Network Monitoring and Diagnosis
Based on Available Bandwidth Measurement

Ningning Hu

CMU-CS-06-122

May 2006

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Peter Steenkiste, Chair
Bruce Maggs
Hui Zhang
Albert Greenberg, AT&T Labs–Research

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored in part by (1) the Air Force Research Laboratory under contract nos. F30602-99-1-0518 and F30602-96-1-0287, (2) the US Army Research Office under contract no. DAAD19-02-1-0389, (3) the National Science Foundation under grant nos. CCR-0205266 and CNS-0411116, and (4) the Korea Information Security Agency in conjunction with CyLab Korea. The views and conclusions contained herein are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other governmental, commercial or legal entity.

Keywords: Network measurement, monitoring, diagnosis, available bandwidth, packet train, bottleneck link, source and sink tree

To my wife, Yun Zhou.

Abstract

Network monitoring and diagnosis systems are used by ISPs for daily network management operations and by popular network applications like peer-to-peer systems for performance optimization. However, the high overhead of some monitoring and diagnostic techniques can limit their applicability. This is for example the case for end-to-end available bandwidth estimation: tools previously developed for available bandwidth monitoring and diagnosis often have high overhead and are difficult to use.

This dissertation puts forth the claim that end-to-end available bandwidth and bandwidth bottlenecks can be efficiently and effectively estimated using packet-train probing techniques. By using source and sink tree structures that can capture network edge information, and with the support of a properly designed measurement infrastructure, bandwidth-related measurements can also be scalable and convenient enough to be used routinely by both ISPs and regular end users.

These claims are supported by four techniques presented in this dissertation: the IGI/PTR end-to-end available bandwidth measurement technique, the Pathneck bottleneck locating technique, the BRoute large-scale available bandwidth inference system, and the TAMI monitoring and diagnostic infrastructure. The IGI/PTR technique implements two available-bandwidth measurement algorithms, estimating background traffic load (IGI) and packet transmission rate (PTR), respectively. It demonstrates that end-to-end available bandwidth can be measured both accurately and efficiently, thus solving the path-level available-bandwidth monitoring problem. The Pathneck technique uses a carefully constructed packet train to locate bottleneck links, making it easier to diagnose available-bandwidth related problems. Pathneck only needs single-end control and is extremely light-weight. Those properties make it attractive for both regular network users and ISP network operators. The BRoute system uses a novel concept—source and sink trees—to capture end-user routing structures and network-edge bandwidth information. Equipped with path-edge inference algorithms, BRoute can infer the available bandwidth of all N^2 paths in an N-node system with only $O(N)$ measurement overhead. That is, BRoute solves the system-level available-bandwidth monitoring problem. The TAMI measurement infrastructure introduces measurement scheduling and topology-aware capabilities to systematically support all the monitoring and diagnostic techniques that are proposed in this dissertation. TAMI not only can support network monitoring and diagnosis, it also can effectively improve the performance of network applications like peer-to-peer systems.

Acknowledgments

First of all, I would like to thank my advisor, Professor Peter Steenkiste, without whom this dissertation will not be possible. Peter is kind, patient, and extremely responsible. It is Peter's kind encouragements and insightful directions that helped me overcome various barriers during my thesis research, and it is Peter's patient training that helped me significantly improve my speaking and writing skills. Peter's attitude, spirit, and experience on research influenced me profoundly, which will benefit my whole future career. It is fortunate to work with Peter for my six years of doctoral study.

I would also like to thank the other members of my thesis committee—Professor Bruce Maggs, Professor Hui Zhang, and Dr. Albert Greenberg. They all provided many insightful comments for my dissertation, which significantly improved it. Besides, I would also like to thank Bruce for his encouragement during several conference presentations. I want to thank Hui for his insightful comments and suggestions on my research directions. I would like to thank Albert for his discussions with me on real-world network problems, for his help in using my measurement tools in real systems, and for the three summer internship opportunities that he made available for me in AT&T Labs–Research.

Many people helped me in many aspects of my thesis, and I enjoyed the collaboration with them all. I would like to thank Aditya Ganjam for helping me integrating the TAMI system with the ESM system, which becomes an important application of TAMI. I want to thank Ming Zhang, now in Microsoft Research Redmond, for providing his source code to bootstrap the Pathneck development. I would like to thank Zhuoqing Morley Mao, now an assistant Professor in University of Michigan, for her discussion with me on BGP related problems. I want to thank Tom Killian from AT&T Labs–Research, who helped me address the last-hop problem in my original Pathneck design. I would like to thank Oliver Spatscheck from AT&T Labs–Research, for his help on deploying Pathneck on a real data center. I want to thank Li Erran Li from Bell Labs who helped me design the dynamic programming algorithm used by Pathneck. I would also like to thank other researchers in AT&T Labs–Research, Jennifer Yates, Aman Shaikh, Guangzhi Li, and Jia Wang, whose knowledge about Internet backbone networks significantly improved my understanding about real-world network operations.

I would like to thank other fellow students in Peter's group, Jun Gao, An-Cheng Huang, Urs Hengartner and Glenn Judd, who provided feedback on my papers and presentations in the last few years. In particular, I would like to thank Jun Gao and his wife Shuheng

Zhou, who not only provided encouragements on my research but also helped me adjust to the life in the US. I would also like to thank the lab administrators Nancy Miller and Christopher Lin, who helped me setting up experimental testbeds during the early stage of my thesis work.

I have made a lot of friends over the years in CMU, and I want to thank them all for being my friend: Mukesh Agrawal, Shang-Wen Cheng, Yang-Hua Chu, Eugene Ng, Sanjay Rao, Vyas Sekar, Adam Wierman, and Yinglian Xie. In particular, I want to thank Jichuan Chang, with whom I worked side by side in the RAINBOW group and on most of my first-year course projects. His optimism and encouragements helped me pass those tough early days in the US. I would also like to thank my officemates Julio Lopez and Rajesh Balan, both system experts. With them, I never need to step out the office to solve Linux system problems. I also thank them for attending most of my talks and proofreading my papers.

During my study in CMU, I was helped by many faculty and staff members. In particular, I would like to thank Sharon Burks, our "Den Mother", for taking care of many administrative issues, and making it trivial to go through those complex procedures. I would like to thank Barbara A Grandillo, Peter's secretary, who provided a lot of help on travel arrangements and meeting room reservations. I would like to thank Ms. Kathy Barnard, who used to be a teaching staff in CMU's International Communication Center. Kathy helped me build the confidence to improve my English communication skill, which is critical for my career.

I want to thank my teachers in China, without whom I will not be where I am now. In particular, I would like to thank Mrs. Guba Liu, my three-year high-school mathematics teacher. Her mother-like care helped me lay a solid foundation for both knowledge and personality for my future study. I will never forget what she told me—"to succeed in your career you must be a good person first".

I want to thank my family in China. My parents helped me recognize important things in life and taught me how to pursue them. They also helped me build a healthy life style, from which I have been and will always benefit. I would like to thank my elder sister, who is now an assistant professor in China. Her school experience helped me learn many things in an easy way and make right choices at various important stages of my early life. Specifically, I thank her for helping me regain interest in school life when I was 15, jumping over perhaps the toughest hurdle in my life so far. I also want to express my appreciation to my younger sister, who chooses to work at my hometown and take care of our parents, making it possible for me to focus on my study while being so far away from home.

Finally, I want to thank my lovely wife, Yun Zhou. I always feel I am a lucky guy just because I have Yun as my wife. It is her love, encouragements, and sacrifice that enable me to focus on my research and thesis writing. For that I dedicate this dissertation to her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Network Monitoring and Diagnosis

A network monitoring and diagnosis system periodically records values of network performance metrics in order to measure network performance, identify performance anomalies, and determine root causes for the problems, preferably before customers' performance is affected. These monitoring and diagnostic capabilities are critical to today's computer networks, since their effectiveness determines the quality of the network service delivered to customers. The most important performance metrics that are monitored include connectivity, delay, packet loss rate, and available bandwidth. (1) Network connectivity is probably the most important metric for a network monitoring and diagnosis system, since the top priority of a network service is to guarantee that any pair of end nodes can communicate with each other. Due to its importance, all network layers, starting from the physical layer, provide mechanisms to automatically monitor network connectivity. (2) Network delay is perhaps the most widely used performance metric in today's network applications. It is monitored mainly at the end-to-end level using ping. Network delay can be used to directly evaluate network path performance, especially for small data transmissions. (3) Packet loss rate refers to the probability that a packet gets dropped on a network path. It is mainly monitored at router interfaces using SNMP packet statistics. For ISPs (Internet Service Providers), since packet loss rate is a key marketing metric, a lot of monitoring and diagnostic effort is devoted to reducing packet loss. For applications, however, packet loss rate does not always significantly affect the performance of data transmissions. For example, single packet loss has a very limited impact on bulk data transmissions that use state-of-art TCP protocols. (4) Available bandwidth is another important performance metric, which directly captures data transmission speed. Although network delay can be used to evaluate the performance of a network path for small data transmissions, the available bandwidth metric is needed for larger data transmissions. However, available bandwidth is much less popular than the delay metric due to its high measurement overhead. People instead use the link load metric, which can be more easily measured (e.g., using SNMP), to capture available bandwidth information.

Network performance metrics can be measured at either a link level or an end-to-end level. Link-level information is easy to obtain since most network devices support link-level performance measurements. For example, link packet loss rate and link load can be measured using the SNMP protocol, and link connectivity can be monitored using routing protocol heart-beat messages. The problem with link-level monitoring, however, is that it is hard to extrapolate end-user performance purely based on link-level information. This is because end users' data flows often go through multiple ISPs' networks, and it is practically impossible to obtain link-level information from all these ISPs. Even if such access is possible, problems still remain: (a) fine-grain synchronization of the measurements on all the links along an end-to-end path is a hard technical problem, (b) it is often not immediately clear how to assemble the performance information (e.g., path delay variance) from multiple links to infer that of the whole path; (c) the overhead of transmitting and managing each link-level measurement can be prohibitive due to the large number of end users and the large number of links that each end-user may use.

End-to-end monitoring matches the experience of a real user's data transmission more closely. Since end-to-end monitoring does not require network internal information, it is not limited by ISPs' control over such data and can be easily used by regular end users or applications. For the same reason, end-to-end monitoring sometimes incurs much less measurement and management overhead. This is why ISP network operators also use end-to-end monitoring techniques such as pairwise pings, to monitor their network's performance, even though they also have access to link-level information. Despite these intriguing properties, end-to-end monitoring also has an obvious problem: it is often hard to design a technique to measure end-to-end performance. Currently the two most popular end-to-end monitoring techniques are ping and traceroute. Both can be used for delay and connectivity measurements, and traceroute is also popular for route measurements. It is much harder to obtain information on other metrics like packet loss rate and available bandwidth, which is why both are still active research topics.

My thesis research focuses on end-to-end available bandwidth monitoring and diagnosis. Throughout this dissertation, I will focus on how to efficiently measure available bandwidth and how to locate bandwidth bottlenecks. In the rest of this chapter, I first define the available bandwidth metric and explain its importance. I then present the thesis statement and technical contributions of this dissertation, which is followed by a discussion of related work. I conclude with a roadmap of the dissertation.

1.2 Available Bandwidth

We define available bandwidth as the residual bandwidth on a network path that can be used by a new data flow without disturbing the transmission of other flows on that path. That is, available bandwidth can be calculated as path capacity minus path load. Tech-

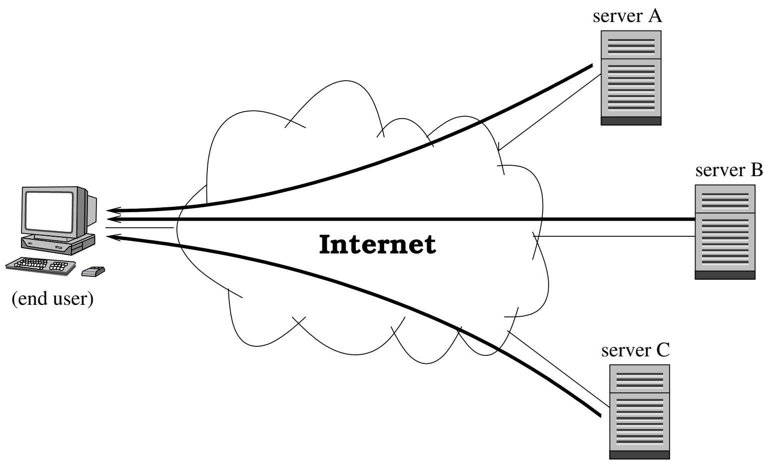


Figure 1.1: Application of end-to-end available bandwidth in server selection.

nically, the term *available bandwidth* is defined as follows. Consider an end-to-end path that has n links $L_1, L_2, ..., L_n$. Their capacities are $B_1, B_2, ..., B_n$, and the traffic loads on these links are $C_1, C_2, ..., C_n$. We define the *bottleneck link* as $L_b(1 \leq b \leq n)$, where $B_b = min(B_1, B_2, ..., B_n)$. The *tight link* is defined as $L_t(1 \leq t \leq n)$, where $B_t - C_t = min(B_1 - C_1, B_2 - C_2, ..., B_n - C_n)$. The unused bandwidth on the tight link, $B_t - C_t$, is called the *available bandwidth* of the path.

End-to-end available bandwidth information is important for many popular applications. I will use three representative applications to illustrate this importance. Figure 1.1 is a typical server-selection application, where an end user wants to download a file from a website. On today's Internet, it is often the case that there are multiple websites that provide the same file (e.g., a popular movie file), and the end user needs to choose one of them. Obviously, the server that has the highest downloading speed is the best choice, and to know that, the end user must have the end-to-end available bandwidth information from each of these websites.

The second type of application is peer-to-peer systems as illustrated in Figure 1.2. In such systems, end users collaborate with each other to improve the overall data transmission performance. Peer-to-peer systems are widely used by Internet end users since they can significantly improve end user's data transmission performance. In these systems, when a new user (e.g., user F in Figure 1.2) wants to join, the system needs to pick an existing system user (user A-E) to forward data packets to that new user. Since different users can have dramatically different data transmission performance, the selected user should have good data forwarding performance towards the new user. This selection procedure also requires the end-to-end available bandwidth information.

The third application in Figure 1.3 is on ISP network engineering. In this figure, the video server provides a video downloading service to its clients. This server is a cus-

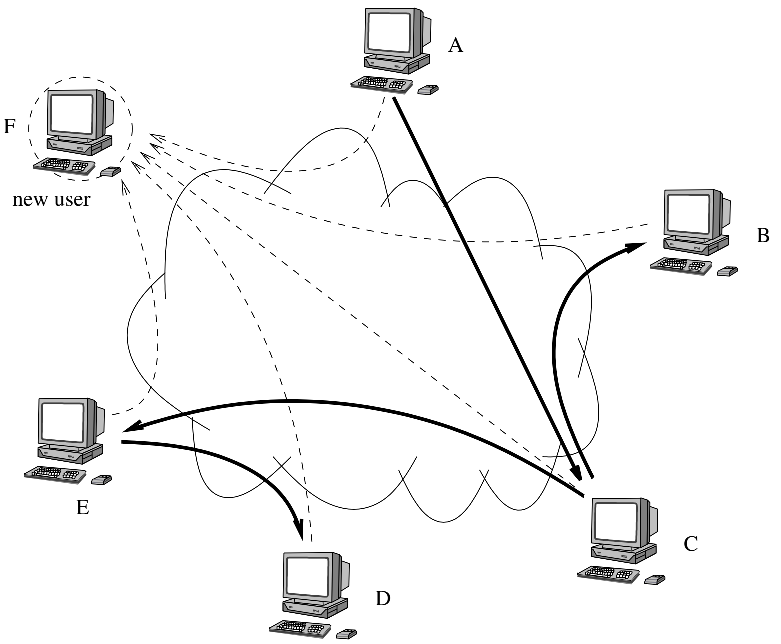


Figure 1.2: Application of end-to-end available bandwidth in P2P systems.

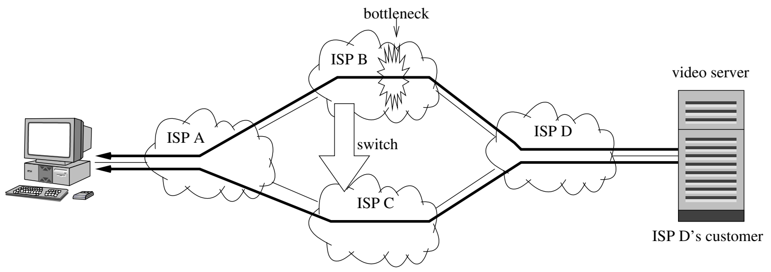


Figure 1.3: End-to-end available bandwidth diagnosis in ISP network engineering.

tomer of ISP D, which means that all data transmissions from the video server will first go through ISP D's network. To attract customers, ISP D often needs to help its customer to adapt to network performance changes. For example, ISP D initially routes the video server's traffic through ISP B to the clients connected with ISP A. If ISP B's network has problems and degrades the transmission performance, ISP D may want to reroute the video server's traffic through ISP C. In this case, ISP D has to use end-to-end measurement techniques since the problem is on a network to which it has no access. This application illustrates that end-to-end available bandwidth diagnostic techniques are also important for ISPs.

