process where packets that enter the queue just before an overflow experience a large queueing delay (since the queue is nearly full). Similarly, those packets successfully entering the buffer soon after an overflow event will also see a large queueing delay, since the queue is still nearly full

Another general observation of the graph is that the loss-conditioned average delay shows periodic behavior, dropping *below* the value of the unconditional mean delay at a given time lag and then rising again above that value for a larger lag. Traces 4.2 and 4.4 have a trace loss probability of less than 0.03. Their graphs show a well-pronounced rise between lags of 50 and 100, a few more after lag 100 and when lags are negative. As the loss probability increases from those traces to Traces 4.3, 4.5, and 4.6 in an increasing order, the correlation graph flattens out.

These observations raise several interesting questions. First, what causes the oscillations in Traces 4.2 and 4.4? Why is it located between lag of 50 and 100. Why does it disappear as the loss probability increases?

We can suggest a few possible answers these questions if we first make a few assumptions. First, assume that there is just one congested link along the path from a source to a destination, and that packets are delayed or dropped mostly at that link. This allows us to focus on single-hop behavior, and the impact of other traffic sources on the observed performance of our probe traffic. Second, let us assume that a significant number of the sources that are competing in this bottleneck queue use some form of TCP congestion control. All applications on the Internet use either TCP or UDP as their transport protocol. Theoretically applications that use UDP can consume unlimited bandwidth, and their

behavior is unpredictable. However, most continuous media applications using UDP try to transmit at a fixed rate that is typically determined by the bottleneck bandwidth.

These two assumptions allow us to focus on the impact of other traffic sources on our probe traffic at a single congested link. Returning back to our earlier observations regarding oscillations, the oscillations may be explained by some form of synchronization among traffic sources. The synchronized sources may cause the oscillations in the graphs of the sample mean delay conditioned on loss with the loss probability less than 0.05. When congestion is detected in the network, all TCP sources whose packets are routed through the congested link initiated congestion avoidance when a loss is detected and cut down the size of their congestion windows. Depending on the round-trip time from those sources to their destinations, they will then start to increase their window size, until congestion again occurs at this link. In this scenario, *the loss events in the congested link serve to synchronize the increase and decrease in traffic* offered by the TCP sources passing through this link. The continuous media source, which can be thought of as a periodic "probe" that samples the congested queue state, thus observes the periodic rise and fall of traffic (and the concomitant rise and fall in delay) that occurs as a result of loss-induced synchronization among TCP connections passing through this link.

Earlier simulation work reported that TCP sources can behave in such a synchronized fashion [55]. A recent study observed in simulations that a UDP source sees more losses if the interfering traffic from TCP sources is synchronized [51]. We note that the impact of synchronized TCP sources on the correlation of delay and loss requires further study in simulations.

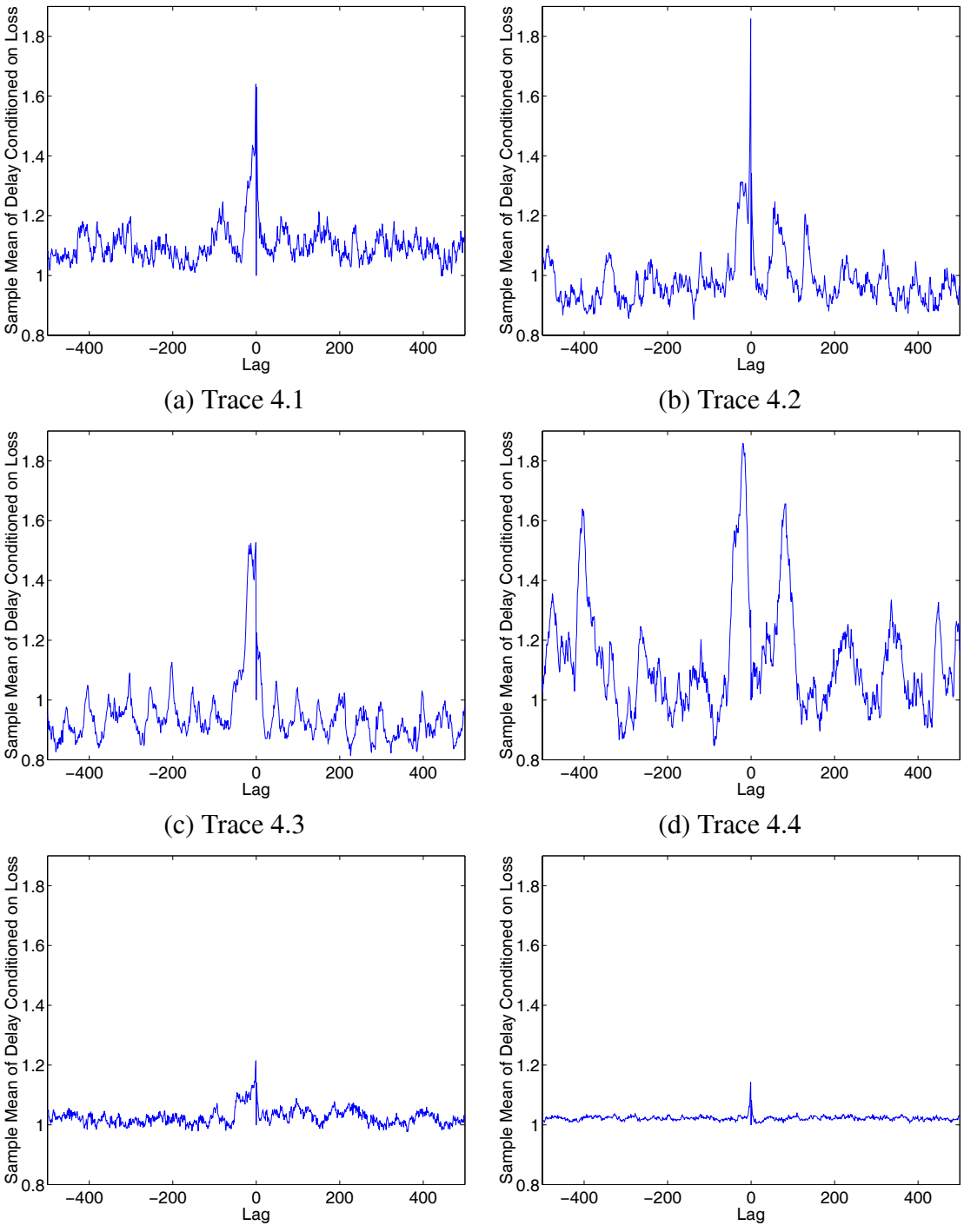
Trace 4.6 stands out from other traces because it shows very little fluctuation in its loss-conditioned average delay over the entire range of lags. This can be explained by noting that the trace was collected over a very lossy link between UMass and Sweden. The loss probability is over 0.20 – more than an order of magnitude larger than that of any other traces. The high loss probability is an indication of heavy congestion on the

bottleneck link. We conjecture that as more losses are detected by the sources, the sources adjust their congestion windows more often, and their self-synchronization becomes less prominent.

4.5 Conclusions

The analysis of the delay and loss measurements using the sample mean of delay conditioned on loss shows that it is likely that the packet delay would increase in a near future if a packet loss is detected. The loss conditioned on delay, on the other hand, is shown to be not sensitive to the threshold values of delay. Also we observe that some traces exhibit an oscillatory behavior when viewed using the sample mean delay conditioned on loss.

We believe that this knowledge about correlation between delay and loss can be utilized in design and improvement of application-level congestion control and repair mechanisms at end hosts.



(a) Trace 4.1

(b) Trace 4.2

(c) Trace 4.3

(d) Trace 4.4

(e) Trace 4.5

(f) Trace 4.6

Figure 4.4. Normalized sample mean delay Conditioned on Loss

CHAPTER 5

INFERENCE OF INTERNAL LOSS RATES IN THE MBONE

5.1 Introduction

As the Internet grows in size and diversity, its internal performance becomes harder to measure. Any one organization has administrative access to only a small fraction of the network's internal nodes, while commercial factors often prevent organizations from sharing internal performance data. As discussed in Section 1, end users have limited access to the information about the internal dynamics of the network. End-to-end measurements using unicast traffic do not rely on administrative privileges, but it is difficult to infer link-level performance from them and they require large amounts of traffic to cover multiple paths. There is a need for practical and efficient procedures that can take an internal snapshot of a significant portion of the network.