Despite the importance of end-to-end available bandwidth for network applications, it is not as widely used as the delay metric. One reason is that delay can be used to approximate available bandwidth information in many scenarios; another reason is that it is still hard to obtain end-to-end available bandwidth information. There are three major challenges in such measurements. First, Internet available bandwidth is very dynamic due to the burstiness of Internet background traffic. Second, unlike the delay measurement, which can be done by just using a small number of packets, estimating available bandwidth often requires a large number of measurement packets in order to “fill” the network path to obtain the available bandwidth information. The high overhead makes available-bandwidth measurement techniques hard to use in practice. Finally, there are few efficient techniques that can be used to diagnose available-bandwidth related problems. For example, there is no efficient way to identify the link that limits the available bandwidth between a pair of end nodes.

To address these challenges, we need the following capabilities: (1) efficient path-level available bandwidth monitoring and diagnosis capability; (2) efficient system wide available bandwidth monitoring and diagnosis capability; and (3) an infrastructure supporting these techniques and systems. Path-level monitoring capability is important because many network applications, especially those used by regular end users, only require path-level performance information. System wide monitoring capability is used to address monitoring problems for large-scale systems (like the peer-to-peer system in Figure 1.2), which often require different techniques than those used for path-level monitoring. For both path-level and system-level monitoring, efficiency is very important, because that determines whether the corresponding techniques can be widely used. Infrastructural support for measurement configurations and managements can significantly lower the bar for regular end users to use these techniques. In this dissertation, we developed the IGI/PTR and Pathneck tools to provide path-level monitoring and diagnosis capability; we proposed the BRoute system to provide system-level monitoring capability; we designed and developed the TAMI system to provide infrastructural support for both path-level and system-level monitoring and diagnosis capabilities.

1.3 Thesis Statement

This dissertation puts forth the claim that end-to-end available bandwidth and bandwidth bottlenecks on individual network paths can be efficiently and effectively estimated using packet-train probing techniques. Large-scale available bandwidth can be estimated efficiently by using the source and sink tree data structures to capture network edge information. Also with the support of a properly designed measurement infrastructure, bandwidth-related measurement techniques can be convenient enough to be used routinely by both ISPs and regular end users.

1.4 Thesis Contributions

This dissertation makes the following technical contributions:

- **An end-to-end available-bandwidth measurement technique.** We designed and developed an active measurement technique that uses packet trains to estimate end-to-end available bandwidth. This technique significantly reduces measurement overhead, while maintaining a similar measurement accuracy compared with other techniques. The implementation of this technique—the IGI/PTR tool—is publicly available [6] and has served as a benchmark for newly developed tools.

- **An Internet bottleneck link locating technique.** We developed a technique that uses a carefully constructed packet trains to quickly locate the bottleneck link of a network path. Its measurement overhead is several orders of magnitude lower than previously designed techniques. These properties make it possible to conduct an Internet-scale study of network bottlenecks for the first time in the research community. Pathneck's implementation [11] is also open source.

- **A thorough understanding of Internet bottleneck properties and their usage.** This dissertation characterizes bottleneck properties including bottleneck link location distribution, end-user access bandwidth distribution, bottleneck location persistence, and the relationship between bottleneck link and link loss/delay. The understanding of these properties helps us improve the performance of various network applications including overlay, multihoming, and content distribution.

- **A data structure that captures end-node routing topology.** We proposed to use the source and sink trees to capture end-node routing topologies and efficiently cover bandwidth bottlenecks. This concept not only helps design a large-scale available bandwidth measurement system, but also motivates a route-similarity metric that can help end users to quantify route sharing.

- **A large-scale available-bandwidth inference scheme.** Similar to the synthetic coordinate systems developed for network delay inference, we designed, validated, and implemented a system that addresses the overhead problem of available-bandwidth monitoring in large network systems. It only uses $O(N)$ overhead to measure the N^2 paths in a N-node system.

- **A bandwidth-oriented monitoring and diagnostic infrastructure.** This infrastructure provides a system support for complicated bandwidth-related monitoring and diagnostic operations. Its distinguishing characteristics are its topology-aware capability and its measurement scheduling functionality.

1.5 Related Work

Previous work on network monitoring and diagnosis can be roughly classified into two categories: measurement techniques, and monitoring and diagnostic systems.

1.5.1 Measurement Techniques

Measurement techniques can be classified based on the performance metric they measure. Since a same metric can be monitored at both the link-level and the end-to-end level, the corresponding techniques can be further labeled as link-level or end-to-end level monitoring techniques. For example, at link level network connectivity can be monitored using both physical-layer signals and IP-layer routing protocol heart-beat messages; while at the end-to-end level it is monitored using ping or traceroute.

The delay metric is measured using ping at both the link and end-to-end levels. However, link-level ping can only be done through the router command-line interface, i.e., manually, so it can not be used directly by a monitoring system. That is a key motivation for developing link-delay *inference* techniques. One simple method is to ping the two ends of link, and use the delay difference as an estimation of the link delay. Tulip [80] is an example of this method. Another method is to apply tomography techniques [31], which use probabilistic models to infer link delay based on multiple end-to-end delay measurements. At the path-level, besides ping, synthetic coordinate systems [88, 41] have been developed for large-scale delay inference. Note that all these techniques measure or infer either just the propagation delay (e.g., the synthetic coordinate systems) or the sum of propagation delay and queueing delay. None of them can directly quantify the queueing delay, which is an important metric for network congestion and delay variance. So far there have been no good techniques to quantify queueing delay, either at the link-level or at the end-to-end level, although there has been some work toward this goal [80][37].

The packet loss-rate metric at the link-level is measured using SNMP, which uses a counter to keep track of the total number of lost packets. End-to-end packet loss rate is hard to measure because Internet packet loss is very bursty. A reasonable estimation of path loss often requires a large number of sampling packets, making overhead a major concern. Sting [102] and Badabing [107] are perhaps the two best known packet loss-rate measurement tools. Sting leverages the TCP protocol to identify packet losses on both the forward and reverse paths. It uses the fast retransmission feature of the TCP protocol to force the packet receiver to acknowledge each data packet, and then identifies packet loss by comparing data-packet sequence numbers and ack-packet sequence numbers. Badabing estimates path loss rate by measuring both the loss episode frequency and the loss episode duration, which are sampled using two-packet or three-packet probings.

Available bandwidth at the link-level is measured using link capacity and traffic load information. Internet link capacity is generally known a priori, and the traffic load can be calculated using the statistics collected by SNMP. At the end-to-end level, people generally

Table 1.1: Classification of network monitoring and diagnostic systems

	Industry (ISP-oriented)	Academia (end-user oriented)
Passive	Sprint's IPMON [49]	OCXMON [84], CoralReef [38], IPMA [62], SPAND [104, 111]
Active	ATT active measurement system [36]	AMP [84], Surveyor [115], MINC [86], Scriptroute [110]
Both	NetScope [47]	NAI [84], IEPM [61], NIMI [93]

use tools like iperf [8] or ttcp [116] which use TCP flows' transmission performance in order to quantify path available bandwidth. Note these tools measure TCP achievable throughput, not the available bandwidth (i.e., the residual bandwidth) as we defined. This is because TCP achievable throughput is not only determined by the available bandwidth of the path, it is also affected by the level of multiplexing of background traffic flows and system configuration parameters such as TCP buffer sizes. This dissertation focuses on end-to-end available bandwidth measurement.

1.5.2 Monitoring and Diagnostic Systems

Monitoring and diagnostic systems provide system support and application interfaces for measurement techniques, thus lowering the bar for regular applications to use those techniques. Currently, most such systems focus on monitoring while leaving diagnosis to applications. Table 1.1 classifies the well-known network monitoring systems according to their application environment (ISP-oriented or end-user oriented) and measurement techniques (active measurement or passive measurement). Four representative monitoring systems are NetScope [47], IPMON [49], NIMI [94], and Scriptroute, each having its own distinguishing characteristics. NetScope was developed by AT&T. It uses both active measurement techniques like ping and traceroute, and passive measurement techniques like SNMP and Netflow to monitor and diagnose AT&T's backbone network. IPMON, used in the Sprint backbone network, is a representative system that only uses passive measurements. IPMON mainly uses DAG (Data Acquisition and Generation) cards [42] to collect router interface packet traces. Packet traces are useful for packet-level history reconstruction to identify performance problems at small time granularities. NIMI is one of the earliest end-user oriented monitoring systems. It provides a measurement infrastructure for end users to install and deploy measurement tools and to collect measurement results. Since NIMI is an open platform, it uses many mechanisms to protect privacy and to quarantine the impact of buggy code. Scriptroute is a more recently proposed monitoring system. It provides both a distributed measurement infrastructure for end users to conduct measurements, and a script language to help develop measurement techniques. Scriptroute employs many security features in both its architectural design and its script language to address security and resource abuse that can result from active measurements. Compared with these monitoring systems, the TAMI system that is described in this dissertation proposes two new monitoring functionalities: measurement-scheduling and topology awareness.

There are far fewer diagnostic systems than monitoring systems. The representative diagnostic systems are AS-level route-event diagnostic systems [48] and the Planetseer system [119]. AS-level route-event diagnostic systems use BGP update messages to identify which AS triggers a BGP event that affects network traffic. Such diagnostic systems can only be used by ISPs since they require internal BGP data. Planetseer proposes a framework to diagnose IP-level route events that affect web client transmission performance. It uses the web-proxy service provided by CoDeen [117] to monitor web clients' TCP performance. For each performance problem, such as packet loss, Planetseer uses traceroute to identify its possible causes. It was used to demonstrate that route anomalies are a common cause for TCP performance problems. Comparatively, the TAMI system presented in this dissertation can also diagnose available-bandwidth related problems using the Pathneck technique.

1.6 Roadmap of The Dissertation

This dissertation is organized as follows. Chapter 2 shows how to accurately measure end-to-end available bandwidth. Specifically, it describes the design and implementation of the IGI/PTR tool, which implements two available-bandwidth measurement algorithms, estimating background traffic load (IGI) and packet transmission rate (PTR), respectively. It also describes the TCP PaSt algorithm, which demonstrates how a packet-train probing algorithm can be integrated into a real application. Chapter 3 explains how the PTR technique can be extended to develop the Pathneck tool, which uses carefully constructed packet trains to efficiently locate Internet bandwidth bottlenecks. Based on Pathneck, in Chapter 4, we present the results of several Internet-scale studies on bottleneck properties, analyzing Internet bottleneck location distribution, Internet end-user access bandwidth distribution, persistence of bottleneck locations, and relationships between bottlenecks and packet loss and queueing delay. Chapter 5 presents the source and sink tree structures that can efficiently capture network-edge information, including path-edge bandwidth. In the same chapter, we also introduce the RSIM metric that can quantify route similarities and infer path edges. Chapter 6 presents the BRoute system, which uses source and sink trees to solve the overhead problem of large-scale available-bandwidth monitoring. The last part of this dissertation (Chapter 7) describes the design and implementation of TAMI—a topology-aware measurement infrastructure. The conclusion and future work are presented in Chapter 8.

Chapter 2

End-to-End Available Bandwidth Measurement

In this chapter, we present the design, implementation, and evaluation of an effective and efficient end-to-end available bandwidth measurement tool. We first describe a single-hop gap model that captures the relationship between the throughput of competing traffic and the sending rate of a probing packet train on a single-hop network. Based on this model, we present the implementation details of an end-to-end available bandwidth measurement tool that implements the IGI and the PTR algorithms. We evaluate this tool from three aspects: (1) measurement accuracy and measurement overhead, (2) the impact of probing packet size and packet-train length on measurement accuracy, and (3) multi-hop effects on measurement accuracy. We also describe how to adapt IGI/PTR for heavily-loaded paths and high-speed paths. Finally, we describe an application of the PTR algorithm in TCP slow-start, where we demonstrate how to integrate the PTR packet-train probing algorithm into an application to improve its transmission performance by obtaining network performance information.

2.1 Overview of Active Measurement Techniques

Active measurement techniques estimate network metrics by sending probing packets and observing their transmission properties. For different performance metrics, the probing packets can be structured differently. A large number of different probing-packet structures have been used, such as a single packet (used by ping and traceroute), packet pair (used by bprobe [32], Spruce [113], and many others), packet tripplet (used by Tulip [80]), packet quartet [91], packet train (used by cprobe [32], Pathload [65] and many others), packet chirp (used by pathChirp [101]), and packet tailgating (used by nettimer [77] and STAB [100]). Besides the number of probing packets, these structures also use different methods to set packet gap values. Packet gap values can be either fixed (as in the packet trains in Pathload [65]), follow a Poisson distribution (as in the packet pairs in Spruce [113]) or an exponential distribution (as in the packet chirp in pathChirp [101]), or follow a TCP algorithm (as in Treno [83] and Sting [102]).

The packet pair is perhaps the most popular structure. Measurement techniques that use this structure generally send groups of back-to-back packets, i.e., packet pairs, to a destination which echos them back to the sender. The spacing between packet pairs is determined by the bottleneck link capacity and is preserved by links that have higher available bandwidth [63]. So by measuring the arriving time difference of the two probing packets, we can estimate path capacity. This is the key idea for most active bandwidth measurement tools. A packet train is an extension of the packet-pair structure, and it is often used for available bandwidth measurement. The difference is that packet trains capture not only path capacity but also traffic load information.

Packet tailgating is probably the most interesting structure. It is designed to avoid the dependence on ICMP error packets. It is first proposed by nettimer [77]. The idea is to let a small packet that has a large TTL follow a large packet that has a small TTL so that the small packet queues behind the large packet. After the large packet is dropped, the small packet can still reach the destination, preserving the delay or rate information of the large packet. When a train of packet-tailgating pairs is used, the sequence of small packets can be used to estimate the available bandwidth on the link where the large packets were dropped. That is the idea used by STAB [100] to obtain link-level available bandwidth and then locate bottleneck links.

2.2 Single-Hop Gap Model

The idea behind using active measurement techniques to estimate available bandwidth is to have the probing host send probing packets in quick succession and to measure how the packet gaps change (Figure 2.1). As the probing packets travel through the network, packets belonging to the competing traffic may be inserted between them, thus increasing the gaps. As a result, the gap values at the destination may be a function of the competing traffic rate, making it possible to estimate the amount of competing traffic. In practice, the way that the competing traffic affects the packet pair gap is much more complex than what is suggested above. In this section, we describe a simple model that captures more accurately the relationship between the gap value of a packet pair and the competing traffic load on a single-hop network.

2.2.1 Single-Hop Gap Model

The 3D graph in Figure 2.2 (the notations are defined in Table 2.1) shows the output gap value g_O as a function of the queue size Q and the competing traffic throughput B_C. This model assumes that the routers use fluid FIFO queueing and that all probing packets

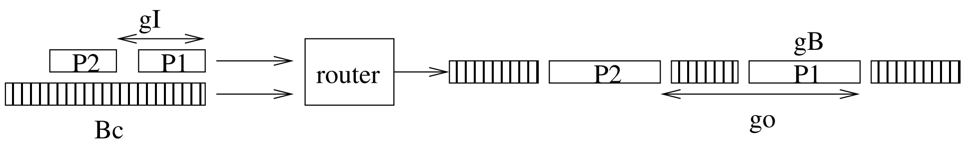


Figure 2.1: Interleaving of competing traffic and probing packets.
g_I is the initial gap. g_B is the probing packet length on the output link. g_O is the gap after interleaving with the competing traffic. B_C is the competing traffic throughput. Also refer to Table 2.1 for the symbols' definition.

Table 2.1: Definition of symbols

g_I	the **initial gap**, the time between the first bits of P1 and P2 when they enter the router; it includes P1's transmission delay (the time for a packet to be placed on a link) on the input link
g_B	the **bottleneck gap**, the transmission delay of the probing packet on the output link; it is also the gap value of two back-to-back probing packets on the bottleneck link
g_O	the **output gap**, the time between the first bits of P1 and P2 when they leave the router, i.e., on the bottleneck link
B_O	the bottleneck link capacity
B_C	the competing traffic throughput for the time interval between the arrival of packets P1 and P2
Q	the queue size when packet P1 arrives at the router
L	the probing packet length
r	$r = g_B/g_I$

have the same size. It also assumes that the competing traffic is constant in the interval between the arrival of packet P1 and P2; given that this interval is on the order of 1ms, this is a reasonable assumption. The model has two regions. As described below, the key difference between these two regions is whether or not the two packets P1 and P2 fall in the same queueing period. A *queueing period* is defined to be the time segment during which the queue is not empty, i.e., two consecutive queueing periods are separated by a time segment in which the queue is empty. For this reason, these two regions in the model are called the *Disjoint Queuing Region (DQR)* and the *Joint Queuing Region (JQR)*.

If the queue becomes empty after P1 leaves the router and before P2 arrives, then, since B_C is assumed to be constant in this (short) interval, P2 will find an empty queue. This means that the the output gap will be the initial gap minus the queueing delay for P1, i.e.,

$$g_O = g_I - Q/B_O. \tag{2.1}$$

Under what conditions will the queue be empty when P2 arrives? Before P2 arrives, the router needs to finish three tasks: processing the queue Q (Q/B_O), processing P1 (g_B), and

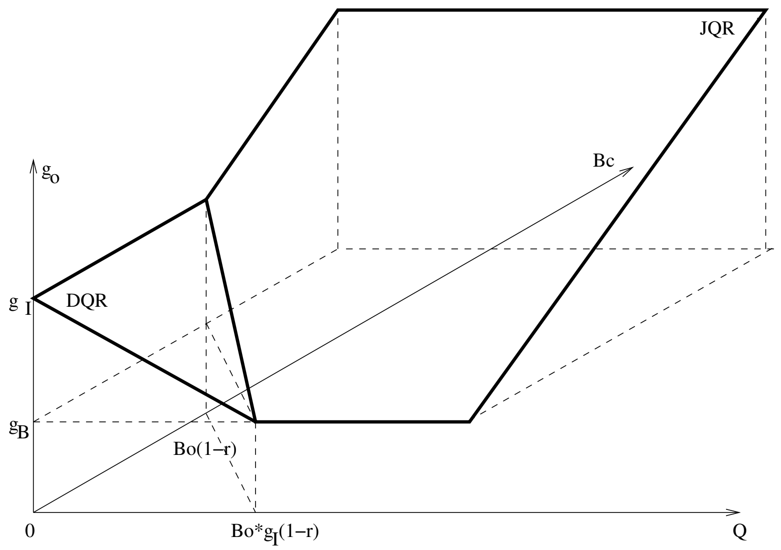


Figure 2.2: Single-hop gap model.

processing the competing traffic that arrives between the probing packets ($B_C \cdot g_I/B_O$). The router has g_I time to complete these three operations, so the condition is $Q/B_O + B_C \cdot g_I/B_O + g_B < g_I$, which corresponds to the triangular *DQR* region in Figure 2.2. In this region, the output gap g_O is independent of the competing traffic throughput B_C. The above equation (2.1) is called the *DQR* equation.

Under all the other conditions, i.e., in *JQR*, when P2 arrives at the router, the queue will not be empty. Since B_C is constant, this means that P1 and P2 are in the same queueing period. The output gap consists of two time segments: the time to process P1 (g_B), and the time to process the competing traffic that arrives between the two probing packets ($B_C \cdot g_I/B_O$). Therefore in this region, the output gap will be

$$g_O = g_B + B_C \cdot g_I/B_O. \tag{2.2}$$

That is, in this region, the output gap g_O increases linearly with the competing traffic throughput B_C. Equation (2.2) is referred to as the *JQR* equation.

This model clearly identifies the challenge in using packet pairs for estimating the competing traffic throughput. If the packet pair happens to operate in the *DQR* region of the bottleneck router, the output gap will bear no relationship with the competing traffic, and using the *JQR* equation (since the user does not know which region applies) will yield an incorrect result. Furthermore, the estimate obtained using a single packet pair will only provide the average competing traffic over g_I, which is a very short period. Since the competing traffic is likely to fluctuate, one in general will want to average the results of multiple samples, corresponding to independent packet pairs. This of course increases the chance that some of the samples will fall in the *DQR* region.

2.2.2 IGI and PTR Formulas

Equation (2.2) shows that in the *JQR* region we can estimate the competing traffic throughput B_C based on the initial gap g_I, the output gap g_O, and the bottleneck gap g_B. However, the single-hop gap model assumes that the competing traffic is a smooth packet stream. In practice, the competing traffic flow will be bursty and a single pair of probing packets will not capture the *average* throughput of the competing traffic. To deal with this problem, people use a packet train [92, 45], i.e., a longer sequence of evenly spaced packets.

The conclusions from the single-hop gap model do not directly apply to a packet train. The main problem is that the "pairs" that make up a packet train are not independent. For example, if one packet pair in the train captures a burst of packets from the competing flow, it is highly likely that adjacent pairs will not see any competing traffic and will thus see a decrease in their packet gap. Intuitively, if we want to estimate the amount of competing traffic, we should focus on the *increased* gaps in a probing packet train since they capture the competing traffic, while decreased gaps saw little or no competing traffic. Note that this observation only applies when the probing packet train operates in the *JQR* region.

More precisely, assume a probing train in which M probing gaps are increased, K are unchanged, and N are decreased. If we now apply equation (2.2) to all the increased gaps, we get the following estimate for the competing traffic load:

$$\frac{B_O \sum_{i=1}^{M} (g_i^+ - g_B)}{\sum_{i=1}^{M} g_i^+ + \sum_{i=1}^{K} g_i^= + \sum_{i=1}^{N} g_i^-}. \tag{2.3}$$

Here, the gap values $G^+ = \{g_i^+ | i = 1, ..., M\}$, $G^= = \{g_i^= | i = 1, ..., K\}$, and $G^- = \{g_i^- | i = 1, ..., N\}$ denote the gaps that are increased, unchanged, and decreased, respectively. In this formula, $B_O \sum_{i=1}^{M} (g_i^+ - g_B)$ is the amount of competing traffic that arrive at router R1 during the probing period. Ideally, $\sum_{i=1}^{M} g_i^+ + \sum_{i=1}^{K} g_i^= + \sum_{i=1}^{N} g_i^-$ is the total probing time. In practice, we exclude gap values that involve lost or reordered packets, so in such cases, the denominator may be smaller than the total probing time. This method of calculating competing traffic load will be used by the *IGI (Initial Gap Increasing)* algorithm in Section 2.3, and it is called the *IGI* formula.

A number methods have been proposed to estimate the available bandwidth along a network path [32, 65]. Using the same notation as used above, the equation they use is

$$\frac{(M + K + N)L}{\sum_{i=1}^{M} g_i^+ + \sum_{i=1}^{K} g_i^= + \sum_{i=1}^{N} g_i^-}. \tag{2.4}$$

Here L is the probing packet size. This formula represents the average transmission rate of the packet train, measured at the destination. We will also use this formula in the *PTR (Packet Transmission Rate)* algorithm described in Section 2.3, and it is called the *PTR* formula.

2.3 IGI and PTR Algorithms

The gap model shows that the *IGI* formula only applies in the *JQR* region, and we will show below that the *PTR* formula is also only valid under similar conditions. Note that the main parameter that is under our control in the single-hop gap model is the initial gap value g_I. It has a large impact on the size of the *DQR* region, and thus on the region in which the packet train operates. Therefore, the key to an accurate available bandwidth measurement algorithm is to find a g_I value so that the probing packet train operates in the *JQR* region. In this section, we first study the role of g_I more carefully, we then describe how to combine the insights on g_I and the *IGI* and *PTR* formulas to develop the two available bandwidth estimation algorithms.

2.3.1 Impact of Initial Gap

According to the single-hop gap model, if we are in the *JQR* region, the output gap of a packet pair or train can give us an estimate of the competing traffic on the bottleneck link. However, in the *DQR* region, output gap is independent of the competing traffic. We also see that increasing the initial gap will increase the *DQR* area. This argues for using small initial gaps. In fact, if $g_I \leq g_B$, i.e., if the initial gap is smaller than the probing packet transmission delay on the bottleneck link, the *DQR* area does not even exist. However, with small initial gaps, such as $g_I \leq g_B$, we are flooding the bottleneck link, which may cause packet losses and disrupt traffic.

We use the following experiment to better understand the impact of the initial probing gap on the accuracy of the *IGI* and *PTR* formulas. We send an Iperf TCP competing traffic flow of 3.6Mbps over a 10Mbps bottleneck link. We then probe the network using a set of packet trains; the packet train length is 256 and the probing packet size is 750Byte. We start with an initial probing gap of 0.022ms, which is the smallest gap that we can get on the testbed, and gradually increase the initial gap. Figure 2.3 shows the average gap difference (averaged output gap minus the averaged initial gap), the competing traffic throughput estimated using the *IGI* formula, and the available bandwidth estimated using the *PTR* formula.

We see that for small initial gaps (smaller than $g_B = 0.6ms$, which is the transmission time on the bottleneck link), we are flooding the network and the measurements underestimate the competing traffic throughput. Note that for minimal initial gaps, the *PTR* formula is similar to the formula used to estimate the bottleneck link capacity by tools such as bprobe [32], and in fact, the *PTR* estimate for small initial gaps is close to 10Mbps, which is the bottleneck link capacity. When the initial gap reaches g_B, the *DQR* effect starts to appear. Note that, unless the network is idle, the probing packet train is still flooding the bottleneck link. So far, the average output gap at the destination is larger than the initial gap. When further increasing the initial probing gap, at some point (0.84ms in the figure), the output gap equals the initial gap; we call this the *turning point*. At this point, the prob-

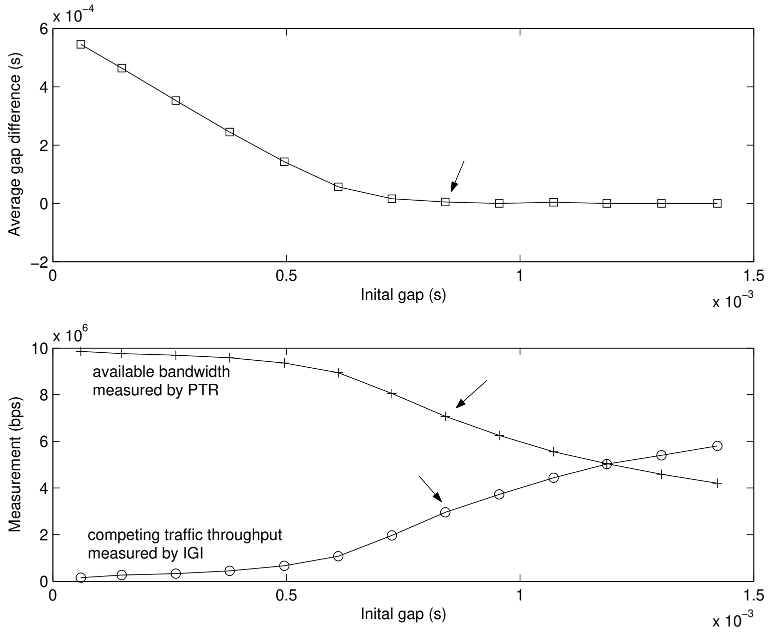


Figure 2.3: Impact of the initial gap on available bandwidth measurements
The arrows point out the measurements at the turning point, the smallest initial gap where the average output gap equals the average initial gap.

ing packets interleave nicely with the competing traffic, and the average rate of the packet train equals the available bandwidth on the bottleneck link. In this experiment, the *IGI* estimate for the competing traffic at the turning point is 3.2Mbps and the *PTR* estimate for the available bandwidth is 7.1Mbps; both match the actual competing traffic (3.6Mbps) quite well. As the initial probing gap continues increasing, the output gap remains equal to the initial gap since all the packets on average experience the same delay.

We believe that the point where the average output gap equals to the initial gap, i.e., the turning point shown in Figure 2.3, is the correct point to measure the available bandwidth. The turning point corresponds to the smallest initial gap value with which the probing packet train is not flooding the bottleneck link. With respect to the single-hop gap model in Figure 2.2 on which the *IGI* formula is based, this initial gap will result in a packet train that keeps the queue as full as possible without overflowing it; the model shows that this puts us in the *JQR* region. With respect to the *PTR* formula, the initial gap at the turning point corresponds to the packet transmission rate where the packet trains consume all the available bandwidth without significant interference with the competing traffic. In other words, the packet train behaves like an aggressive, but well behaved (i.e., congestion controlled) application flow, so its rate is a good estimate of the available bandwidth.

2.3.2 IGI and PTR Algorithms

The *Initial Gap Increasing (IGI)* and *Packet Transmission Rate (PTR)* algorithms discussed below are based on packet trains that operate at the turning point. That is, they send a sequence of packet trains with increasing initial gap from the source to the destination host. They monitor the difference between the average source (initial) and destination (output) gap and they terminate when it becomes zero. At that point, the packet train is operating at the turning point. We then use the *IGI* and *PTR* formulas to compute the final measurement.

The pseudocode for the *IGI* algorithm is shown in Figure 2.4. The available bandwidth is obtained by subtracting the estimated competing traffic throughput from an estimate of the bottleneck link capacity. The bottleneck link capacity can be measured using, for example, bprobe [32], nettimer [77], or pathrate [45]. Note that errors in the bottleneck link capacity measurement will affect the accuracy of the available bandwidth estimate, since the bottleneck link capacity B_O is used in the calculation of the bottleneck gap g_B, the competing traffic throughput c_bw, and the available bandwidth a_bw. However, the analysis of the above mentioned tools and our experience show that the bottleneck link capacity measurement is fairly accurate, so in the implementation of IGI/PTR, we do not consider this factor.

The *PTR* algorithm is almost identical to the *IGI* algorithm. The only difference is that the last three lines in Figure 2.4 need to be replaced by

$$ptr = packet\_size * 8 * (probe\_num - 1)/dst\_gap\_sum;$$

These formulas assume that there is no packet loss or packet reordering.

In both algorithms, to minimize the number of probing phases, the gap_step and $init\_gap$ need to be carefully selected. The first probing uses an $init\_gap$ that is as small as possible. This allows us to estimate the bottleneck link capacity and g_B. it then sets $gap\_step = g_B/8$, and $init\_gap = g_B/2$. Another key step in both algorithms is the automatic discovery of the turning point. This is done in the procedure GAP_EQUAL(). It tests whether the source and destination gaps are "equal", which is defined as

$$\frac{|src\_gap\_sum - dst\_gap\_sum|}{\max(src\_gap\_sum, dst\_gap\_sum)} < \delta.$$

In the experiments, δ is set to 0.1. These two steps are a key difference between *PTR* algorithm and other techniques based on Formula (2.4) since they allow us to quickly find a good initial gap. We evaluate how fast this algorithm converges in Sections 2.5.1 and 2.6.2.

Besides the initial gap, two other parameters also affect the accuracy of the *IGI* and *PTR* algorithms:

1. *Probing packet size*. Measurements using small probing packets are very sensitive to interference. The work in [45] also points out significant post-bottleneck effects for small packets. This argues for sending larger probing packets.

Algorithm *IGI*:
{
    /* initialization */
    $probe\_num = \text{PROBENUM}$; $packet\_size = \text{PACKETSIZE}$;
    $g_B = \text{GET\_GB}()$;
    $init\_gap = g_B/2$; $gap\_step = g_B/8$;
    $src\_gap\_sum = probe\_num * init\_gap$; $dst\_gap\_sum = 0$;

    /* look for probing gap value at the turning point */
    while $(!\text{GAP\_EQUAL}(dst\_gap\_sum, src\_gap\_sum))$ {
        $init\_gap$ += gap_step;
        $src\_gap\_sum = probe\_num * init\_gap$;
        $\text{SEND\_PROBING\_PACKETS}(probe\_num, packet\_size, init\_gap)$;
        $dst\_gap\_sum = \text{GET\_DST\_GAPS}()$;
    }
    /* compute the available bandwidth using IGI formula */
    $inc\_gap\_sum = \text{GET\_INCREASED\_GAPS}()$;
    $c\_bw = b\_bw * inc\_gap\_sum/dst\_gap\_sum$;
    $a\_bw = b\_bw - c\_bw$;
}

Figure 2.4: IGI algorithm

SEND_PROBING_PACKETS() sends out $probe\_num$ $packet\_size$ probing packets with the initial gap set to $init\_gap$; GET_DST_GAPS() gets the destination (output) gap values and adds them; GET_INCREASED_GAPS() returns the sum of the initial gaps that are larger than the bottleneck gap; c_bw, b_bw, and a_bw denote the competing traffic throughput, the bottleneck link capacity, and the available bandwidth, respectively.

2. *The number of probing packets*. It is well known that the Internet traffic is bursty, so a short snapshot cannot capture the average traffic load. That argues for sending a fairly large number of probing packets. However sending too many packets can cause queue overflow and packet losses, increase the load on the network, and lengthen the time it takes to get an estimate.

Through experiments we found that that the quality of the estimates is not very sensitive to the probing packet size and the number of packets, and that there is a fairly large range of good values for these two parameters. For example, a 700-byte packet size and 60-packet train work well on the Internet. We discuss the sensitivity to these two parameters in more detail in Section 2.6.

Finally, we would like to point out the two limitations of the IGI/PTR design. One is that IGI/PTR requires access to both the source and the destination of a path. That limits its applicability since regular end users often only have local access. This problem is partially solved by the Pathneck technique that is presented in the next chapter. The other is that the performance model of IGI/PTR presented in Section 2.2 assumes that routers use a FIFO algorithm to schedule packets. This model may not apply to wireless networks and broadband networks where packets are often scheduled using non-FIFO algorithms.

2.4 Evaluation Methodology

The evaluation includes three parts:

- In Section 2.5, we compare the performance of *IGI*, *PTR*, and Pathload, focusing on the measurement accuracy and the convergence time.
- In Section 2.6, we analyze how the probing packet size and the number of probing packets (packet train length) affect the measurement accuracy of *IGI* and *PTR*.
- In Section 2.7, we study the performance of *IGI* and *PTR* on a network path where the tight link is not the same as the bottleneck link. We also look into a related issue about the impact of gap timing errors.

The first two parts are based on Internet measurements; while the last part is based on ns2 simulations, because we need to carefully control the competing traffic load in the network.

The Internet measurements are collected from the 13 Internet paths listed in Table 2.2. In this table, CORNELL, CMU[1-3], NYU, ETH, NCTU are machines in Cornell University, Carnegie Mellon University, New York University, ETH Zurich (Switzerland), and National Chiao Tung University (Taiwan), respectively. MA, SLC[1-2], SV, FC, SWEDEN, NL are machines on commercial networks, and they are located in Massachusetts, Silicon Valley, Foster City, Sweden, and The Netherlands, respectively. For each path in Table 2.2, the first site is the sender, and the second site is the receiver. The capacities in the third column denote the bottleneck link capacities, which we will also refer to as the *path capacity*. The path capacities are measured using bprobe [32], and the RTTs are measured using ping. The path capacities shown in the table are obtained by "rounding" the measured values to the nearest well-known physical link capacity.