Cáceres et al. have developed a measurement technique that addresses these problems. *Multicast Inference of Network Characteristics* (MINC) [36] uses end-to-end multicast traffic as measurement probes. It exploits the inherent correlation in performance observed by multicast receivers to infer the loss rate and other attributes of paths between branch points in a multicast routing tree. These measurements do not rely on administrative access to internal nodes since they are done between end hosts. In addition, they scale to large networks because of the bandwidth efficiency of multicast traffic.

The intuition behind packet loss inference is that the event that a packet has reached a given internal node in the tree can be inferred from the packet's arrival at one or more receivers descended from that node. Conditioning on this event, we can determine the

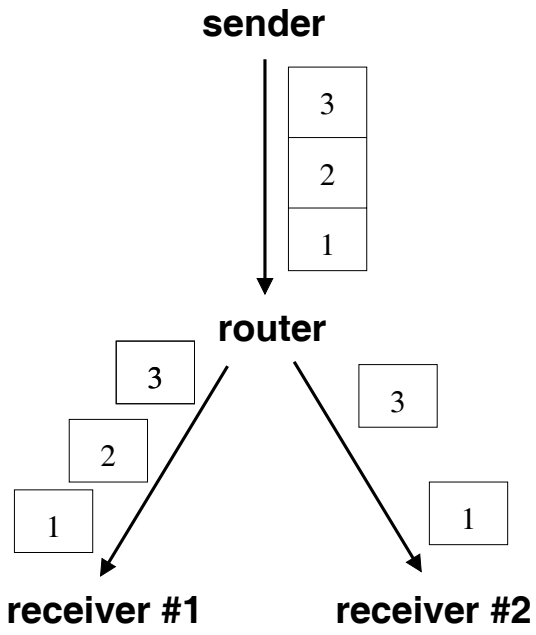


Figure 5.1. Example of a multicast tree and a multicast packet loss

probability of successful transmission to and beyond the given node. Consider, for example, a simple multicast tree in Figure 5.1 with a root node (the source), two leaf nodes (the receivers #1 and #2), a link from the source to a branch point (the shared link), and a link from the branch point to each of the receivers (the links to the receivers #1 and #2). The source sends a stream of multicast packets sequenced from 1 to 3 through the tree to the two receivers. As illustrated in the figure, if the second packet reaches only the receiver #1, we can infer that the packet reached the branch point. Thus the ratio of the number of packets that reached both receivers to the number that reached only the right receiver gives an estimate of the probability of successful transmission on the left link. The probability of successful transmission on the other links can be found by similar reasoning.

It is not immediately clear whether this technique applies to more than just binary trees or whether it enjoys desirable statistical properties. Cáceres et al. [5] extended this technique to general trees and showed that the estimate is consistent, that is, it converges

to the true loss rates as the number of probes grows. More specifically, a *Maximum Likelihood Estimator* (MLE) for internal loss rates in a general tree was developed assuming independent losses across links and across probes. They derived the MLE's rate of convergence and established its robustness with respect to certain violations of the independence assumption. They also validated these analytical results using the `ns` simulator [42]. We give a brief account of these results in Section 5.2.

In more recent work [6], Cáceres et al. explored the accuracy of the MLE estimates for packet loss under a variety of network conditions. Again using `ns` simulations, they evaluated the error between inferred and actual loss rates, varying the network topology, propagation delay, packet drop policy, background traffic mix, and probe traffic type. They report that, in all cases, MINC accurately inferred the per-link loss rates of multicast probe traffic.

In this chapter, we further validate MINC through experiments under real network conditions, and study MINC given a limited number of observations. We used a collection of end hosts connected to the MBone, the multicast-capable subset of the Internet [31]. We chose one host as the source of multicast probes and used the rest as receivers. We then made two types of measurements simultaneously: end-to-end loss measurements between the source and each receiver, and direct loss measurements at every internal node of the multicast tree. Finally, we ran our inference algorithm on the results of the end-to-end measurements, and compared the inferred loss rates to the directly measured loss rates. Across all our experiments, the inferred values closely matched the directly measured values. The differences between the two were usually well below 1% and never above 3%, while loss rates varied between 0% and 35%. Furthermore, the inference algorithm converged well within 2-minute, 1200-probe measurement intervals.

We also study the performance of MINC estimates in comparison to the directly measured link loss rates given a limited number of packets, and under a wide range of values of loss rates and different tree topologies. The simulation results help us understand how

rapidly the differences between MINC estimates and the directly measured link loss rates decrease as the loss rate, tree depth, and tree branching factor change.

The rest of this chapter is organized as follows: Section 5.2 describes the MINC methodology; Section 5.3 presents the design and results of our MBone experiments; Section 5.4 presents the simulation results of the MINC performance given a limited number of packets and a wide range of parameters; Section 5.5 surveys related work; and Section 5.6 offers some conclusions.

5.2 MINC Methodology

In this section we give a summary of the MINC methodology.

5.2.1 Statistical inference

MINC works on *logical* multicast trees. A logical tree is one where all nodes, except the root and the leaves, have at least two children. A physical tree can be converted into a logical tree by deleting all nodes, other than the root, that have only one child and then collapsing the links accordingly. A link in a logical tree may thus represent multiple physical links. This conversion is necessary because inference based on correlation among receivers cannot distinguish between two physical links unless these links lead to two different receivers. Henceforth when we speak of trees we will be speaking of logical trees.

5.2.1.1 Inference algorithm

The model for loss on a multicast tree assumes that packet loss is independent across different links of the tree, and independent between different probes. With these assumptions, the loss model is specified by associating a probability α_k with each node k in the tree. α_k is the probability that a packet is transmitted successfully across the link terminating at node k, given that it reaches the parent node $p(k)$ of k.

When a probe is transmitted from the source, we can record the outcome as the set of receivers the probe reached. The loss inference algorithm is based on probabilistic analysis that allows us to express the α_k directly in terms of the *expected* frequencies of such outcomes. More precisely, for each node k let γ_k denote the probability of the outcome that a given packet reaches at least one receiver that has k as an ancestor in the tree. Let A_k denote the probability that a given packet reaches the node k, i.e., $A_k = \alpha_k \alpha_{k_1} \alpha_{k_2} \ldots \alpha_{k_m}$ where $k_1, k_2, \ldots, k_m$ is the chain of m adjacent nodes leading back from node k to the root of the tree. Then it can be shown that A_k satisfies

$$(1 - \gamma_k / A_k) = \prod_{j \in c(k)} (1 - \gamma_j / A_k) \tag{5.1}$$

where the product is taken over all nodes j in $c(k)$, the set of children of the node k. It was shown in [5] that under generic conditions the A_k can be recovered uniquely through (5.1) if the γ are known. The α_k can in turn be recovered since $\alpha_k = A_k / A_{p(k)}$. Generally, finding A_k requires numerical root-finding for (5.1). In the special case of a node k with two offspring j and j', (5.1) can be solved explicitly:

$$A_k = \frac{\gamma_j \gamma_{j'}}{\gamma_j + \gamma_{j'} - \gamma_k} \tag{5.2}$$