To evaluate the accuracy of the different probing algorithms on the Internet, we interleave probing experiments with large application data transfers that show how much bandwidth is actually available and usable on the network path. However, it is sometimes hard to determine the actual available bandwidth on an Internet path. In practice, most applications, especially bulk data transfer applications, use TCP. Unfortunately, for high bandwidth paths, TCP is often not able to fully utilize the available bandwidth. In most cases the reason was simply that TCP end-to-end flow control is limiting the throughput, and without root permission, we can not increase the size of socket buffers. On other paths

Table 2.2: Internet Paths

ID	*Path* (sender→receiver)	*Capacity* (Mbps)	*RTT (std dev)* (ms)
1	CORNELL→ MA	1.5	27.59 (2.82)
2	SLC1→ CMU2	10	59.65 (0.58)
3	NWU→ CMU1	100	10.29 (0.94)
4	CORNELL→ CMU1	10	13.43 (0.15)
5	ETH→ SWEDEN	10	76.04 (0.35)
6	SLC2→ CMU1	100	83.21 (0.41)
7	SLC2→ NYU	100	53.52 (0.36)
8	ETH→ CMU1	100	125.00 (0.30)
9	ETH→ NL	100	28.21 (0.21)
10	SV→ NYU	2.5	78.29 (0.21)
11	SLC1→ FC	4.5	43.39 (10.10)
12	SLC2→ FC	4.5	80.65 (23.60)
13	NCTU→ CMU3	100	265.54 (0.41)

we observe a significant amount of packet reordering or unexplained packet losses, both of which can have a significant impact on TCP performance.

For the above reasons, we use a mixture of techniques to measure the "true" available bandwidth. When possible, we use a single TCP flow. When small window sizes prevent us from filling the pipe, we use a number of parallel TCP flows. The number of flows is selected on a per path basis. A typical example of how the end-to-end throughput increases with the number of flows is shown in Figure 2.5. The throughput increases initially and then flattens out. Typically 10 or at most 20 flows are sufficient to fill the available bandwidth pipe.

Note that this approach provides only a rough idea of the accuracy of the probing techniques. A first problem is that the probing and the data transfers cannot be run at the same time, so they see different traffic conditions, and we should expect slightly different results. Moreover, because of the bandwidth sharing characteristics of TCP, a single TCP flow is not equivalent with multiple parallel TCP flows. On the other hand, our approach does model the way applications will typically use probing tools, so our approach captures the accuracy that applications will perceive. Our experience with tools such as Remos [43] shows that applications in general only require rough estimates of path properties.

The implementation of the *IGI* and *PTR* algorithms needs accurate timestamp measurement. As a result, we would expect the best results with kernel support, such as libpcap [9]. However, for most of the end hosts we use for our experiments, we only have guest accounts, so all the Internet measurements are collected with a user-level implementation. The probing packets are UDP packets, and timestamps are measured when the client or

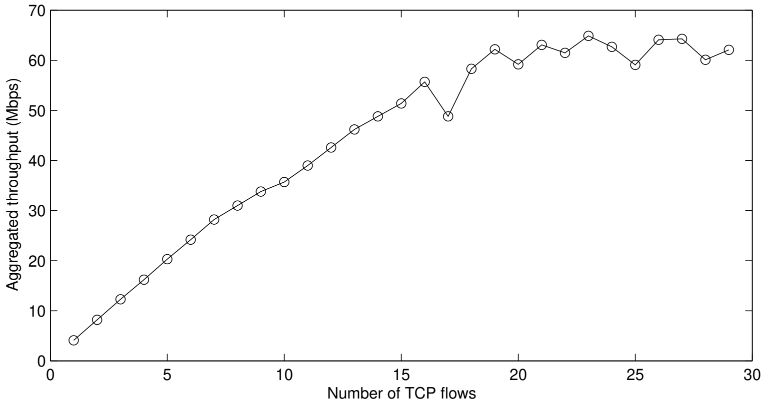


Figure 2.5: Throughput of parallel TCP flows on the path ETH → NWU

server applications sends or receives the UDP packets.

2.5 Measurement Accuracy and Overhead

In this section, we analyze the performance of *IGI* and *PTR* algorithms using experiments on the 13 Internet paths listed in Table 2.2. We also compare their performance with that of Pathload. The experiments are conducted as follows. For each Internet path, we measure the available bandwidth using the following three methods:

1. *IGI* and *PTR*: we use both *IGI* and *PTR* algorithms to estimate the available bandwidth. The probing packet size is set to 700Byte, and the probing packet number is 60. We discuss why we choose these two values in Section 2.6.

2. *Pathload*: The resolution parameter is set to 2Mbps. The Pathload implementation[1] returns a measurement interval that should contain the actual available bandwidth. We use the center of the interval in our analysis.

3. *Bulk data transfer*: We use one or more Iperf TCP flows to probe for the actual available bandwidth. The transmission time is 20 seconds, and the TCP window size at both ends is set to 128KB, which is supported on all machines we have access to.

We separate the above three measurements by a 5-second sleep period to avoid interference between the measurements. We separate experiments by 10 minutes of idle time. The measurements run for anywhere from 6 to 40 hours.

[1]http://www.cis.udel.edu/∼dovrolis/pathload_1.0.2.tar.gz.

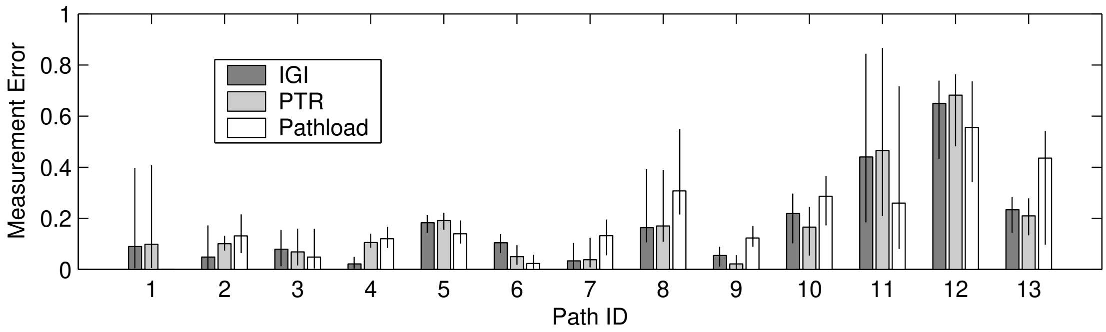


Figure 2.6: Available bandwidth measurement error from *IGI*, *PTR*, and Pathload
Each bar shows the median value, and the line on each bar shows the 5% and 95% percentile values.

2.5.1 Measurement Accuracy

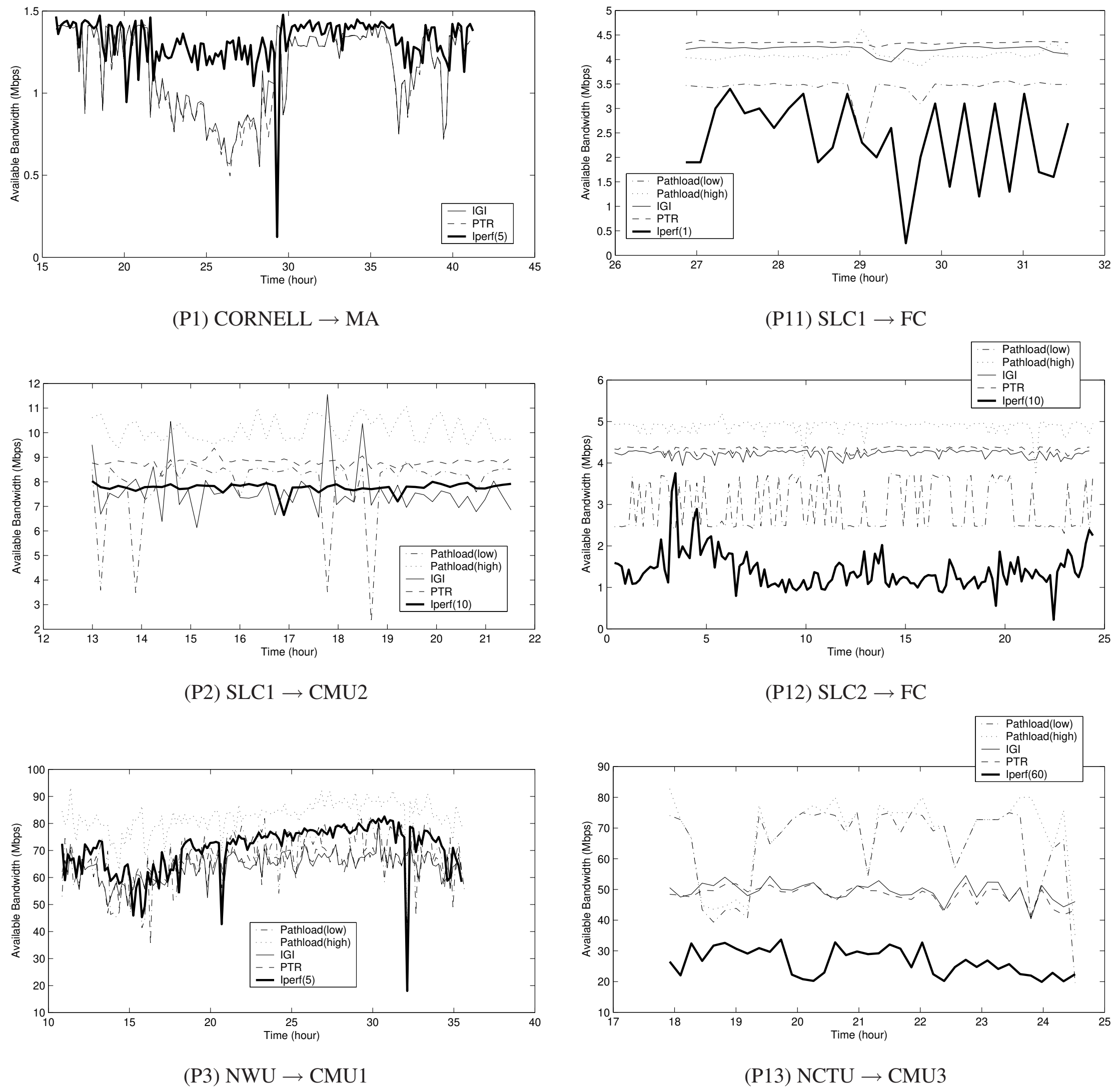
We use the metric relative measurement error to evaluate available bandwidth measurement accuracy. It is define as:

$$relative\_error = \frac{|a\_bw_X - throughput_{TCP}|}{B_O}$$

Here a_bw_X can be a_bw_{IGI}, a_bw_{PTR}, and $a\_bw_{Pathload}$, i.e., the available bandwidth estimates generated by the different techniques; $throughput_{TCP}$ is the bulk data transmission rate, and B_O is the bottleneck link capacity.

Figure 2.6 shows the relative measurement error of *IGI*, *PTR*, and Pathload. The Pathload code used in this experiment does not apply to paths with available bandwidth below 1.5Mbps (it returns the interval [0, link capacity]), so we have no Pathload measurements for Path 1. The measurement errors for path 1-10 are below 30%, and in most cases the error is less than 20%. That is, the estimates produced by the *IGI/PTR* and the Pathload algorithms match the TCP performance fairly well. For paths 11-13, the relative measurement error is much higher. Without the information from the service providers, it is hard to tell what causes the higher errors. Because all three methods have low accuracy, we hypothesize that TCP has difficulty using the available bandwidth due to bad path properties. For example, Table 2.2 shows that the RTT variances for paths 11 and 12 are large compared with those for the other paths. This may be caused by route flaps, which may negatively influence TCP's performance.

Figure 2.7 includes more detailed comparison of the bandwidth estimates for six of the paths. We pick three "good" paths with different path properties (paths 1-3, see Table 2.2) and all three of the bad paths (path 11-13). For paths P1, P2, and P3, the graphs confirm that all three techniques provide good estimates of the available bandwidth, as measured by Iperf. Which technique is more accurate depends on the path. For example,



(P1) CORNELL → MA

(P11) SLC1 → FC

(P2) SLC1 → CMU2

(P12) SLC2 → FC

(P3) NWU → CMU1

(P13) NCTU → CMU3

Figure 2.7: Available bandwidth measurements and the TCP performance.
The number in the brackets of Iperf is the number of Iperf TCP flows used. The x-axis is the clock time value, a number larger than 24 is the time next day.

IGI seems more accurate for P2 and Pathload for P3. One notable exception is the period from hour 22 to hour 28 for P1, where both *IGI* and *PTR* appear to underestimate the available bandwidth. For this path, the bottleneck link is a DSL line, which is in general idle, as is shown by the high available bandwidth. During the 22-28 hour interval, the DSL line is used. Since only one or a few TCP connections are active, they consume only part of the available bandwidth. The bulk data transfer, however, uses five parallel Iperf flows and appears to be grabbing bandwidth from the other flows. This illustrates that the

Table 2.3: Measurement Time

Path ID	*IGI/PTR* (s) (5%, **median**, 95%)	Pathload (s) (5%, **median**, 95%)	Ratio($\frac{Pathload}{IGI/PTR}$) median
1	(1.60, **2.05**, 6.27)	(14.98, **30.56**, 31.03)	13.22
2	(0.58, **0.73**, 1.56)	(13.67, **15.37**, 31.81)	20.86
3	(0.11, **0.11**, 0.18)	(7.55, **13.17**, 14.91)	99.78
4	(0.49, **0.52**, 0.52)	(11.78, **12.26**, 12.76)	23.48
5	(0.78, **0.80**, 0.83)	(15.58, **15.86**, 16.55)	19.75
6	(0.62, **0.80**, 1.20)	(49.07, **56.18**, 62.24)	70.08
7	(0.51, **0.51**, 0.67)	(14.01, **22.40**, 28.51)	45.94
8	(1.01, **1.02**, 1.27)	(27.57, **31.51**, 47.62)	27.80
9	(0.24, **0.30**, 0.30)	(15.35, **16.14**, 27.66)	65.81
10	(1.27, **1.27**, 1.50)	(20.95, **21.04**, 21.77)	16.50
11	(1.03, **1.10**, 2.03)	(19.97, **25.78**, 38.52)	23.45
12	(2.17, **2.32**, 3.60)	(19.24, **21.54**, 42.00)	9.20
13	(1.10, **1.11**, 1.13)	(12.24, **12.76**, 47.22)	11.24
Geometric Mean			26.39

“available bandwidth” is not necessarily well-defined and depends on how aggressive the sender is. Note that this is a somewhat atypical path: on most Internet paths, individual senders will not be able to affect the bandwidth sharing as easily.

For the three paths where the relative measurement error is high, we see the available bandwidth estimates produced by all three methods are much higher than the bandwidth measured using Iperf. As we already suggested above, this probably means that TCP, as used by Iperf, is not able to function well because of problems such as window size [103], loss rate, and variable round trip time [89]. Note that the three bandwidth estimation techniques provide fairly similar results, except for path P13, where the Pathload estimates are extremely high.

In terms the difference between the estimate from the *IGI* algorithm and that from the *PTR* algorithm, for most paths, they are within 10% of each other. One exception is for path P2 (Figure 2.7(P2)), where the *IGI* estimates change over a wider range than those provided by the *PTR* method. We believe this is caused by traffic on links other than the bottleneck link. As we will discuss in Section 2.7, the *IGI* method is more sensitive to competing traffic from non-bottleneck links than the *PTR* method.

2.5.2 Convergence Times

So far our measurements have shown that the three algorithms have similar accuracy in terms of predicting available bandwidth. However, the *IGI* and *PTR* methods, which have the same measurement time, are much faster than Pathload, as is shown in Table 2.3. In this table, we show the percentile values of the measurement times at 5%, 50% (median), and 95% for each path for both the *IGI/PTR* and the Pathload techniques. We see that the *IGI* and *PTR* methods typically take about 1-2 seconds while Pathload takes at least 12 seconds [65]. We also compute the ratio between Pathload and *IGI/PTR* for each round of measurements; the median values are listed in the last column of the table. The geometric mean [67] of all ratios shows that the *IGI/PTR* method is on average more than 20 times faster than Pathload for the 13 paths used in this study.

The long measurement time for Pathload is due to its convergence algorithm. Pathload monitors changes in the one-way delay of the probing packets in order to determine the relationship between probing speed and available bandwidth. This can be difficult if probing packets experience different levels of congestion. This can slow down the convergence process and can result in long probing times as shown in Table 2.3. In contrast, the convergence of *IGI/PTR* is determined directly by the packet train dispersion at the source and destination. Moreover, the *IGI* and *PTR* algorithms use the bottleneck link capacity, which is estimated using the same probing procedure, to adjust *init_gap* and *gap_step* so as to optimize convergence.

2.6 Impact of Probing Configurations

The *IGI* and *PTR* algorithms select the appropriate initial gap for the probing trains by searching for the turning point, as described in Section 2.3. In this section, we use Internet experiments to study the impact of the other two packet train parameters—the probing packet size and the number of probing packets (packet train length).

2.6.1 Probing Packet Size

To study the impact of the probing packet size on the measurement accuracy of the *IGI* and *PTR* algorithms, we conduct experiments on two Internet paths, using probing packet sizes ranging from 100Byte to 1400Byte. We repeat each individual measurement 20 times. The entire experiment takes about one hour. On the assumption that Internet path properties do not change much on the scale of hours [120], we would expect all measurements to have very similar result.

The first Internet path we use is from NWU to CMU. It has a path capacity of 100Mbps. The measurement results are shown in Figure 2.8(a1) and Figure 2.8(b1). Figure 2.8(a1) shows how the available bandwidth measurements change with the probing packet size. The available bandwidth measured using a TCP bulk data transfer (based on the method discussed in Section 2.5) is 64Mbps. The packet sizes that result in the closest estimates are 500Byte and 700Byte. For smaller packet sizes, both methods underestimate the available bandwidth by a significant margin. For larger probing packet sizes, the two methods overestimate the available bandwidth by a much smaller amount.