Suppose that in place of the γ_k in (5.1), we use the *actual* frequencies $\widehat{\gamma}_k$ with which n probes reach at least one receiver with ancestor k. We denote the corresponding solutions to (5.1) by $\widehat{A}_k$ and estimate the link probabilities by $\widehat{\alpha}_k = \widehat{A}_k / \widehat{A}_{p(k)}$. The calculation of the $\widehat{\gamma}_k$ is achieved though a simple recursion as follows. Define new variables $Y_k(i)$ as function of the measured outcomes of n probes by

$$Y_k(i) = \begin{cases} 1 & \text{if probe } i \text{ reaches node } k \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

if k is a leaf node, and

$$Y_k(i) = \max_{j \in c(k)} Y_j(i) \tag{5.4}$$

otherwise. Then

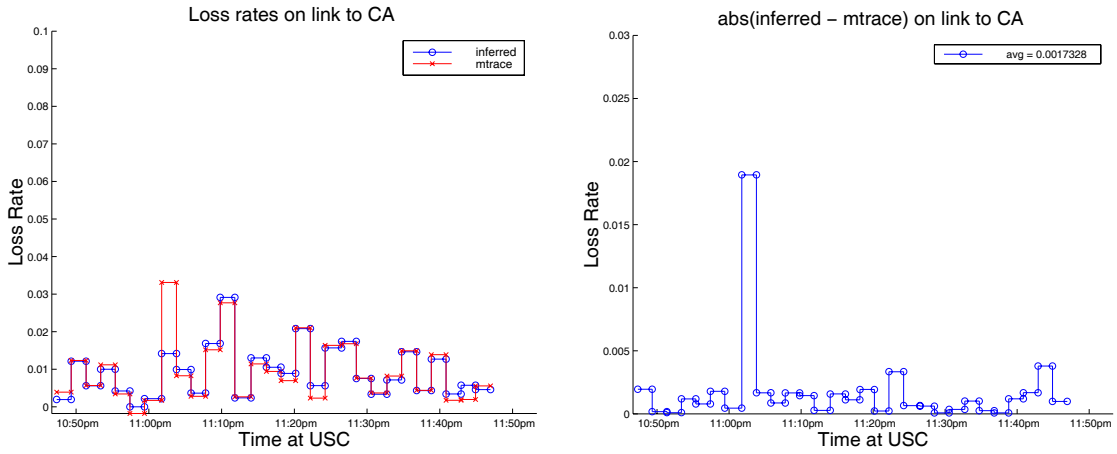
$$\widehat{\gamma}_k = \frac{1}{n} \sum_{i=1}^{n} Y_k(i) \tag{5.5}$$

It is shown in [5] that the estimator $\widehat{\alpha}_k$ enjoys two useful properties: (i) *consistency*: $\widehat{\alpha}_k$ converges to the true value α_k almost surely as the number of probes n grows to infinity, and (ii) *asymptotic normality*: the distribution of the normalized difference $\sqrt{n}(\widehat{\alpha}_k - \alpha_k)$ converges to a normal distribution as n grows to infinity. Also investigated in [5] are the effects of correlations that violate the independent loss assumptions. Consistency is preserved under a large class of temporal correlations, although convergence of the estimates with n can be slower. Spatial correlations perturb the estimate continuously, in that small correlations lead to small inconsistencies. When losses on sibling links are correlated the perturbation is a second-order effect, in that the degree of inconsistency depends not on the size of the correlations, but on the degree to which they change across the tree.

Earlier papers on MINC [5, 6] contain a detailed description and analysis of the above inference algorithm, including rules to handle special cases of the data in which the generic conditions required for the existence of solutions to (5.1) fail. In the interests of brevity, we omit these details from this chapter.

5.3 MBone Validation

During each of our MBone experiments, we had a source send a stream of sequenced packets to a collection of receivers while we made two types of measurement at each receiver. At the source, we used our `mgen` [38] traffic generation tool to send one 40-byte packet every 100 milliseconds to a specific multicast group. The resulting traffic stream placed less than 4 Kbps of load on any one MBone link. We reserved multicast



(a) Inferred vs. directly measured loss rates (b) Difference between the two

Figure 5.2. Loss rates on link to CA when running `mtrace` from USC. CA experienced an order of magnitude lower loss rates than GA (see Figs. 5.4 and 5.5). Nevertheless, inferred and directly measured loss rates agreed closely. Differences were usually below 0.5%, never above 2%, while loss rates varied between 0 and 4%.

Physical location	**Abbreviation**
Atlanta, Georgia	GA
Cambridge, Massachusetts	MA
San Francisco, California	CA
West Orange, New Jersey	NJ

Table 5.1. Routers at multicast branch points during our representative MBone experiment.

address 224.2.130.64 and port 22778 for our experiments using the `sdr` session directory tool [16]. At each receiver, we ran the `mtrace` [41] and `mbat` [27] tools to gather statistics about traffic on this multicast group. Below we describe our use of `mtrace` and `mbat` in more detail.

5.3.1 Direct measurements

`mtrace` traces the *reverse* path from a multicast source to a receiver. It runs at the receiver and issues trace queries that travel hop-by-hop up the multicast tree towards the source. Each router along the path responds to these queries with information about traffic

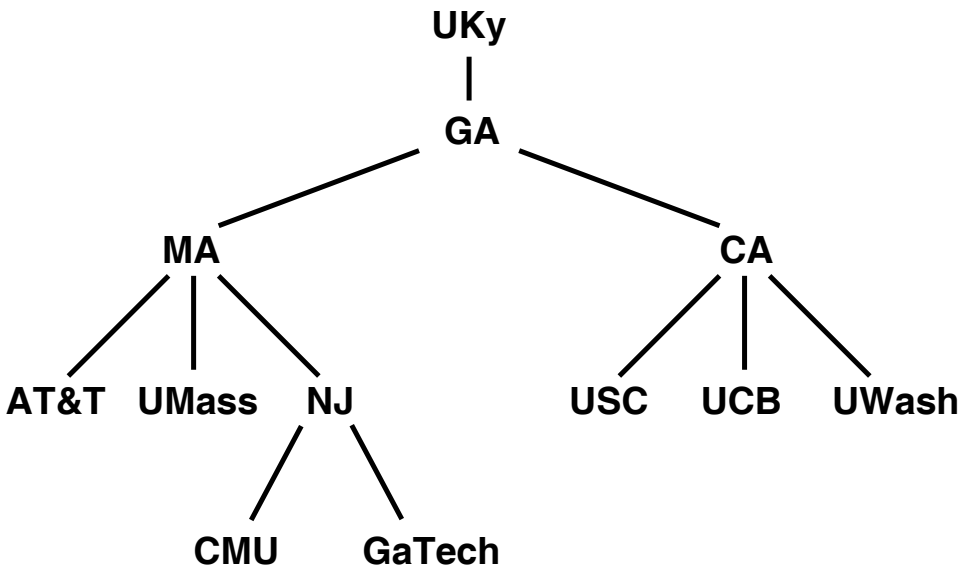
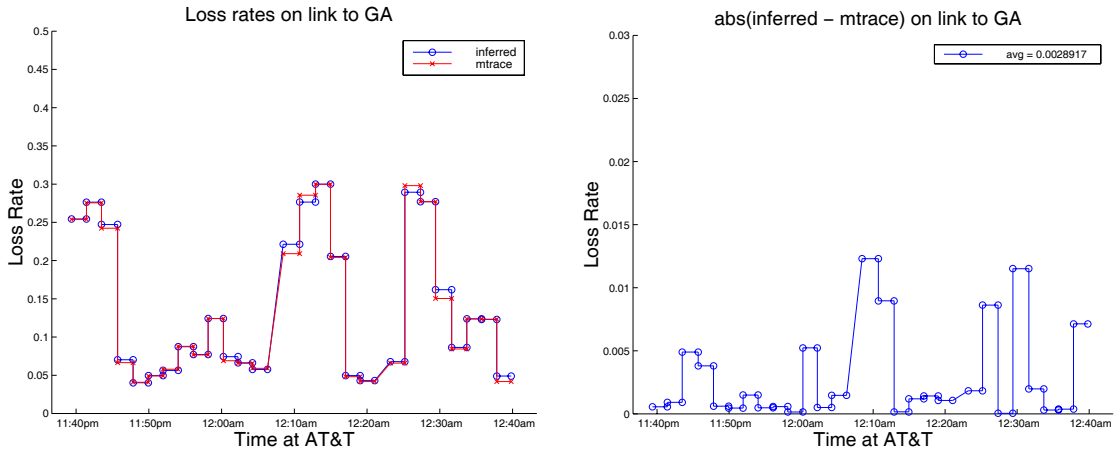


Figure 5.3. Multicast routing tree during our representative MBone experiment.

on the specified multicast group as seen by that router, including counts of incoming and outgoing packets. `mtrace` calculates packet losses on a link by comparing the packet counts returned by the two routers at either end of the link.