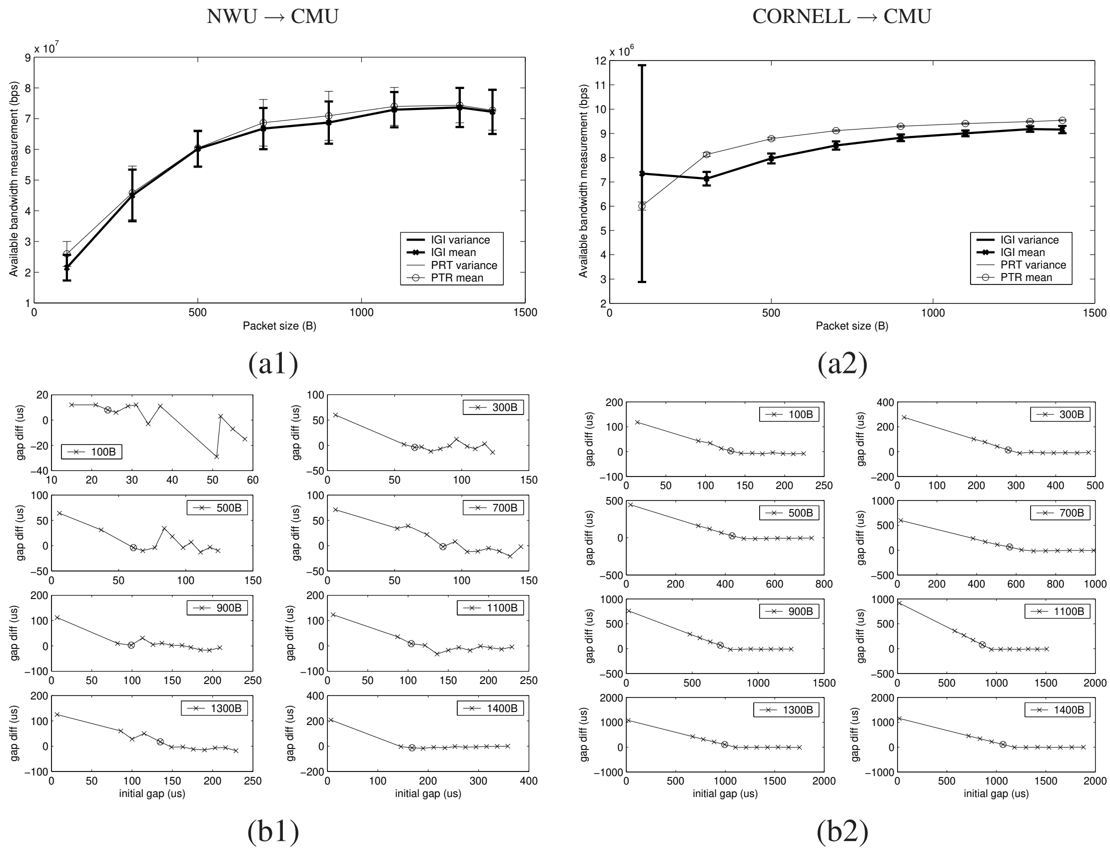


Figure 2.8: Impact of probing packet sizes.
Graphs (a1) and (a2) show the final available bandwidth estimates. Graphs (b1) and (b2) show the gap convergence for individual measurements: the x-axis is the initial source gap, and the y-axis is the gap difference, i.e., the destination (output) gap value minus the source (input) gap value; the points marked with circles are the turning points where the final estimates are computed.

There are at least two reasons why small probing packet sizes can result in high errors in the available bandwidth estimation. First, as illustrated in Figure 2.8(b1), at the turning point the gap value is proportional to the packet size. This means that with small packet sizes, we will have small gaps, especially if the available bandwidth is high, as is the case for the NWU to CMU path. The resulting probing train is more sensitive to the burstiness of the competing traffic. The graph for 100Byte probing packets in Figure 2.8(b1) confirms this: the gap difference does not converge as nicely as it does with larger probing packets. The second reason is that the small gap values that occur with small probing packets are harder to measure accurately, so measurement errors can affect the result significantly. Gap values on the order of 10μs are hard to generate and measure accurately, especially for user-level applications.

It is less clear why with larger probing packets, the available bandwidth estimates further increase and in fact exceed the measured bulk throughput. We conjecture that this is a result of the aggressiveness of the probing packet train flow. Probing flows with larger packets are more aggressive than probing flows with smaller packets, so they “observe” a higher available bandwidth. The packet size distribution on Internet has clusters around 40Byte, 500Byte and 1500Byte [70], so a flow with only 1200Byte or 1500Byte packets, for example, is more aggressive than average. A TCP bulk data transfer is likely to use mostly maximum sized packets (1500B in this case), but its dynamic congestion control behavior reduces how much bandwidth it can use.

The second experiment is on the path from CORNELL to CMU. The results are summarized in Figures 2.8(a2) and 2.8(b2). The link capacity of the bottleneck link is only 10Mbps, as opposed to 100Mbps for the NWU to CMU path. As a result, the available bandwidth is significantly lower. The results confirm the main results of the measurements for the NWU to CMU path. First, the available bandwidth estimates increase with the packet size. Second, since the available bandwidth is much lower, we are seeing fairly smooth convergence of the gap difference, even for small probing packet sizes (Figure 2.8(b2)). Finally, even though we observe nice convergence, the burstiness of the competing traffic does affect the probes with small packets more than the probes with larger packets. For the *IGI* algorithm, the results with 100Byte probing packet are suspicious and have a large variance. Because the *IGI* algorithm uses the changes in individual gap values instead of the average packet train rate (as used by *PTR*), it is more sensitive to small changes in gap values, for example as a result of bursty traffic or traffic on non-bottleneck links. We discuss this point in more detail in Section 2.7.

Our conclusion is that in general, average-sized probing packets of about 500Byte to 700Byte are likely to yield the most representative available bandwidth estimate. Smaller packet sizes may underestimate the available bandwidth and may be more sensitive to measurement errors, while larger probing packet sizes can overpredict the available bandwidth.

2.6.2 Packet Train Length

The packet train length has a large impact on the cost of the *PTR* and *IGI* algorithms, since it affects both the number of packets that are sent (i.e., the load placed on the network) and the probing time (i.e., the latency associated with the probing operation). Another important parameter, the number of phases needed to converge on the best initial gap value (the turning point), is tied very closely to the packet train length. Intuitively, shorter packet trains provide less accurate information, so more phases may be needed to converge on the turning point. For this reason, we will study the packet train length and the number of phases in the *IGI/PTR* algorithm together.

In Section 2.3, we mentioned that trains of 60 packets work well. In this section we experimentally evaluate how much we can reduce this number without a significant

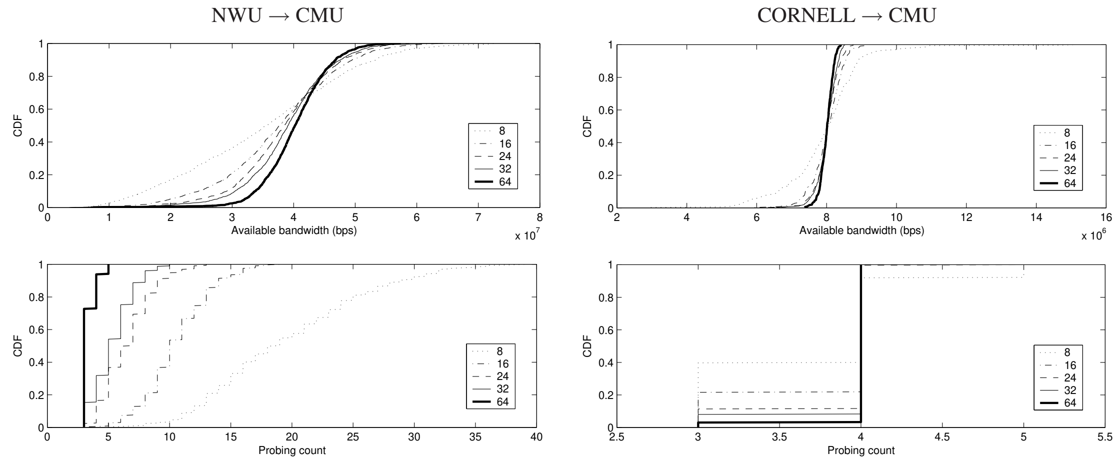


Figure 2.9: Performance with packet trains of different lengths

loss in accuracy. We conduct experiments over the same two Internet paths as in the previous section, i.e., NWU to CMU and CORNELL to CMU. For each path, we use packet trains of different lengths to estimate the available bandwidth. The measurements take about two hours. Since the available bandwidth over the Internet is fairly stable [120], we do not expect the available bandwidth to change significantly during the 2-hour period. The measurements with different train lengths are also interleaved to further reduce any possible bias towards a specific train length.

Figure 2.9 shows the cumulative distribution function (CDF) of the estimated available bandwidth using *IGI* (top), and the number of probing phases needed to converge on the turning point (bottom). The distributions for the *PTR* measurement are similar and are not included here. Each graph has five curves, corresponding to five different packet train lengths: 8, 16, 24, 32, and 64. First, we observe that shorter packet trains need more phases to converge, which we had already conjectured earlier. The measurements also show, again not surprisingly, that shorter packet trains result in a wider range of available bandwidth estimates, as shown by a CDF that is more spread out. The reason is that the competing traffic (and thus the available bandwidth) is bursty, and since a shorter packet train corresponds to a shorter sampling interval, we are seeing a wider range of estimates. Note however that as the packet train length increases, the impact of the packet train length on the distribution of the bandwidth estimates becomes smaller, i.e., the estimates converge on a specific value.

It is interesting to compare the results for the two paths. For the NWU to CMU path, changing the packet train length has a fairly significant impact on the distributions for both the available bandwidth and the phase count. In other words, increasing the packet train length helps in providing a more predictable available bandwidth estimate. Using longer

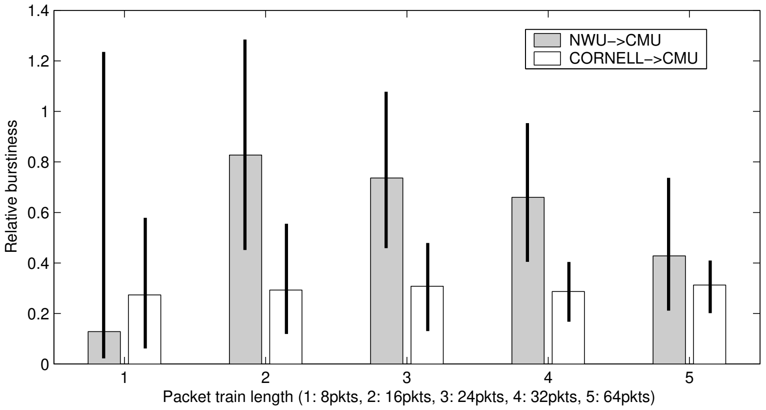


Figure 2.10: Relative burstiness measurements based on the gap values.
Each bar shows the median value, and the lines on each bar show the 5% and 95% percentile values.

trains is also "rewarded" with a reduction in the the number of probing phases. In contrast, for the CORNELL to CMU path the CDF functions for both the available bandwidth and phase count are fairly similar for train lengths of 16 packets or more. The reason is that the competing traffic on this path is not as bursty as that on the NWU to CMU path.

The difference between the two paths raises the question of what packet train length we should use for available bandwidth estimation. Clearly, the most appropriate train length depends on the path. For the NWU to CMU path, we probably would want to use a fairly large value (32 or 64 packets), while for the CORNELL to CMU path, a train length of 16 packets is sufficient. Since the difference between the paths appears to be caused by the burstiness of the traffic, we decide to use the changes in the packet gaps to characterize the burstiness of the competing traffic. Specifically, we define the *relative burstiness* as:

$$relative\_burstiness = \frac{\frac{1}{N-1}\sum_2^N |g_i - g_{i-1}|}{\frac{1}{N}\sum_1^N g_i},$$

where $g_i (1 \leq i \leq N)$ are the N gap measurements of a probing train.

Figure 2.10 shows the relative burstiness of the *IGI* measurements at the turning point for the two paths and for the different packet train lengths. We record the detailed gap values at the turning point for 65 measurements (around 20% of the measurements collected). The relative burstiness for the path from NWU to CMU is significantly higher than that for the path from CORNELL to CMU. Interesting enough, the results for 8-packet probing trains do not follow this trend. We suspect that eight packets is simply not long enough to get a reliable measurement (note the wide spread).

These results suggest that we can reduce the cost of probing by dynamically adjusting the length of the packet train. For example, we could use a packet train of 32 packets for the first few phases and use the burstiness results of those phases to adjust the length of later packet trains. We decide not to do this because, as the results in Table 2.3 show, the *IGI/PTR* algorithm is already quite fast. The distribution of the probing phase counts shows that 80% of the measurements only need 4–6 phases to converge to the turning point, so the corresponding probing time is around 4–6 round trip times. Dynamically adjusting the packet train length is thus not likely to have a large impact on the probing time. Of course, we could make the burstiness information available to users so they can know how variable the available bandwidth is likely to be for short data transfers.

2.7 Multi-hop Effects

The *IGI* and *PTR* algorithms are based on the gap model presented in Section 2.2. It is derived for a simple single-hop network, or more generally, for a network in which the bottleneck link is the tight link and the effect of all other links can be ignored. In this section we use simulations to study more general multi-hop networks. Specifically, we address two questions. First, how should we interpret the model if the tight link is not the bottleneck link, and what are the implications for the *IGI* and *PTR* method? Second, how does the competing traffic on links other than the tight link affect the accuracy of the algorithms?

2.7.1 Tight Link Is Not the Bottleneck Link

When the tight link and the bottleneck link are different, the gap model shows that the *IGI* algorithm should use the B_O and g_B values for the tight link when estimating the available bandwidth. Unfortunately, tools such as bprobe only estimate the capacity of the bottleneck link. This will have an impact on the accuracy of the method. Note that *PTR* does not use the B_O and g_B values explicitly, so it will not be affected by this tight link issue. In the remainder of this section we will use ns2 [10] simulation to evaluate the accuracy of both algorithms in this scenario. While simulation has the drawback that it leaves out many real-world effects, it has the advantage that we can study topologies that are difficult or impossible to build.

We use the simulation topology shown in Figure 2.11, using 20Mbps, 10Mbps, and 20Mbps for the link capacities X, Y and Z, respectively. By changing the competing loads C1, C2, and C3 we can change the tight link of the path and also change the level of traffic on links other than the tight link. The probing packet size used in the simulation is 700Byte and the probing packet train length is 60. The competing traffic consists of CBR UDP traffic. Note that by picking link capacities that are fairly close, the available bandwidths on different links are likely to be close as well, which is a challenging case.

In the first set of simulations, we set C2 to 3Mbps and change C1 from 0 to 19Mbps. When C1 is in the range 0–13Mbps, the bottleneck link <R2, R3> is also the tight link,

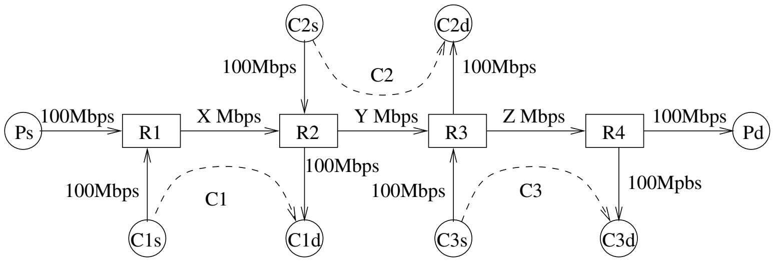


Figure 2.11: Simulation configuration.
Ps and Pd are used for probing. C1s, C1d, C2s, C2d, C3s, and C3d are used for the competing traffic generation.

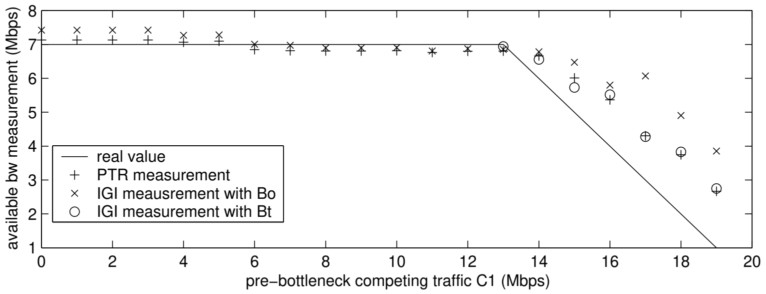


Figure 2.12: Pre-tight link effect.

but when C1 falls in 13–19Mbps, the tight link is <R1, R2>. Figure 2.12 presents the simulation results. We see that when the bottleneck link is equal to the tight link ($0 \leq C1 \leq 13$Mbps), the *IGI* method accurately predicts the available bandwidth, as expected. When <R1, R2> is the tight link, we show the *IGI* estimates based on the B_O and g_B values for both the tight ("o" points) and bottleneck links ("x" points). We see that the results using the tight link values are much closer. The error is the result of interference from competing traffic on the "non-tight" link, as we discuss in more detail in the next subsection.