In each of our experiments, we collected `mtrace` statistics for consecutive two-minute intervals over the course of one hour. We ran a separate instance of `mtrace` for each interval. Each `mtrace` run issued a trace query at the beginning of the interval and another query at the end. We thus measured link-level loss rates for all thirty intervals in one hour as shown in Figures 5.2, 5.4, and 5.5. These intervals are not exactly two minutes long due to delays incurred in collecting responses to the queries. We recorded timestamps for the actual beginning and end of each `mtrace` run to help synchronize our inference calculations to these direct measurements.

We chose to measure two-minute intervals based on previous observations from [6]. The simulations have shown that the statistical inference algorithm at the heart of MINC converges to true loss rates after roughly 1,000 observations. Given the 100 milliseconds between probes in our MBone experiments, two minutes allow for 1,200 probes between



(a) Inferred vs. directly measured loss rates (b) Difference between the two

Figure 5.4. Loss rates on link to GA when running `mtrace` from AT&T. The two sets of loss rates agreed closely over a wide range of values. Differences remained below 1.5% while loss rates varied between 4 and 30%.

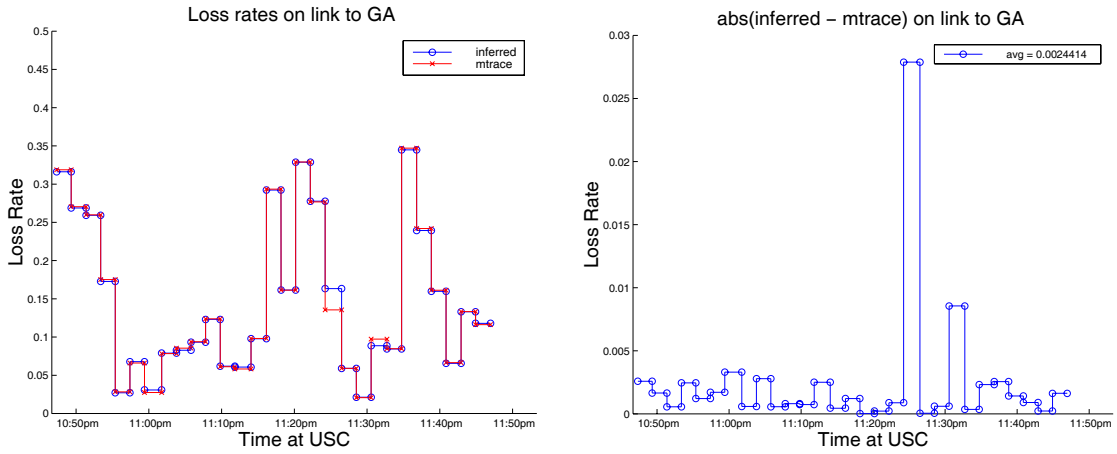
measurements. As shown in Figure 5.6, 1,200 probes were indeed enough for MINC to converge.

It is important to note that `mtrace` does not scale to measurements of large multicast groups if used in parallel from all receivers as we describe here. Parallel `mtrace` queries come together as they travel up the tree. Enough such queries will overload routers and links with measurement traffic. We used `mtrace` in this way only to validate MINC on relatively small multicast groups before we move on to use MINC alone on larger groups.

5.3.2 Inference calculations

We encoded the loss inference algorithm in a program called `infer`. `infer` takes two inputs: a description of the tree topology and a description of the end-to-end losses experienced by each receiver. It produces as output the estimated loss rates on every link in the tree.

We determined the tree topology by combining the `mtrace` output from all the receivers. Along with packet counts, `mtrace` reports the domain name and IP address of each router on the path from the source to a receiver. We built a complete multicast tree by



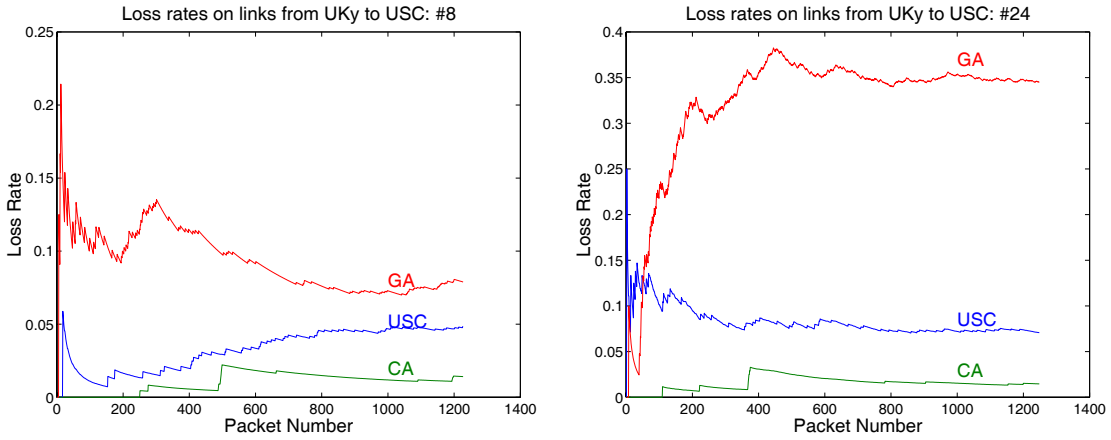
(a) Inferred vs. directly measured loss rates (b) Difference between the two

Figure 5.5. Loss rates on link to GA when running `mtrace` from USC. These measurements span different two-minute intervals than those from AT&T (see Fig. 5.4) because of clock asynchrony. Nevertheless, inferred and directly measured loss rates agreed closely. Differences were usually below 0.5%, never above 3%, while loss rates varied between 2 and 35%.

looking for common routers and branch points on the paths to all the receivers. The topology of the MBone is relatively static due to that network's current reliance on manually configured IP-over-IP tunnels. These tunnels are themselves logical links that may each contain multiple physical links. We verified that the topology remained constant during our experiments by inspecting the path information we obtained every two minutes from `mtrace`.

We measured end-to-end losses using the `mbat` tool. `mbat` runs at a receiver, subscribes to a specified multicast group, and collects a trace of the incoming packet stream, including the sequence number and arrival time of each packet. We ran `mbat` at each receiver for the duration of each experiment. At the conclusion of an experiment, we transferred the `mbat` traces and `mtrace` output from all the receivers to a single location.

There we ran the loss inference algorithm on the same two-minute intervals on which we collected `mtrace` measurements. For each receiver, we used the timestamps for the



(a) Eighth measurement interval (b) Twenty-fourth measurement interval

Figure 5.6. Inferred loss rates on the three links between UKy and USC (i.e., the links to GA, CA, and USC) during individual 2-minute, 1200-probe measurement intervals. The inference algorithm converged well before the measurement interval ended for all links during all measurement intervals.

beginning and end of `mtrace` measurements to segment the `mbat` traces into corresponding two-minute subtraces. Then we ran `infer` on each two-minute interval and compared the inferred loss rates with the directly measured loss rates. We discuss the results in the next section.

5.3.3 Experimental Results

We performed a number of MBone experiments using different multicast sources and receivers, and thus different multicast trees. Inferred loss rates agreed closely with directly measured loss rates throughout our experiments. Here we discuss results from a representative experiment on August 26, 1998. Table 5.1 lists the branch routers involved in this experiment, and the end hosts are listed in Appendix B, while Figure 5.3 shows the resulting multicast tree.

Figure 5.4 shows that inferred and directly measured loss rates agreed closely despite a link experiencing a wide range of loss rates. In this case, loss rates as measured by

`mtrace` varied between 4 and 30%. Nevertheless, differences between inferred and directly measured loss rates remained below 1.5%.

Figures 5.2, 5.4, and 5.5 all show that inferred and directly measured loss rates agreed closely despite imperfect synchronization between `infer` and `mtrace` intervals. The two sets of intervals do not always match because of variable network delays. The timestamps for the beginning and end of `mtrace` intervals are recorded before a trace query is issued and after a trace query returns, both according to the clock at the relevant receiver. However, the corresponding packet counts are recorded at the time the trace query arrives at each router. Therefore, although the `infer` intervals are derived from the `mtrace` intervals using the same receiver clock, the inference is not always applied to exactly the same 1,200 probe packets as the direct loss measurement. Nevertheless, differences between inferred and directly measured loss rates across Figures 5.2, 5.4, and 5.5 were usually well below 1%, never above 3%.