Next we run a similar set of simulations, but we now keep C2 fixed to 3Mbps and change the competing traffic C3 from 0 to 19Mbps. The tight link switches from <R2, R3> to <R3, R4> when C3 goes above 13Mbps. Figure 2.13 shows that the results are similar to those in Figure 2.12: when the tight link is not the bottleneck link ($13 \leq C3 \leq 19$Mbps), using the B_O and g_B values for the tight link gives a more accurate prediction for the available bandwidth on the path. However, the results when $0 \leq C3 \leq 13$Mbps are less clear than for the pre-tight link case in Figure 2.12, we will explain it in the next section.

In Figures 2.12 and 2.13 we also plot the corresponding *PTR* values. The *PTR* esti-

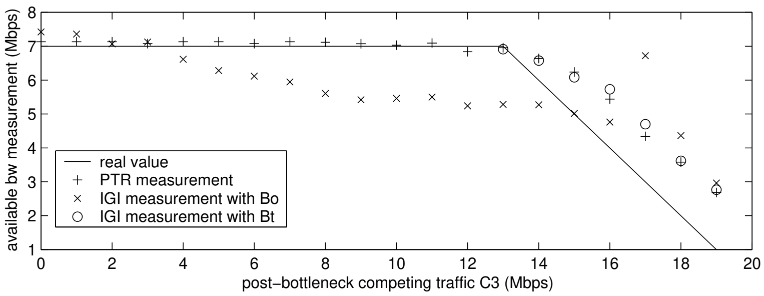


Figure 2.13: Post-tight link effect.

mates are almost identical to the *IGI* estimates that use the B_O and g_B values for the tight link. The reason is that the *PTR* formula does not explicitly use any information about the tight link capacity. The fact that the *IGI* algorithm uses the capacity of the tight link explicitly is a problem because we only have techniques for identifying the link capacity of the bottleneck link, not the tight link. In practice, this is not likely to be a problem: we expect that on many paths, the access link from the client network to the ISP will be both the bottleneck and the tight link. Our Internet measurements in Section 2.5 confirm this.

2.7.2 Interference from Traffic on "Non-tight" Links

In a multi-hop network, each link will potentially affect the gap value of a packet pair or packet train, so we have to effectively concatenate multiple instances of the single-hop gap model. Such a multi-hop gap model is hard to interpret. However, it is fairly easy to see that it is the link with the lowest unused bandwidth (i.e., the tight link) that will have the largest impact on the gap at the destination. The intuition is as follows. On links that have a lot of unused bandwidth, the packets of the probing flow are likely to encounter an empty queue, i.e., these links will have a limited impact on the gap value. Of course, these links may still have some effect on the gap values, as we analyze in this section using the simulation results from the previous section.

The results in Figure 2.12 for $0 \leq C1 \leq 13$Mbps show that both *IGI* and *PTR* are very accurate, even when there is significant competing traffic on a link preceding the tight link. Interesting enough, the second set of simulations show a different result. The results in Figure 2.13 for $0 \leq C3 \leq 13$Mbps correspond to the case that there is significant competing traffic on a link following the tight link. We observe that while *PTR* is still accurate, the *IGI* accuracy suffers.

The different impact on *IGI* of competing traffic in links upstream and downstream of tight link can be explained as follows. Changes in gap values before the tight link will be *reshaped* by the router which the tight link connects with, and the competing traffic on the tight link ends up having the dominating impact. In contrast, any changes in gap values that are caused by traffic on links following the tight link will directly affect the available

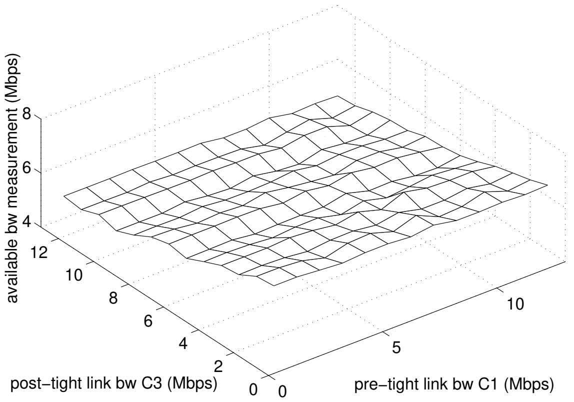


Figure 2.14: Combined pre- and post-tight link effects, with 20Mbps pre- and post-tight link capacities.

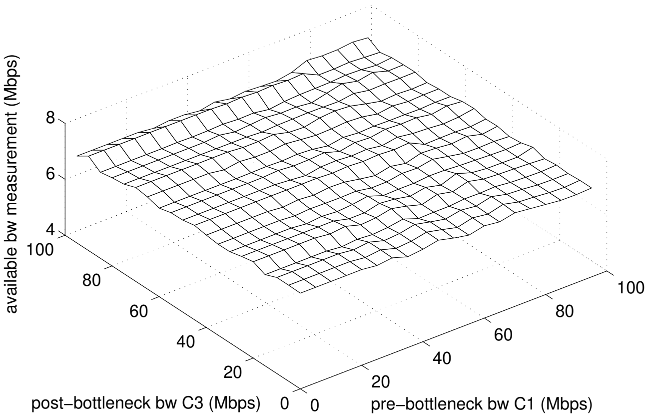


Figure 2.15: Combined pre- and post-tight link effects, with 100Mbps pre- and post-tight link capacities.

bandwidth estimates, so they have a larger impact. Since *IGI* is based on more fine-grain information than *PTR*, it is more sensitive to this effect.

In Figure 2.14 we show the available bandwidth, as estimated by *IGI*, when there is significant competing traffic on both the links before and after the tight link. The actual available bandwidth is 7Mbps for all data points. It is determined by link <R2, R3>, which has 10Mbps capacity and 3Mbps competing traffic (C2). The results confirm the above observation. Even significant competing traffic before the tight link has almost no impact on the accuracy: the curve is basically flat along the C1-axis. Competing traffic after the tight link does however have an effect and, not surprisingly, its impact increases

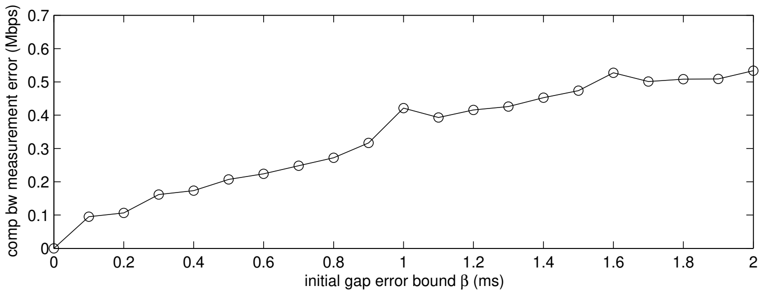


Figure 2.16: Impact of initial gap error

with the level of competing traffic.

Note that the above simulations results are designed to highlight a particularly challenging case. In practice, it is not common to have links with capacities and/or available bandwidths that are this similar. In such cases, the effect of competing traffic on other links is very minimal. For example, we run a set of simulations similar to those described above, but with the <R1, R2> and <R3, R4> set to 100Mbps instead of 20Mbps. The capacity of <R2, R3> and it competing traffic throughput (C2) keep to be 10Mbps and 3Mbps, respectively, i.e., the available bandwidth is still 7Mbps. The results are shown in Figure 2.15. We see that the *IGI* method gives accurate results—the mean value for the data points in this figure is 7.24Mbps, and the standard deviation is 0.10Mbps. The fact that *IGI* and *PTR* typically produce very similar estimates in our Internet experiments shows that the results in Figure 2.15 are much more typical than the worst case results in Figures 2.12 and 2.13.

2.7.3 Impact of Multi-hop Effects on Timing Errors

The above described multi-hop effects also have an important implication for the time errors in the IGI/PTR implementation. There are two types of gap measurement errors: the errors in the initial gap value generated by the source host, and the measurement errors in the final gap value measured on the destination host.

To illustrate the effect of source gap generation error, we use the topology shown in Figure 2.11, with X, Y, and Z set to 20Mbps, 10Mbps, and 20Mbps, respectively. The flow C2 is the only competing flow and we change its throughput in the range of 0–9Mbps. For each experiment, the initial gap (g_I) is incremented by a random value ϵ that is uniformly distributed in $(-\beta, \beta)$, i.e.,

$$g = \max(0, g_I + \epsilon), -\beta < \epsilon < \beta.$$

We run simulations for β ranging from 0–2ms, and for each β value, we collect results when C2 changes between 0 and 9Mbps. Figure 2.16 shows the average absolute error in

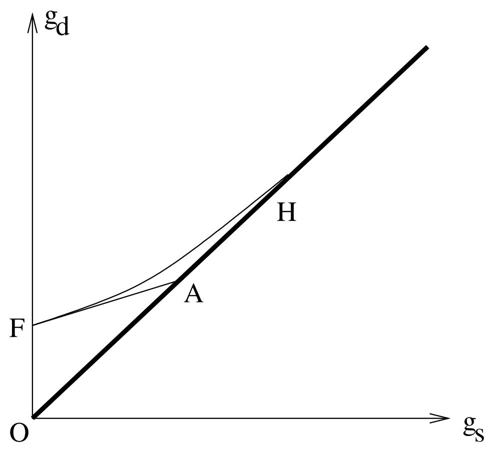


Figure 2.17: Difference between the fluid model and the stochastic model.

the *IGI* estimate as a function of β. We see that the error is small. Note the turning gap value for this simulation is 0.3–1.7ms, so the errors inflicted on the initial gap are quite large. We believe that the reason for the high error tolerance is the same as the reason for the low sensitivity of *IGI* to the pre-tight link traffic. Specifically, the tight link ends up reshaping the gaps according to the competing traffic, thus in effect hiding the initial gap errors. Therefore, measurement errors on the destination side will have a more significant impact since they will directly change the gap values that are used in the *IGI* and *PTR* formulas.

2.8 Improvement for Heavy-Loaded and High-Speed Paths

After we released its source code [6], IGI/PTR has become one of the benchmarks for newly developed available-bandwidth measurement tools like Spruce [113]. These work further confirm the key properties of IGI/PTR like small measurement overhead and short measurement time. However, they also discovered some problems. For example, Strauss et.al. [113] showed that IGI/PTR over-estimates available bandwidth when path load is high, and Shriram et.al. [106] pointed out IGI/PTR does not work well on high-speed network paths. In this section, we show that both problems are due to implementation or system details, not shortcomings of the algorithm. Specifically, the first problem is due to the mechanism used to identify the turning point; while the second is due to small-gap measurement errors. In this section, we present a detailed explanation and describe our approaches to address these problems.

2.8.1 Better Estimation of the Turning Point

Liu et.al. [78] pointed out that a possible reason for the measurement error in IGI/PTR is the divergence from the fluid model used by the IGI/PTR design and their stochastic model. Figure 2.17 illustrates this difference. Using the fluid model, the probing follows line FA, while in the stochastic model, it should follow curve FH. If an implementation uses linear regression to *infer* the turning point, there can be a big error. However, the IGI/PTR implementation does not use this method. Instead, it gradually increases the source gap until the measured destination gap "equals" the source gap. Therefore, the original IGI/PTR implementation is not affected by the simplification of the design model.

The measurement errors mentioned in [113] are instead due to two implementation features. One is that we only sample each source gap once, which makes measurement results sensitive to measurement noise. The other is that the turning-point identification method could be too coarse-grained when the turning-point has a large gap value. In the original implementation, the turning point is identified when the difference between the source gap (g_s) and the destination gap (g_d) is within 20%, i.e., $(g_d - g_s)/g_d < 20\%$. However, when traffic load is high, 20% of gap difference can be too large and the probing can stop prematurely and under-estimate the path load. For example, using 500 byte probing packets on a path with 50Mbps capacity, when the load is 46Mbps, the turning-point gap value is $1ms$. The smallest g_s that satisfies the above 20% requirement is $240us$ (using $g_d = g_B + B_C \cdot g_s/B_O$). That corresponds to 16Mbps probing rate, which is four times of the real available bandwidth value.

Filter measurement noise by sampling a same source gap multiple times

The first issue is relatively easy to address—we only need to collect a number (k) of measurements for each source-gap value and use the average of all k source/destination gap values in the algorithm. To reduce the impact of measurement noise, we prefer a large k value. To minimize measurement overhead, however, small k is better. As a result, the choice of k is a trade-off between measurement accuracy and measurement overhead. To decide the exact value of k, we study the performance of the improved algorithm by changing k's value from 1 to 5. For each k value, the experiment was done with different path load, which was generated using a custom load generator whose packet inter-arrival times follow a Poisson distribution. For each path load, we collected 10 available bandwidth measurements, and plotted their average and variances (i.e., max - min) in Figure 2.18. We can see, with different values of k, the average (in the top graph) does not change much, but the variance (in the bottom graph) are dramatically different. It is clear that measurement variances are larger when k is 1 or 2 compared with the other cases. Although $k = 5$ is overall the best, its improvements over $k = 3$ and $k = 4$ are minor in most cases. In our current implementation, we choose $k = 5$. In practice, we suggest setting k as at least 3.

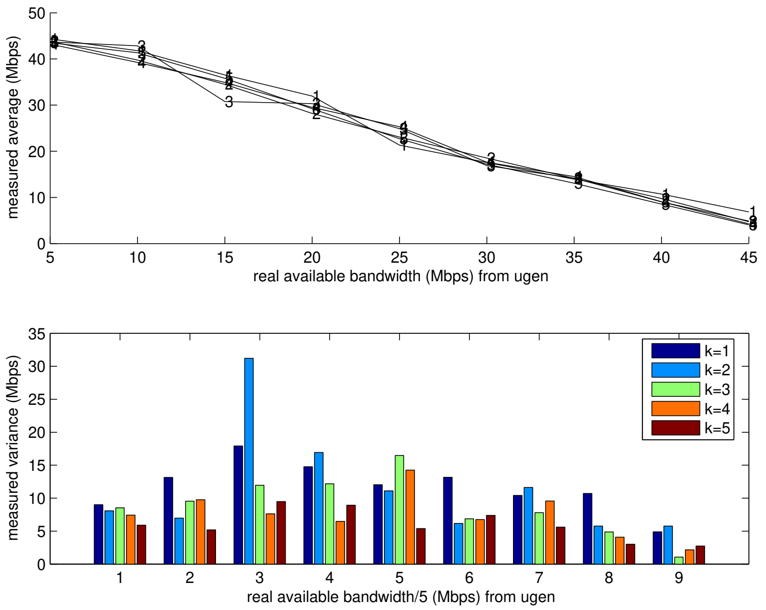


Figure 2.18: Impact of repeated measurements

A better turning-point identification formula

As explained above, the original IGI/PTR implementation detects the turning point based on the *relative* difference between the source and the destination gap values. However, as we will show below in Claim 2.8.1, this relative difference in gap values transforms to an constant absolute error in the available bandwidth estimation. Therefore, for smaller available bandwidth, this constant error results in a larger relative measurement error. That is why we see a large measurement error when path load is high.

Figure 2.19(a) illustrates this insight. In this figure, the x-axis is the source gap value, denoted as g_s, the y-axis is the destination gap value, denoted as g_d. Line $L1$ is the line $g_s = g_d$. Any turning point should be on line $L1$. Point F indicates the bottleneck gap value g_B (see the single-hop gap model in Figure 2.2). Let ϵ be the error margin allowed for the relative difference between source and destination gap values, the IGI/PTR probing stops when $|(g_s - g_d)/g_s| < \epsilon$. Since the algorithm starts sampling with small g_s, it is reasonable to assume $g_d > g_s$. Therefore, when the turning point is detected:

$$g_d < (1 + \epsilon)g_s \tag{2.5}$$

This condition corresponds to the area below line $L2$ in Figure 2.19(a). Assume we have two paths with different available bandwidth, and their turning point are point C and E, respectively. In the extreme case, the estimated turning point for these two paths could be at point A and H,respectively. Visually, the measurement error for turning point E is very large—$g_H < g_E/2$, i.e., the corresponding bandwidth over-estimation is over 50%.

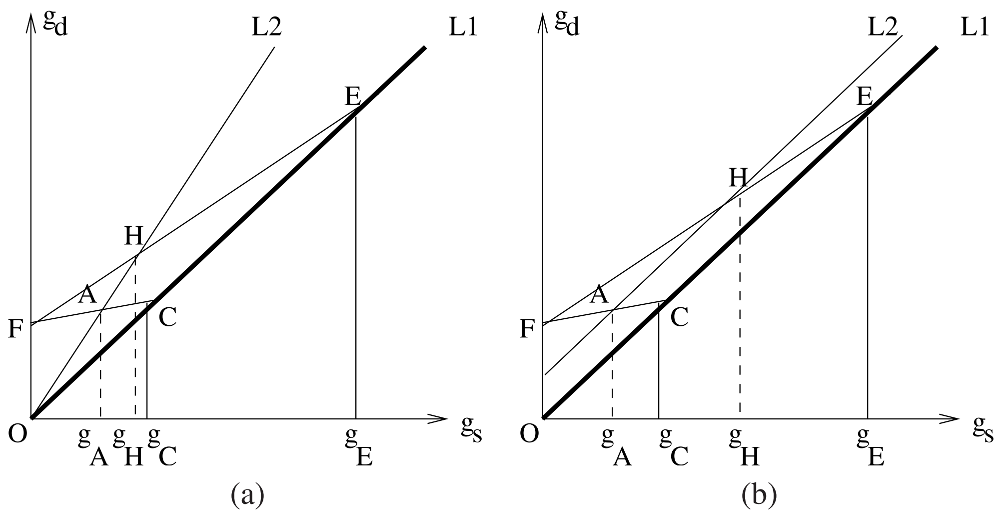


Figure 2.19: The performance of using different turning-point detection formula

To reduce the measurement error, we change line $L2$ into a parallel line with $L1$, as illustrated in Figure 2.19(b). In this figure, the difference between g_H and g_E is significantly reduced. The formula corresponding to the area below line $L2$ can be expressed as

$$g_d < g_s + \Delta \tag{2.6}$$

In Claim 2.8.2 below we will prove that the *relative* measurement error using this method is constant. Therefore, we use this formula in our improved version of IGI/PTR implementation.