Along the same lines, Figures 5.4 and 5.5 together show that inferred and directly measured loss rates agreed closely for different two-minute intervals on the same link. We have multiple sets of `mtrace` measurements for links shared by multiple receivers, one set for each receiver. In these cases, we can run `infer` on different sets of intervals corresponding to the different sets of `mtrace` intervals. `mtrace` intervals are different for each receiver because of clock asynchrony between receivers and because of the variable network delays discussed above. Nevertheless, differences between inferred and directly measured loss rates across Figures 5.4 and 5.5 remained below 3%.

Figure 5.2 shows that inferred and directly measured loss rates agreed closely even for links with very low loss rates. In this case, loss rates varied between 0 and 4%, an order of magnitude lower than the loss rates in Figure 5.4. Nevertheless, differences between inferred and directly measured loss rates were usually below 0.5%, never above 2%.

Finally, Figure 5.6 shows that the inference algorithm converged quickly to the desired loss rates. Each inferred loss rate reported in Figures 5.2, 5.4, and 5.5 is the value

calculated by `infer` at the end of the corresponding 2-minute, 1200-probe measurement interval. However, `infer` outputs a loss rate value for every probe. Figure 5.6 reports these intermediate values. As shown, inferred loss rates stabilized well before a measurement intervals ends. Our algorithm converged after fewer than 800 probes for all links and all measurement intervals in our experiments.

5.4 Parametric Analysis

Previous work [5, 6] on the performance of MINC shows that the MLE estimates converge to the true loss rates as the number of observations goes to infinity. When simulating the idealized model, there is not only the true loss rate, α, but also the measured loss rate, we note as $\tilde{\alpha}$, and in many cases what is of interest is how closely $\widehat{\alpha}$ comes to $\tilde{\alpha}$ and not to α. In Section 5.3 we use `mbat` to obtain $\tilde{\alpha}$ on all the links of the multicast tree, and compare it with the MINC estimate, $\widehat{\alpha}$.

The loss rate of a link is determined by the amount of traffic and congestion on the link, and also the time period during which it is meausured. Usually one is interested in loss rates measured over an interval of length that ranges from 100ms to several minutes, since many applications and protocols adapt to the network congestion in those time scales, and a loss rate is an important indicator of a congestion.

In this section we extend the work further and investigate the difference between directly measured loss rates and the MINC estimates under a wide range of values for parameters. We introduce three measures to base our comparisons on, run simulations varying the link loss rates, tree depth, and tree branching factor, and using a limited number of observations, and present the results.

5.4.1 Difference between inferred and directly measured loss rates

In our MBone experiments we use 1,200 probe packets or 2 minutes as an inference interval. We compared the inferred link loss rates, $\widehat{\alpha}_k, k = 1, \ldots, N$, where N is the total

number of nodes in a tree, with the actual loss rates measured by `mtrace`. We note that the directly measured loss rates are not the true loss rates, $\alpha_k, k = 1, \ldots, N$: the loss rate of a link inside the network is not a fixed constant, but changes as congestion occurs and is resolved.

Let us denote the directly measured loss rates as $\tilde{\alpha}_k, k = 1, \ldots, N$. The difference between inferred and directly measured loss rates in Figures 5.4 to 5.2 is not $\widehat{\alpha}_k - \alpha_k$, but $\widehat{\alpha}_k - \tilde{\alpha}_k$.

We design a set of simulations to study the difference between inferred and directly measured loss rates, and compare it with the difference between inferred and true loss rates. Also we study the impact of tree topologies on the accuracy of inferred loss rates. The parameters of the simulations are the link loss rates, the height of a multicast tree, and the branching factor at an internal node of the tree. All internal nodes of a tree have the same number of childen as specified by the branching factor. Each simulation run generates 1000 packets. All the links of a tree are assumed to have the same loss rate. We vary the link loss rate from 0.01 to 0.3.

In the analysis we use the following three performance metrics. Let $\mathcal{T} = (V, L)$ denote the multicast tree, consisting of the set of nodes V, including the sender and the receivers, and the set of links L.

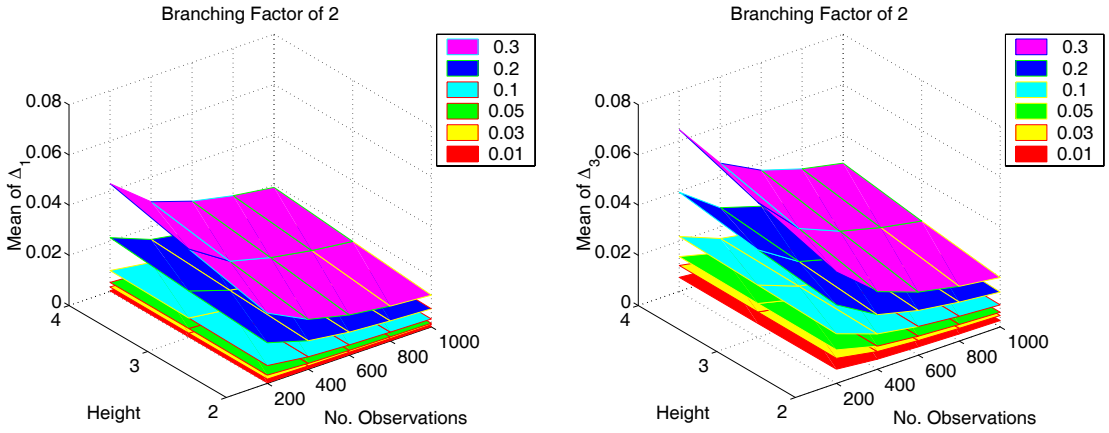
$$\Delta_1 = \frac{1}{|L|} \sum_{k \in L} |\widehat{\alpha}_k - \tilde{\alpha}_k| \tag{5.6}$$

$$\Delta_2 = \frac{1}{|L|} \sum_{k \in L} \max \left(\frac{\widehat{\alpha}_k(\epsilon)}{\tilde{\alpha}_k(\epsilon)}, \frac{\tilde{\alpha_k}(\epsilon)}{\widehat{\alpha}_k(\epsilon)} \right) \tag{5.7}$$

$$\text{where} \quad p(\epsilon) = \max(\epsilon, p) \quad \text{and} \quad \epsilon = 10^{-}3 \tag{5.8}$$

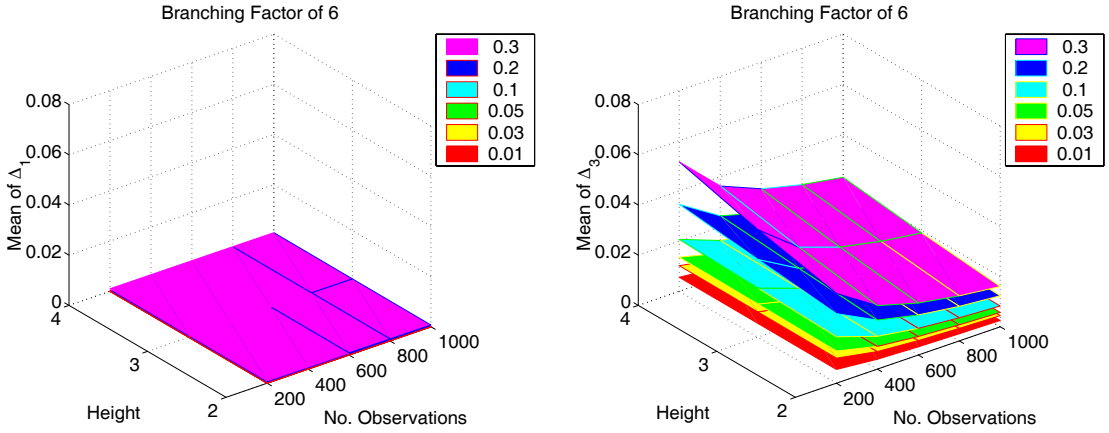
$$\Delta_3 = \frac{1}{|L|} \sum_{k \in L} |\widehat{\alpha}_k - \alpha_k| \tag{5.9}$$

Note that Δ_2 is a sample mean of the error factor in (16) from [6]. Δ_1 is the average of the absolute difference between the inferred and directly measured loss rates, while