Claim 2.8.1 *Ignoring measurement noise, the absolute available bandwidth measurement error (*$\frac{L}{g_s} - \frac{L}{g_o}$*) using Formula (2.5) is constant,* g_o *is the turning point gap value, and* L *is probing packet size.*

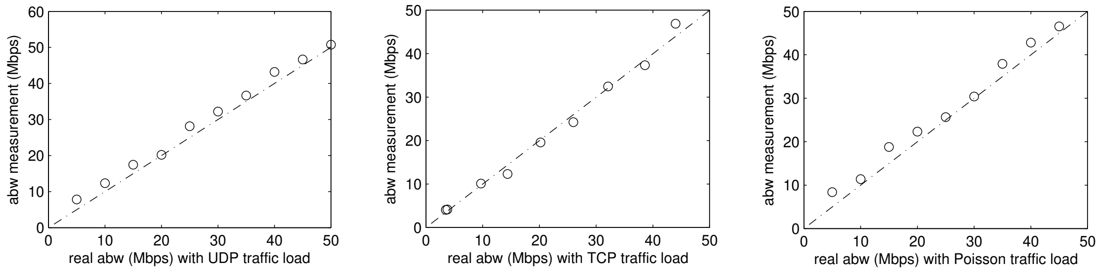
Claim 2.8.2 *Ignoring measurement noise, the relative available bandwidth measurement error (*$\frac{L/g_s - L/g_o}{L/g_o}$*) using Formula (2.6) is constant.*

Proof: Since packet size L is constant, we only need to prove that the value of $\frac{1}{g_s} - \frac{1}{g_o}$ and $\frac{1/g_s - 1/g_o}{1/g_o}$ are constant. From formula (2.2), i.e., $g_d = g_B + B_C \cdot g_s/B_O$, we know the g_d and g_s have the following relationship before reaching the turning point

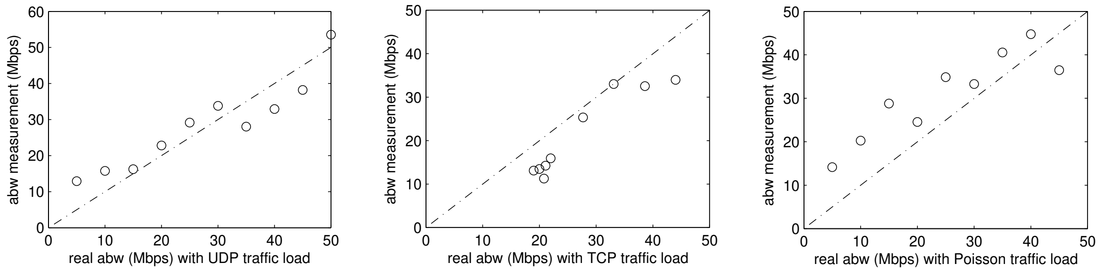
$$g_d = g_B + (1 - a)g_s \tag{2.7}$$

Here $a = B_A/B_O$, and B_A is the available bandwidth. Combining formula (2.5) (treat "$<$" as "$=$") with formula (2.7), we can get

$$\frac{1}{g_s} - \frac{1}{g_o} = \frac{\epsilon}{g_B} = constant$$



(a) Performance of the improved implementation



(a) Performance of the original implementation

Figure 2.20: Performance of the improved IGI/PTR implementation

Similarly, combining formula (2.6) with formula (2.7), we have

$$\frac{1/g_s - 1/g_o}{1/g_o} = \frac{\Delta}{g_B + \Delta} = constant$$

■

Improved implementation

Based on the above two claims, we improved the implementation of IGI/PTR as follows. For each source gap, we probe five times, and use the averages of source gaps and destination gaps in the comparison. The turning-point identification uses formula (2.6), where $\Delta = 5us$. In Figure 2.20, we compare the performance of this improved implementation with that of the original implementation, using three types of background traffic: UDP CBR traffic, TCP flows, and UDP based Poisson traffic. We can see, in all three scenarios, the improved implementation has very small measurement error, which is much better than the original implementation.

Adjustment to the estimated turning point

To accommodate inevitable measurement noise, when identifying the turning point, the source and the destination gap values are allowed to be as large as Δ. Now we estimate the

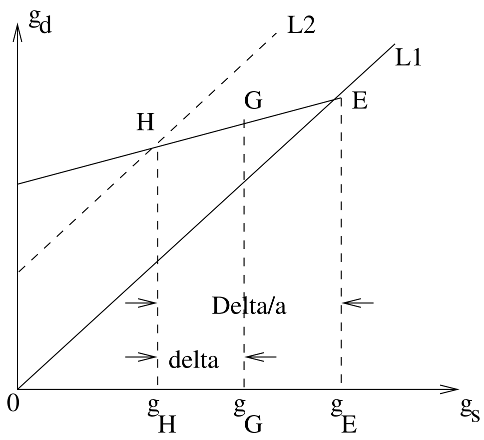


Figure 2.21: Adjust the estimated turning point

average difference between the measured turning point and the real turning point, because that will allow us to adjust our estimation so that it can be probabilistically closer to the real turning point.

There are two main factors that affect this difference: the destination gap measurement error, and the source gap sampling interval. Below we consider the destination gap measurement error first, by assuming the sampling interval can be arbitrarily small; we then consider the second factor.

Let us denote the sampling interval as δ and assume it is very small. In Figure 2.21 (copied from Figure 2.19(b)), if the real turning point is E, without the measurement error, the probing should stop at point H ($g_H = g_o - \Delta/a$). If g_d has error of β, which is assumed to follow an uniform distribution on $[-\gamma, \gamma]$. Using formula (2.2) and (2.7), we know $g_B + (1-a)g_s + \beta = g_s + \Delta$. So when the probing stops, $g_s = g_o + (\beta - \Delta)/a$. If $\gamma > \Delta$, then β can be larger than Δ (with probability of $(\gamma - \Delta)/2\gamma$), thus $g_s > g_o$, that is, the probing stops too late. For the other cases, $g_s \leq g_o$ and the probing stops too early (with probability of $(2\gamma - \Delta)/2\gamma$). If $\gamma \leq \Delta$, then $g_s \leq g_o$ is always true, i.e., the probing always stops early. Regardless of the value of γ, the average error between g_s and g_o is Δ/a, since the average of β is 0. Therefore, g_s should be adjusted as $(g_s + \Delta/a)$ (a can be estimated using the measurement available bandwidth). This means that the white noise β in destination gap measurement does not impact the adjustment of the turning point.

If the source-gap sampling interval δ is not negligible, the probing might not stop at point H; the estimated turning point can be anywhere between H and G ($g_G = g_H + \delta$, assuming $\delta < \Delta$). If we assume any point between H and G has equal probability to be the stop point, then in average the probing should stop at $g_{H'} = g_H + \delta/2$, and the adjustment for the estimated turning point should be $(g_s + (\Delta/a - \delta/2))$.

2.8.2 Accommodating High-Speed Network Paths

[106] pointed out that IGI/PTR does not work well on Gigabit network paths. The reasons include those that have been discussed in related work [68]—interrupt coalescence (also called interrupt throttling) and the system-call overhead of `gettimeofday()`. [97] has demonstrated how to conduct available bandwidth measurements by identifying the signature of interrupt coalescence. That technique, however, has difficulty in obtaining correct measurements when background traffic load is high, where the signature can become obscured.

For IGI/PTR, interrupt coalescence completely disables IGI, which relies on adjacent packet gap values. PTR can continue to work if it uses long enough packet train, it needs to deal with three factors that can affect the measurement accuracy of PTR:

1. Overhead of the `gettimeofday()` system call. Time used for executing this system call can be ignored when the packet gap value is large enough. On a Gigabit network path, packet gap values often do not satisfy this requirement. To address this problem, an obvious method is to avoid using this system call as much as possible. For example, one can use libpcap to obtain kernel timestamps. However, our experiments show that this method does not obviously improve timestamp measurement accuracy. Another method is to only measure the timestamps on the two end packets of the packet train, and then calculate the average packet gap values. This method, however, can not be used to measure destination gaps, because we often do not know which packet is the last one due to packet loss. [2]

2. Packet buffering in the OS introduces errors in source-gap measurements. On the sender side, the sending times are measured using `gettimeofday()` immediately after `sendto()` returns. However, the return of `sendto()` only indicates the end of packet buffer copying, which is not necessarily the packet transmission time. For example, when we send back-to-back 1400B packets from an 1Gbps interface on Emulab pc3000 nodes[3], the average packet gap is measured to be $5 - 9us$, while the theoretical value is $(1400 * 8/1000) = 11.2us$.

3. Source-gap generation error. In IGI/PTR, fine-granularity packet gaps are generated using CPU arithmetic instructions executed between two `sendto()` system calls. This is the best method we are aware of that can reliably generate small time intervals on light-loaded hosts. However, if the turning-point gap value is small, this method can easily miss the real turning point. A slightly larger gap values can significantly under-estimate available bandwidth estimation. This is especially true for

[2] We have also tried to reduce the number of times invoking `gettimeofday()` by measuring the timestamp of every N (say 10) packets. It does not alleviate the problem.

[3] Dell PowerEdge 2850s with a single 3GHz processor, 2GB of RAM, and two 10,000 RPM 146GB SCSI disks.

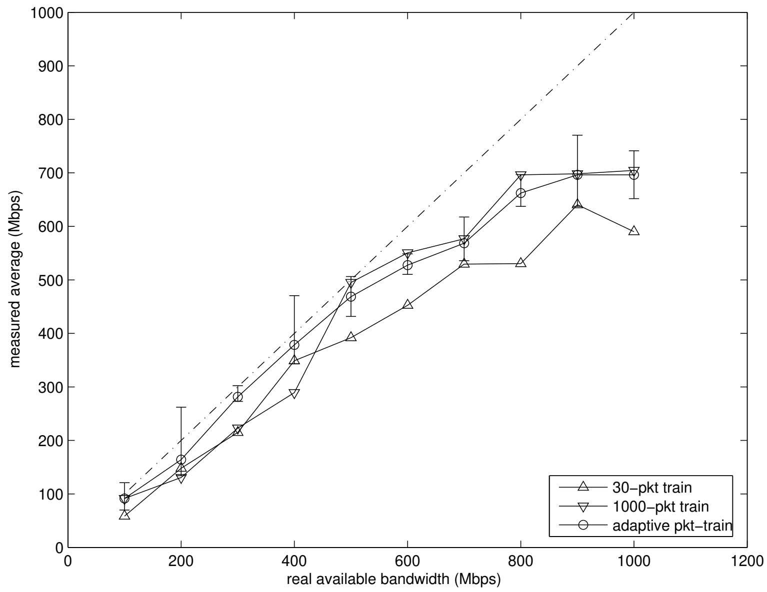


Figure 2.22: PTR measurement of the improve algorithms on 1Gbps network path

high-speed CPUs whose sophisticated CPU architecture makes it hard to accurately emulate small time delay using instruction executions.

It is easy to see that the above three factors tend to have less impact when the path available bandwidth is relatively small. In the following, we show that, by properly setting the length of the packet trains, the PTR algorithm can get reasonably accurate measurements when available bandwidth is less than 800Mbps, which is a significant improve over the original implementation. The idea is to maintain a constant packet-train length in *time*, instead of the number of packets. That is, each time the source node sends out a packet train, it changes the number of packets (N) in the train according to the source gap value, so that the train always covers the same time interval (T). When the source gap value increases, N decreases. In this way, we can use long packet trains while also limiting the measurement overhead, which is an important design principle of our original IGI/PTR technique. As shown below in our analysis, compared with using a packet train that has a constant number of packets, the adaptive algorithm performs better in terms of measurement accuracy.

Figure 2.22 plots the experimental results when running three different versions of the PTR algorithm on a network path with 1Gbps capacity. The first version uses 30-packet packet trains, the second version uses 1000-packet packet trains, while the third version uses the above adaptive algorithm (with T set as $5ms$). Probing packets are all 1400 byte. Path load is generated using UDP CBR traffic. With each load, ten PTR measurements are collected, and we plot the averages in the figure. For the adaptive version, we also plot the measurement variance (i.e., max-min).

Comparison of the three versions shows that the adaptive version is indeed the best one. Between the 30-packet version and the 1000-packet version, the 30-packet version tends to have better accuracy when available bandwidth is smaller than 500Mbps, while the 1000-packet version is better when the available bandwidth is larger than 500Mbps. The adaptive version, however, can combine the better part of both versions, thus achieving the best measurement accuracy.

For all three versions, the measurement errors are higher for larger available bandwidth. This is because the difficulties in timestamp measurement when the turning-point gap value is small. For example, with no background traffic, the turning-point gap value should be $(1400 * 8/1000) = 11.2us$. However, in the first probing phase, the average source gap is $9us(< 11.2us)$, because of source-gap measurement error. The corresponding destination gap is measured as $14us(> 11.2us)$, because of the overhead of running `gettimeofday()` for each packet received. Since $g_s < g_d$, the source gap will be incremented, and eventually results in an under-estimation. We have tried several approaches to alleviate this problem, including using libpcap kernel timestamps, but none was successful.

2.8.3 Discussion

In this section, we have demonstrated the techniques that can improve the performance of IGI/PTR on high-load paths and high-speed paths. However, even with these techniques, the clock granularity of current Linux systems only allows IGI/PTR to measure available bandwidth upto around 800Mbps. That still limits the usage of IGI/PTR on high speed network paths. Due to similar errors in time measurements, IGI/PTR does not work well when the host load is high. These problems deserve future research effort.

2.9 An Application of PTR — TCP PaSt

There are two different ways to use the IGI/PTR technique. One is to directly use it as a tool to measure end-to-end available bandwidth. This is the most straight-forward and also the most popular method. The other way is to use its idea to manipulate data packet transmission within an application to obtain available bandwidth information automatically. In this section, we present an example of the latter method, where we integrate the PTR technique into the TCP Slow Start algorithm to improve TCP startup performance. In this section, we motivate our design, describe our algorithm in detail, and discuss the insights of this integration procedure.

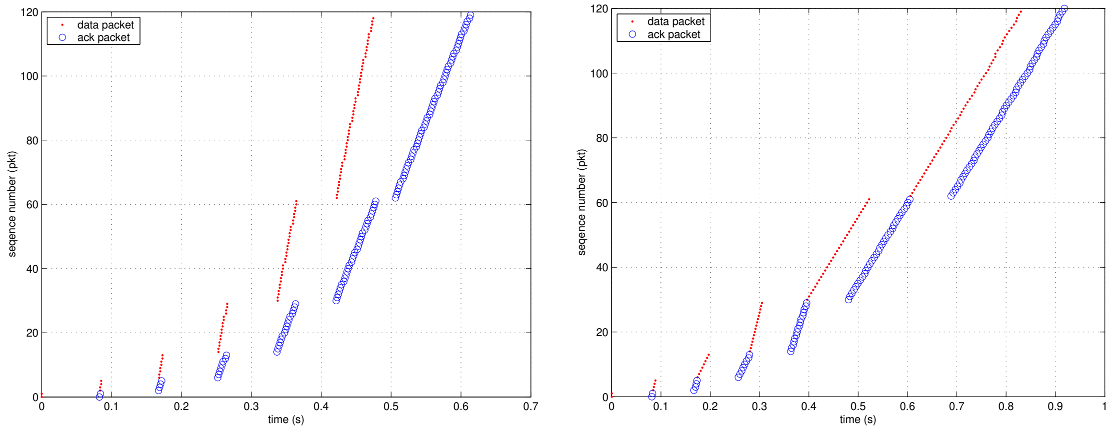


Figure 2.23: Sequence plot for Slow Start (Sack) and Paced Start.
These are from an *ns2* simulation of a path with a roundtrip time of 80ms, a bottleneck link of 5Mbps, and an available bandwidth of 3Mbps. Delayed ACKs are disabled, so there are twice times as many outgoing data packets as incoming ACKs.

2.9.1 PaSt Design

TCP Slow Start algorithm is used at the startup phase of TCP transmission to exponentially increase the window size to identify the right sending rate. It ends either when the congestion window reaches a threshold *ssthresh*, at which point TCP converts to a linear increase of the congestion window, or when packet loss occurs. The performance of Slow Start is unfortunately very sensitive to the initial value of *ssthresh*. If *ssthresh* is too low, TCP may need a very long time to reach the proper window size, while a high *ssthresh* can cause significant packet losses, resulting in a timeout that can greatly hurt the flow's performance. Traffic during Slow Start can be very bursty and can far exceed the available bandwidth of the network path. That may put a heavy load on router queues, causing packet losses for other flows. Furthermore, steady increases in the bandwidth delay products of network paths are exacerbating these effects.

To address this problem, we integrate the PTR algorithm into TCP startup algorithm so that TCP can obtain available bandwidth information, and thus automatically setting a good initial congestion window value. The TCP startup algorithm so modified is referred as TCP Paced Start (PaSt). The idea behind Paced Start is to apply the PTR algorithm to the packet sequence used by TCP Slow Start to get a reasonable estimate for the available bandwidth without flooding the path. An advantage of the using PTR for TCP Slow Start is that TCP startup period only needs to obtain a good approximation. It is sufficient that the initial value of the congestion window is within a factor of two of the "true" congestion window, so that TCP can start the congestion avoidance phase efficiently.