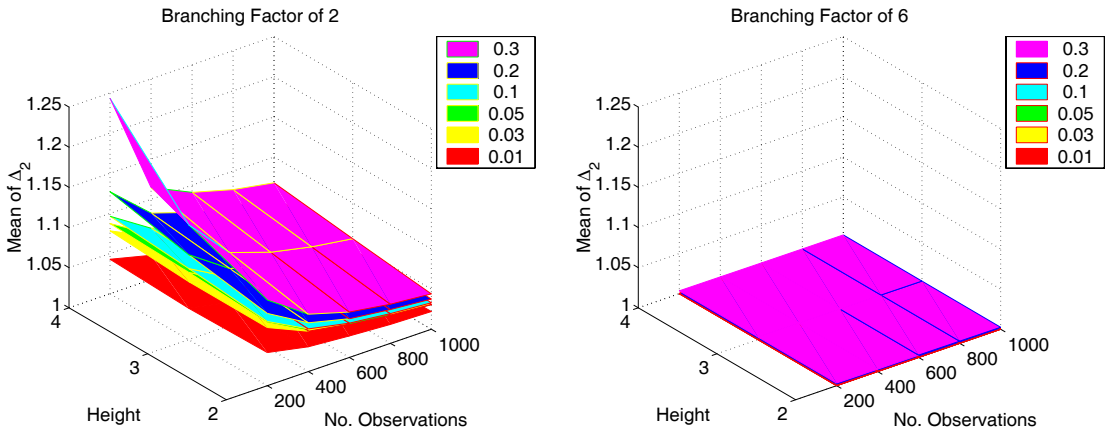
(a) Δ_1 (b) Δ_3

Figure 5.7. Δ_1 and Δ_3; All the links on the tree have the same link loss rates; The branching factor is 2, and the height varies from 2 to 4. The legend shows the loss rates.



(a) Δ_1 (b) Δ_3

Figure 5.8. Δ_1 and Δ_3; All the links on the tree have the same link loss rates; The branching factor is 6, and the height varies from 2 to 4. The legend shows the loss rates.



(a) Δ_2 (b) Δ_2

Figure 5.9. Δ_2; All the links on the tree have the same link loss rates; The branching factor is 2 in (a) and 6 in (b); The height varies from 2 to 4. The legend shows the loss rates.

Δ_2 uses the ratio between them. Δ_3 is the average of the absolute difference between the inferred and true loss rates over all of the links on a tree.

We first compare Δ_1 and Δ_3. In Figures 5.7 and 5.8 x-axis is the height of the tree, and y-axis is the number of probe packets. In Figure 5.7(a) the z-axis plots the sample mean of Δ_1 in 1000 simulation runs; In Figure 5.7(b), the sample mean of Δ_3. We observe in the figures that Δ_1 is consistently less than Δ_3 for the same values for a loss rate and a height. Figure 5.8 illustrates the sample averages of Δ_1 and Δ_3 when the branching factor of an internal node of the tree is set at 6. The x and y axes plot the height of the tree and the number of probe packets as in 5.7. As in Figure 5.7 Δ_1 is consistently less than Δ_3.

We compare Δ_1 between Figure 5.7(a) and Figure 5.8(a). As the branching factor increases from 2 to 6, Δ_1 decreases consistently over the entire range of loss rates and number of probe packets. This is consistent with the analytical and simulation results from [5, 6]. We note that even when a relatively small number of 200 probe packets are used and the loss rate is as high as 0.3, Δ_1 lies below 0.01.

In Figure 5.9 the x and y-axes plot the height and the number of probe packets as in Figures 5.7 and 5.8; The z-axis plots Δ_2. As in the case of Δ_1, Δ_2 is less as the loss rate decreases, the height decreases, and the branching factor increases.

The simulation results show that the difference between the inferred and directly measured loss rates is less than that between the inferred and true loss rates. It is an encouraging finding in the sense that we generally need a smaller number of probe packets to infer the actual link loss rates. It should be an advantage to applications that are intended in obtaining quick estimates of loss rates.

5.5 Related Work

A growing number of measurement infrastructure projects (e.g., AMP [2], Felix [15], IPMA [19], NIMI [43], Surveyor [56], and Test Traffic [57]) aim to collect and analyze end-to-end performance data for a mesh of unicast paths between a set of participating hosts. We believe our multicast-based inference techniques would be a valuable addition to these measurement platforms. As mentioned in the previous section, we are working to incorporate MINC capabilities into NIMI.

A lot of recent experimental work has sought to understand internal network behavior from end-to-end performance measurements (e.g., see [7, 32, 44, 45]). In particular, `pathchar` [21] is under evaluation as a tool for inferring link-level statistics from end-to-end unicast measurements. Much work remains to be done in this area and with MINC we are contributing a novel multicast-based methodology.

Regarding multicast-based measurements, we have already described the `mtrace` tool [41]. In addition, the `tracer` tool [29] performs topology discovery through the use of `mtrace`. However, `mtrace` suffers from performance and applicability problems in the context of large-scale Internet measurements. First, as mentioned earlier in this chapter, `mtrace` needs to run once for each receiver in order to cover a complete multicast tree. This behavior does not scale well to large numbers of receivers. In contrast, MINC

covers the complete tree in a single pass. Second, `mtrace` relies on multicast routers to respond to explicit measurement queries. Although current routers support these queries, Internet Service Providers (ISPs) may choose to disable this feature since it gives anyone access to detailed delay and loss information about paths inside their networks. In contrast, MINC does not rely on cooperation from any network-internal elements.

5.6 Conclusions

We have presented experimental results that validate the MINC approach to inferring link-level loss rates from end-to-end multicast measurements. We compared loss rates in MBone tunnels as inferred using our technique and as measured by `mtrace`. Inferred values closely matched directly measured values – differences were usually well below 1%, never above 3%, while loss rates varied between 0 and 35%. In addition, our inference algorithm quickly converged to the directly measured loss rates – inferred values stabilized well within 2-minute, 1200-probe measurement intervals. Through simulations we show that the MINC estimates converge more quickly to the directly measured loss rates than to the true loss rates over a wide range of parameter values.

We feel that MINC is an important new methodology for network measurement, particularly Internet measurement. It does not rely on network cooperation and it scales to very large networks. MINC is firmly grounded in statistical analysis that is backed up by packet-level simulations and now experiments under real network conditions.

CHAPTER 6

CONCLUSIONS

6.1 Summary of the Dissertation

In this section we summarize the research presented in this dissertation. In Chapter 1 we presented the general framework

In Chapter 2 we have presented a framework for understanding the systematic errors introduced in one-way network delay measurements by unsynchronized clocks, and discussed several properties desirable of a skew estimation algorithm. We developed linear programming based (LP) algorithm to estimate the clock skew in network delay measurements and compared it with three other algorithms, namely, Paxson's, linear regression, piece-wise minimum algorithms. We show that the LP algorithm has time complexity of $O(N)$, where N is the number of delay measurements, and is robust in the sense that the error margin of the skew estimate is independent of the magnitude of the skew. We use traces of real Internet delay measurements to assess the algorithm, and compare its performance to that of three other algorithms. For a trace with a relatively low loss rate of 2.7%, and a conspicuous skew, all three algorithms perform well with the exception of the linear regression algorithm. For a trace with a very high loss rate of 42%, and conspicuous skew, only LP and Paxson's algorithms perform reasonably. Furthermore, we show through simulation that the LP algorithm is unbiased, and that the sample variance of the sskew estimate is better (smaller) than Paxson's algorithm

In Chapter 3 we have considered the problem of adaptively adjusting the playout delay in order to keep this delay as small as possible, while at the same time avoiding excessive "loss" due to the late arrival of packets at the reeiver after their playout time has already

passed. The contributions of this chapter are twofold. First, given a trace of packet audio receptions at a receiver, we have presented efficient algorithms for computing a bound on the achievable performance of any playout delay adjustment algorithm. More precisely, we have computed upper and lower bounds (which are shown to be tight for the range of loss and delay values of interest) on the optimum (minimum) average playout delay for a given number of packet losses (due to late arrivals) at the receiver for that trace. Second, we have presented a new adaptive delay adjustment algorithm that tracks the network delay of recently received packets and efficiently maintains delay percentile information. This information, together with a "delay spike" detection algorithm is used to dynamically adjust talkspurt playout delay. We have shown that this algorithm outperforms existing delay adjustment algorithms over a number of measured audio deleay traces and performs close to the theoretical optimum over a range of parameter values of interest.

In Chapter 4 we have examined the correlation between packet delay and loss. Our goal is to study the extent to which one performance measure can be used to predict the future behavior of the other (e.g., whether observed increasing delay is a good predictor of future loss). We collected six sets of delay and loss measurements in which periodic traffic was sent from a sender to a receiver. We have quantified the correlations between delay and loss as a sample mean of delay conditioned on loss, and loss conditioned on delay, of which threshold varies from $\widehat{d}$ to $2\widehat{d}$, where $\widehat{d}$ is the sample mean. The results show that the loss conditioned on delay is not sensitive to the threshold values of delay, and the sample mean of delay conditioned on loss is a better measure in predicting an upcoming loss.

In Chapter 5 we have presented MBone experiments that validate an end-to-end measurement technique called MINC. The key to this approach is to track the correlation in the end-to-end losses observed by receivers, and use it to infer link-level loss rates on the multicasting routing tree from a sender to receivers. We validate MINC by comparing the loss rates on internal MBone tunnels as inferred using our technique and as measured

using the `mtrace` too. Inferred values closely matched directly measured values - differences were usually well below 1% , never above 3%, while loss rates varied between 0 and 35%. We also have run simulations to study the performance of MINC estimates given a limited number of observations, and under a wide range of loss rates and tree topology. The simulation results show that the difference between the inferred and directly measured loss rates is less than that between the inferred and true loss rates.

6.2 Issures for Future Research

In this section we indicate several issues of future work which follow naturally from the work in this dissertation.

The knowledge of the network-internal performance can be used in many different ways, depending on the need of applications. From public multicast to web TV, applications require different levels of reliability and adapt differently to network congestion. We will investigate how multicast applications can use the MINC approach to improve their performance.

The Multicast Routing Monitor(MRM) effort of the IETF MBONED working group is to help debug problems on the MBone. The MINC approach can be employed in MRM, and help extensive debugging of the MBone. We propose to investigate how to merge these two efforts and demonstrate the strength of the merged approach.

APPENDIX A

PSEUDO CODE FOR ALGORTHMS IN CHAPTER 2

A.1 Pseudo Code for Linear Programming Algorithm

1. PROCEDURE: IN($\tilde{d}_i, \tilde{t}_i^s, N$) OUT($\widehat{\alpha}, \widehat{\beta}$)
2. // $line$(slope, y-intercept) returns a line.
3. // x(line1, line2) and y(line1, line2) return x- and y-coordinates
4. // of intersection.
5. $n_1 = 1,\ n_2 = 2,\ k = 2$
6. FOR $i = 3$ to N
7. FOR $j = k$ downto 2
8. IF $x(line(\tilde{t}_i^s, -\tilde{d}_i), line(\tilde{t}_{n_j}^s, -\tilde{d}_{n_j}))$
9. $> x(line(\tilde{t}_{n_j}^s, -\tilde{d}_{n_j}), line(\tilde{t}_{n_{j-1}}^s, -\tilde{d}_{n_{j-1}}))$
10. BREAK;
11. ENDIF
12. ENDFOR
13. // Note that j is 1 when the FOR loop in line 6 runs to the end.
14. $k = j + 1, n_k = i$
15. ENDFOR
16. $opt_s = \sum_i \tilde{t}_i^s / N$

```
17.   FOR  i = 1 to k − 1
18.     IF  (t̃^s_{n_i} < opt_s) AND (opt_s < t̃^s_{n_{i+1}})
19.       α̂ − 1 = x(line(t̃^s_{n_i}, −d̃_{n_i}), line(t̃^s_{n_{i+1}}, −d̃_{n_{i+1}})),
20.       β̂ = y(line(t̃^s_{n_i}, −d̃_{n_i}), line(t̃^s_{n_{i+1}}, −d̃_{n_{i+1}}))
21.       BREAK;
22.     ENDIF
23.   ENDFOR
24. ENDPROCEDURE
```

A.2 Pseudo Code for Paxson's Algorithm

```
1.  PROCEDURE: IN(d̃_i, t̃^s_i, N)  OUT(α̂)
2.  //  slope(p1, p2) returns the slope of the line
3.  //  R(n, k) is from [47]
4.  //  threshold is either 10^−3 or 10^−6.
5.    M = ⌊√N⌋
6.    FOR  i = 1 to M
7.      m_i = min_{(i−1)M<j<≤iM} d̃_j,
8.      n_i = arg_j min_{(i−1)M<j≤iM} d̃_j
9.    ENDFOR
10.   IF  N − M^2 > M/2
11.     m_{i+1} = min_{(M^2+1)≤j≤N} d̃_j,
12.     n_{i+1} = arg_j min_{(M^2+1)≤j≤N} d̃_j,
13.     M = M + 1
14.   ENDIF
```

15. $G_s = median_{1 \le i,j \le k} slope((\tilde{t}^s_{n_i}, m_{n_i}), (\tilde{t}^s_{n_j}, m_{n_j}))$

16. // Here we assume that G_s is negative and thus the trend is decreasing.

17. $k = 1$

18. $cm_1 = m_1$

19. FOR $i = 2$ to M

20. IF $m_i < min_{1 \le j \le (i-1)} m_j$

21. $k = k + 1, cm_k = m_i, n_k = i$

22. ENDIF

23. ENDFOR

24. IF $R(M,k) > threshold$

25. RETURN(1)

26. ENDIF

27. $\widehat{\alpha} = median_{1 \le i,j \le k} slope((\tilde{t}^s_{n_i}, cm_{n_i}), (\tilde{t}^s_{n_j}, cm_{n_j}))$

28. ENDPROCEDURE

APPENDIX B

HOST NAME ABBREVIATIONS

Physical location	Abbreviation
AT&T Labs – Research, Florham Park, New Jersey	AT&T
Carnegie Mellon University, Pittsburgh, Pennsylvania	CMU
Georgia Institute of Technology, Atlanta, Georgia	GaTech
GMD FOKUS - Research Institute for Open Communiocation Systems, German National Research Center for Information Technology, Berlin, Germany	GMD
INRIA - French National Institute of Research in Computer Science and Control	INRIA
Osaka University, Osaka, Japan	Osaka
University of California, Berkeley, California	UCB
University of Kentucky, Lexington, Kentucky	UKy
University of Massachusetts, Amherst, Massachusetts	UMass
University of Southern California, Los Angeles, California	USC
University of Texas, Austin, Texas	UTexas
University of Virginia, Charlottesville, Virginia	UV
University of Washington, Seattle, Washington	UWash
Washington University at St. Louis, St. Louis, Missouri	WashU

Table B.1. End Hosts Names and Their Abbreviations

BIBLIOGRAPHY

[1] Alvarez-Cuevas, Felipe, Bertran, Miquel, Oller, Francesc, and Selga, Joseph M. Voice synchronization in packet switching networks. *IEEE Networks Magazine 7*, 5 (September 1993), 20–25.

[2] Active measurement project. http://amp.nlanr.net.

[3] Bertsekas, Dimitri P. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall Inc., Englewood Cliffs, NJ 07632, 1987.

[4] Bolot, Jean-Chrystostome. End-to-end packet delay and loss behavior in the Internet. In *Proceedings of SIGCOMM '93* (1993), pp. 289–298.

[5] Cáceres, Ramón, Duffield, Nick, Horowitz, Joseph, and Towsley, Don. Multicast-based inference of network-internal characteristics. Tech. Rep. Computer Science Technical Report 98-17, University of Massachusetts at Amherst, February 1998.

[6] Cáceres, Ramón, Duffield, Nick G., Horowitz, Joseph, Towsley, Don, and Bu, Tian. Multicast-based inference of network-internal loss characteristics. In *Proceedings of INFOCOM '99* (New York, 1999).

[7] Carter, R. L., and Crovella, M. E. Measuring bottleneck link speed in packet-switched networks. In *PERFORMANCE '96* (October 1996).

[8] Casner, Stephen, and Deering, Stephen. First IETF Internet Audiocast. *ACM Computer Communication Review* (July 1992), 92–97.

[9] Clark, David D. The design philosophy of the DARPA internet protocols. In *Proceedigs of SIGCOMM '88* (1988), pp. 106–114.

[10] Cohen, Danny. Issues in transnet packetized voice communication. In *Proc. Fifth Data Communications Symposium* (Snowbird, UT, September 1977).

[11] Deering, S., and Hinden, R. Internet Protocol, version 6 (IPv6) specification. RFC 1883, Internet Engineering Task Force, December 1995.

[12] Dixit, Sudhir, and Skelly, Paul. MPEG-2 over ATM or video dialtone networks: issues and strategies. *IEEE Network 9*, 5 (September-October 1995), 30–40.

[13] Dyer, M. E. Linear algorithm for two- and three-variable linear programs. *SIAM Journal on Computing 13* (1983), 31–45.

[14] Eriksson, Hans. MBONE:the multicast backbone. *Communications of ACM 37*, 8 (August 1994).

[15] Felix: Independent monitoring for network survivability. for more information see `ftp://ftp.bellcore.com/pub/mwg/felix/index.html`.

[16] Handley, M., and Jacobson, V. Session directory tool(sdr). http://www-mice.cs.ucl.ac.uk/multimedia/software/sdr.

[17] Hardman, Vicky, and Kouvelas, Isidor. Robust audio tool(rat). `http://www.vcas.video.ja.net/mice/rat/info.html`.

[18] Hoaglin, D., Mosteller, F., and Tukey, J., Eds. *Understanding Robust and Exploratory Data Analysis*. John Wiley & Sons, 1983.

[19] IPMA internet performance measurement and analysis. For more information see `http://www.merit.edu/ipma`.

[20] ITU, Telecommunication Standardization Sector Of. ITU-T Recommendation G.114. Tech. rep., International Telecommunication Union, March 1993.

[21] Jacobson, Van. Pathchar - a tool to infer characteristics of internet paths. For more information see `ftp://ftp.ee.lbl.gov/pathchar`.

[22] Jacobson, Van. Tutorial notes: Multimedia conferencing on the Internet. at ACM SIGCOMM '94, 1994.

[23] Jacobson, Van, and McCanne, Steven. vat: audio conferencing tool. `ftp://ftp.ee.lbl.gov/conferencing/vat/`.

[24] Jayant, N. S. Effects of packet loss on waveform coded speech. In *Proc. Fifth Int. Conference on Computer Communications* (Atlanta, GA, October 1980).

[25] Jon Postel, editor. Transmission control protocol specification. RFC 793, Internet Engineering Task Force, September 1981.

[26] Kasera, Sneha, Kurose, Jim, and Towsley, Don. Exploring the dynamic behaviour of the internet using ip options. Tech. Rep. 96-12, Department of Computer Science at University of Massachusetts at Amherst, Amherst, MA 01003, March 1996.

[27] Kasera, Sneka, and Rizkalla, Mike. Mbone analysis tool(mbat). ftp://www-net.cs.umass.edu/pub/mist.

[28] Kirstein, Peter, Clayman, Stuart, Handley, Mark, and Sasse, Angela. Recent activities in the MICE conferencing project. In *Proceedings of INET '95* (Honolulu, Hawaii, June 1995).

[29] Levine, Brian Neil, Paul, Sanjoy, and Garcia-Luna-Aceves, J. J. Organizing multicast receivers deterministically by packet-loss correlation. In *Proceedings of ACM Multimedia '98* (Bristol, UK, September 1998).

[30] Ljung, Lennart, and Söderstrom, Törsten. *Theory and Practice of Recursive Identification*. MIT Press, 1983.

[31] Macedonia, M., and Brutzman, D. Mbone provides audio and video across the internet. *IEEE Computer Magazine* (April 1994), 30–35.

[32] Mathis, M., and Mahdavi, J. Diagnosing internet congestion with a transport layer performance tool. In *Proceedings of INET '96* (Montreal, June 1996).

[33] Megiddo, N. Linear time algorithm for linear programming in r^3 and related problems. *SIAM Journal on Computing 12*, 4 (November 1983), 759–776.

[34] Mills, D. Modelling and analysis of computer network clocks. Tech. Rep. 92-5-2, Electrical Engineering Department, University of Delaware, May 1992.

[35] Mills, D. Network time protocol(version 3): Specification, implementation and anlysis. Tech. rep., Network Information Center, SRI International, Menlo Park, CA, March 1992.

[36] MINC: Multicast-based inference of network-internal characteristics. For more information see `http://www.research.att.com/ duffield/minc/`.

[37] Montgomery, Warren A. Techniques for packet voice synchronization. *IEEE Journal on Selected Areas in Communications 6*, 1 (December 1983), 1022–1028.

[38] Moon, Sue B. Multicast/unicast traffic generator. ftp://www-net.cs.umass.edu/pub/sbmoon/mgen.tar.gz.

[39] Moon, Sue B., Kurose, Jim, and Towsley, Don. Packet audio playout delay adjustment: Performance bounds and algorithms. *ACM/Springer Multimedia Systems 5* (January 1998), 17–28.

[40] Moon, Sue B., Skelly, Paul, and Towsley, Don. Estimation and removal of clock skew from network delay measurements. Tech. Rep. 98-43, Department of Computer Science, University of Massachusetts at Amherst, Amherst, MA 01003, 1998.

[41] mtrace. For more information see `ftp://ftp.parc.xerox.com/pub/net-research/ipmulti`.

[42] NS: UCB/LBNL/VINT network simulator. http://www-mash.cs.berkeley.edu/ns/, 1997.

[43] Paxson, V., Mahdavi, J., Adams, A., and Mathis, M. An architecture for large-scale internet measurement. *IEEE Communications 36*, 8 (August 1998), 48–54.

[44] Paxson, Vern. End-to-end routing behavior in the internet. In *Proceedings of SIGCOMM '96* (August 1996), pp. 25–38.

[45] Paxson, Vern. End-to-end internet packet dynamics. In *Proceedings of SIGCOMM '97* (1997).

[46] Paxson, Vern. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, 1997.

[47] Paxson, Vern. On calibrating measurements of packet transit times. In *Proceedings of SIGMETRICS '98* (Madison, Wisconsin, June 1998).

[48] Ramjee, Ramachandran, Kurose, Jim, Towsley, Don, and Schulzrinne, Henning. Adaptive playout mechanisms for packetized audio applications in wide-area networks. In *Proceeding of INFOCOM '94* (1994).

[49] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), 277–288.

[50] Sanghi, D., Gudmundsson, O., Agrawala, A., and Jain, B.N. Experimental assessment of end-to-end behavior on Internet. In *Proceedings of INFOCOM '93* (1993), pp. 867–874.

[51] Sawashima, Hidenari, Hori, Yoshiaki, and Sunahara, Hideki. Characteristics of UDP packet loss: Effect of tcp traffic. In *Proceedings of INET '97: The Seventh Annual Conference of the Internet Society* (Kuala Lumpur, Malaysia, June 1997).

[52] Schulzrinne, H. RTP profile for audio and video conferences with minimal control. RFC 1890, Internet Engineering Task Force, jan 1996.

[53] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. RTP: A transport protocol for real-time applications. RFC 1889, Internet Engineering Task Force, jan 1996.

[54] Schulzrinne, Henning. Voice communication across the Internet: A Network Voice Terminal. Tech. rep., Dept. of ECE, Dept. of CS, University of Massachusetts, Amherst, MA 01003, July 1992.

[55] Shenker, Scott, Zhang, Lixia, and Clark, David D. Some observations on the dynamics of a congestion control algorithm. In *Proceedings of SIGCOMM '90* (1990), pp. 30–39.

[56] Surveyor. For more information see `http://io.advanced.org/surveyor`.

[57] Test traffic project. http://www.ripe.net/test-traffic.

[58] Weinstein, Clifford, and Forgie, James W. Experience with speech communication in packet networks. *IEEE Journal on Selected Areas in Communications 6*, 1 (1983), 963–980.

[59] Yajnik, Maya, Kurose, Jim, and Towsley, Don. Packet loss correlations in the Mbone multicast network: Experimental measurements and markov models. In *Proceedings of GLOBECOM '96* (November 1996).