Figure 2.23(a) shows an example of a sequence number plot for Slow Start. We have disabled delayed ACKs during Slow Start as is done by default in some common TCP implementations, e.g. Linux; the results are similar when delayed ACKs are enabled. The graph clearly shows that Slow Start already sends a sequence of packet trains. This sequence has the property that there is one packet train per round trip time, and consecutive trains grow longer (by a factor of two) and become slower (due to the clocking). We decided to keep these general properties in Paced Start, since they keep the network load within reasonable bounds. Early trains may have a very high instantaneous rate, but they are short; later trains are longer but they have a lower rate. Using the same general packet sequence as Slow Start also has the benefit that it becomes easier to engineer Paced Start so it can coexist gracefully with Slow Start. It is not too aggressive or too "relaxed", which might result in dramatic unfairness.

The two main differences between Slow Start and Paced Start are (1) how a packet train is sent and (2) how we transition into congestion avoidance mode. The self-clocking nature of Slow Start means that packet transmission is triggered by the arrival of ACK packets. Specifically, during Slow Start, for every ACK it receives, the sender increases the congestion window by one and sends out two packets (three packets if delayed ACKs are enabled). The resulting packet train is quite bursty and the inter-packet gaps are not regular because the incoming ACKs may not be evenly spaced. This makes it difficult to obtain accurate available bandwidth estimates. To address this problem, Paced Start does not use self-clocking during startup, but instead directly controls the gap between the packets in a train so that it can set the gap to a specific value and make the gaps even across the train. As we discuss in more detail below, the gap value for a train is adjusted based on the average gap between the ACKs for the previous train (we use it as an approximation for the inter-packet gaps at the destination). To do that, we do not transmit the next train until *all* the ACKs for the previous train have been received.

Note that this means that Paced Start is less aggressive than Slow Start. First, in Slow Start, the length of a packet train (in seconds) is roughly equal to the length of the previous ACK train. In contrast, the length of the packet train in Paced Start is based on the sender's estimate on how the available bandwidth of the path compares with the rate of the previous packet train. As a result, Paced Start trains are usually more stretched out than the corresponding Slow Start trains. Moreover, the spacing between the Paced Start trains is larger than that between the Slow Start trains. In Figure 2.23(b), this corresponds to a reduced slope for the trains and an increased delay between trains, respectively. Since Slow Start is widely considered to be very aggressive, making it less aggressive is probably a good thing.

Another important design issue for Paced Start is how to transition into congestion avoidance mode. Slow Start waits for packet loss or until it reaches the statically configured *ssthresh*. In contrast, Paced Start iteratively calculates an estimate for the congestion window of the path and then uses that estimate to transition into congestion avoidance mode. This typically takes three or four probing phases (RTTs), as is discussed in Section 2.9.2. If packet loss occurs during that period, Paced Start transitions into congestion

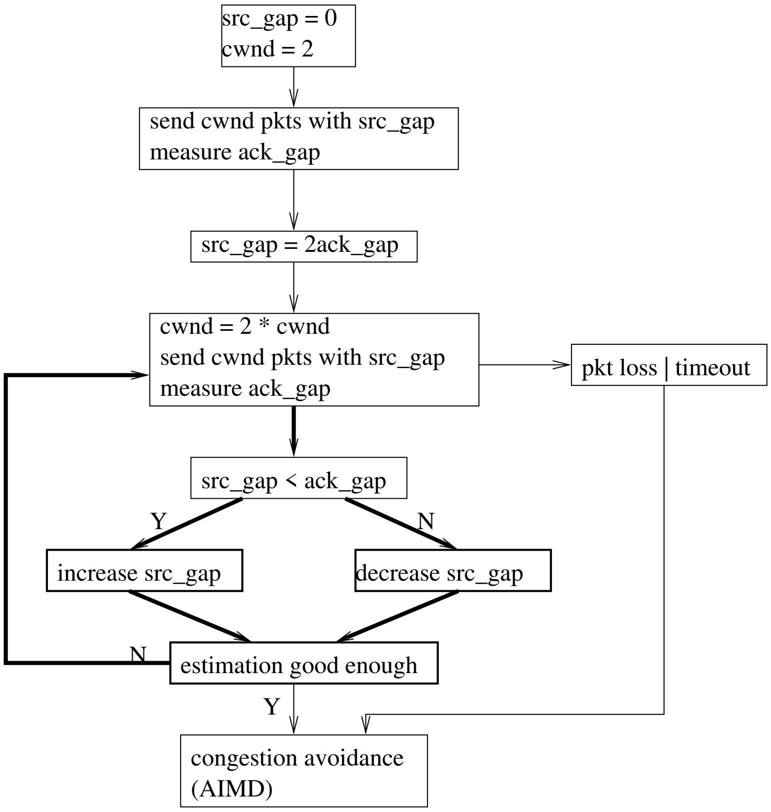


Figure 2.24: The Paced Start (PaSt) algorithm

avoidance mode in exactly the same way as Slow Start does.

2.9.2 PaSt Algorithm

The Paced Start algorithm is shown in the diagram in Figure 2.24. It starts with an initial probing using a packet pair to get an estimate of the path capacity B; this provides an upper bound for the available bandwidth. It then enters the main loop, which is highlighted using bold arrows: the sender sends a packet train, waits for all the ACKs, and compares the average ACK gap with the average source gap. If the ACK gap is larger than the source gap, it means the sending rate is larger than the available bandwidth and we increase the source gap to reduce the rate; otherwise, we decrease the source gap to speed up. In the remainder of this section, we describe in detail how we adjust the gap value and how we terminate Paced Start.

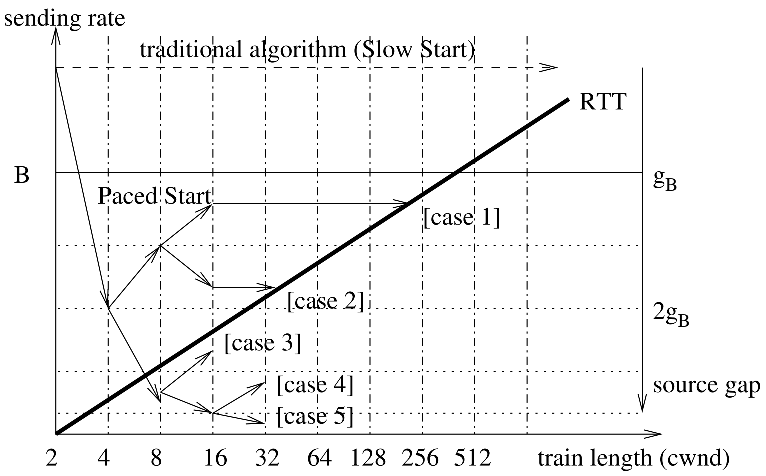


Figure 2.25: Behavior of different startup scenarios.

Gap Adjustment

Figure 2.25 provides some intuition for how we adjust the Paced Start gap value. The bold line shows, for a path with a specific RTT (roundtrip time), the relationship between the congestion window (x-axis) and the packet train sending rate (1/$source\_gap$). The goal of the TCP startup algorithm is to find the point ($cwnd$, $sending\_rate$) on this line that corresponds to the correct congestion window and sending rate of an ideal, stable, well-paced TCP flow. Since the "target" window and rate are related ($cwnd = RTT * sending\_rate$), we need to find only one coordinate.

The traditional Slow Start algorithm searches for the congestion window by moving along the x-axis ($cwnd$) without explicitly considering the y-axis ($sending\_rate$). In contrast, Paced Start samples the 2-D space in a more systematic fashion, allowing it in many cases to identify the target more quickly. In Figure 2.25, the area below the B line includes the possible values of the available bandwidth. The solid arrows show how Paced Start explores this 2-D space; each arrow represents a probing cycle. Similar to Slow Start, Paced Start explores along the x-axis by doubling the packet train length every roundtrip time. Simultaneously, it does a binary search of the y-axis, using information about the change in gap value to decide whether it should increase or decrease the rate. Paced Start can often find a good approximation for the available bandwidth after a small number (3 or 4) of cycles, at which point it "jumps" to the target point, as shown in case 1 and case 2.

The binary search proceeds as follows. We first send two back-to-back packets; the gap at the destination will be the value g_B. In the next cycle, we set the source gap to $2 * g_B$, starting the binary search by testing the rate of $B/2$. Further adjustments of the gap are made as follows:

1. If $g_s < g_d$, we are exploring a point where the Packet Transmission Rate (PTR)

is higher than the available bandwidth, so we need to reduce the PTR. In a typical binary search algorithm, this would be done by taking the middle point between the previous PTR and the current lower bound on PTR. In Paced Start, we can speed up the convergence by using $2 * g_d$ instead of $2 * g_s$. That allows us to use the most recent probing results, which are obtained from longer packet train and generally have lower measurement error.

2. If $g_s \geq g_d$, the PTR is lower than the available rate and we have to reduce the packet gap. The new gap is selected so the PTR of the next train is equal to the middle point between the previous PTR and the current upper bound on PTR.

Algorithm Termination

The purpose of the startup algorithm is to identify the "target" point, as discussed above. This can be done by either identifying the target congestion window or the target rate, depending on whether we reach the target along the x or y axis in Figure 2.25. This translates into two termination cases for Paced Start:

- **Identifying the target rate**: This happens when the difference between source and destination gap values shows that the PTR is a good estimate of the available bandwidth. As we discuss below, this typically takes 3 or 4 iterations. In this case, we set the congestion window size as $cwnd = RTT/g$, where g is the gap value determined by Paced Start. Then we send a packet train using $cwnd$ packets with packet gap g. That fills the transmission pipe, after which we can switch to congestion avoidance mode.

- **Identifying the target congestion window**: When we observe packet loss in the train, either through a timeout or duplicate ACKs, we assume we have exceeded the transmission capacity of the path, as in traditional TCP Slow Start. In this case, we transition into congestion avoidance mode. If there was a timeout, we use Slow Start to refill the transmission pipe, after setting *ssthresh* to half of the last train length. Otherwise we rely on fast recovery.

How Paced Start terminates depends on many factors, including available bandwidth, RTT, router queue buffer size, and cross traffic properties. From our experience, Paced Start terminates by successfully detecting the available bandwidth about 80% of the time, and in the remaining 20% cases, it exits either with a timeout or fast retransmit after packet loss.

Gap Estimation Accuracy

An important question is how many iterations it takes to obtain an available bandwidth estimate that is "close enough" for TCP, i.e. within a factor of two. This means that we

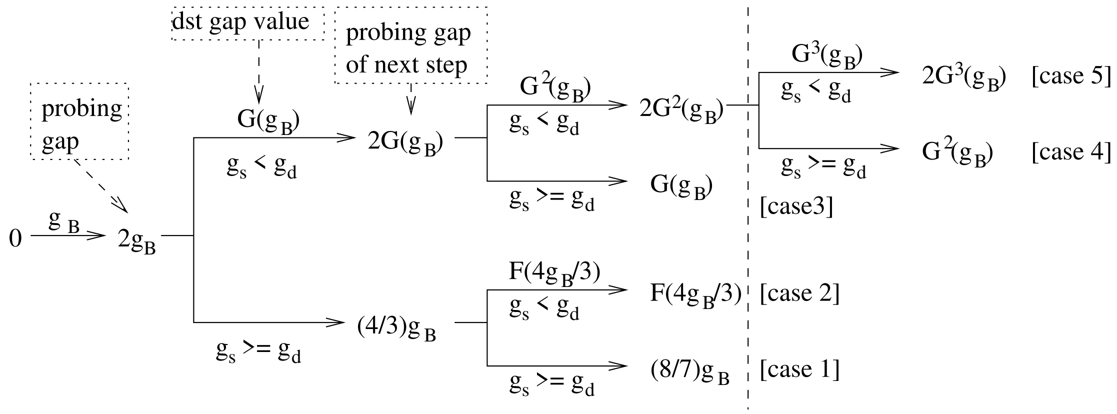


Figure 2.26: Paced Start gap adjustment decision tree.
The formula pointed by an arrow is a possible probing gap g_s; the formula above an arrow is a g_d when $g_s < g_d$, no formula above an arrow means $g_s \geq g_d$.

Table 2.4: Paced Start exiting gap values

case	a	g_{PaSt}	$\frac{1/a}{g_{PaSt}}$
1	(0.75,1)	$8g_B/7$	(0.9, 1.2)
2	(0.5,0.75)	$F((4/3)g_B) = (7/3 - 4a/3)g_B$	(1.0, 1.2)
3	(0.19,0.5)	$G(g_B) = (3 - 2a)g_B$	(1.0, 2.0)
4	(0.08,0.19)	$G^2(g_B) = (1 + 2(1 - a)(3 - 2a))g_B$	(1.0, 2.0)
5	(0,0.08)	$2G^3(g_B) = 2(1 + 2(1 - a)(1 + (1 - a)2(3 - 2a)))g_B$	(0.5, ∞)

need to characterize the accuracy of the available bandwidth estimate obtained by Paced Start.

Figure 2.26 shows the gap values that are used during the binary search assuming perfect conditions. The conditions under which a branch is taken are shown below the arrows while the values above the arrows are the destination gaps; the values at the end of the arrows indicate the source gap for the next step. From Section 2.2, we know if $g_s < g_d$, then the relationship between the source and destination gap is given by $g_d = g_B + (1 - a)g_s$. We use $F(g) = g_B + (1 - a)g$ to denote this relationship. We also use another function, $G(g) = F(2g)$.

This model allows us to calculate how we narrow down the range of possible values for a as we traverse the tree. For example, when during the second iteration we probe with a source gap of $2g_B$, we are testing the point $a = 0.5$. If $g_s < g_d$, we need to test a smaller a by increasing the gap value to $2G(g_B)$ (based on $g_d = G(g_B)$); otherwise, we want to test a larger value ($a = 0.75$, following a binary search) by reducing the gap value to $(4/3)g_B$.

Table 2.4 shows the ranges of a for the 5 exiting cases shown in Figure 2.26. It also lists the ratio between $1/a$ and g_{PaSt}. This corresponds to the ratio between the real send-

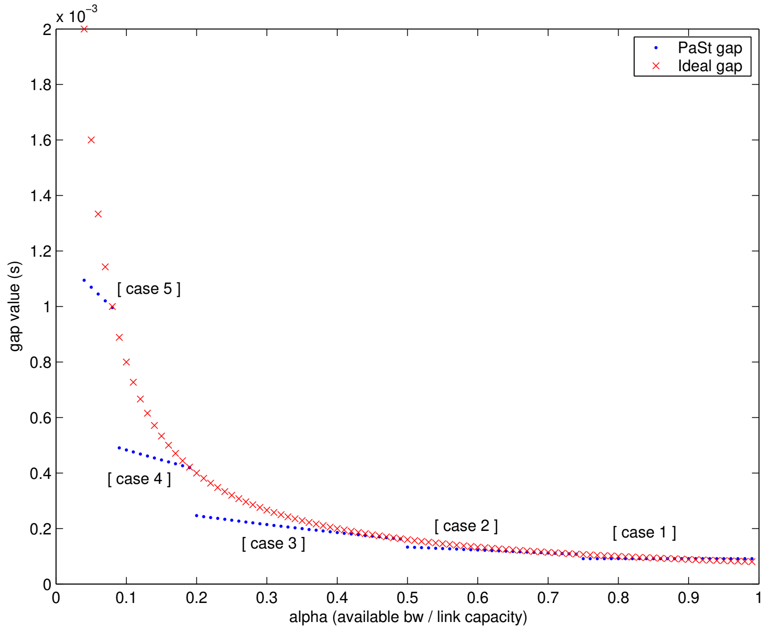


Figure 2.27: Difference between PaSt gap value and ideal gap value.

ing rate and the available bandwidth, i.e. it tells us how much we are overshooting the path available bandwidth when we switch to congestion avoidance mode. Intuitively, Figure 2.27 plots the difference between $1/a$ and g_{PaSt} for a network path with a bottleneck link capacity of 100 Mbps.

From Table 2.4 and Figure 2.27, we can see that if a is high (e.g. cases 1, 2, and 3), we can quickly zoom in on an estimate that is within a factor of two. We still require at least 3 iterations because we want to make sure we have long enough trains so the available bandwidth estimate is accurate enough. This is the case where Paced Start is likely to perform best relative to Slow Start: Paced Start can converge quickly while Slow Start will need many iterations before it observes packet loss.

For smaller a, more iterations are typically needed, and it becomes more likely that the search process will first "bump" into the target congestion window instead of the target rate. This means that Paced Start does not offer much of a benefit since its behavior is similar to that of Slow Start — upon packet loss it transitions into congestion avoidance mode in exactly the same way. In other words, Paced Start's performance is not any worse than that of Slow Start.

2.9.3 Discussion

Using both testbed emulation and Internet experiments, we show that PaSt can significantly improve TCP throughput by reducing its startup time and packet loss. For example, by implementing the PaSt algorithm in Linux kernel and comparing with a regular Linux which implements the TCP Sack protocol, we show that PaSt can remove 90% of the packet loss during startup for a set of web-page transmissions while also achieving 10% average throughput improvement. More evaluation details can be found in [59].

This work demonstrates that to successfully integrate a measurement technique into applications, we need a good understanding of both the network properties and the application requirements. Understanding the network properties allows us to improve the measurement accuracy (e.g., by adjusting the length of packet trains), while understanding the application requirements helps the application measure the right value efficiently (e.g., by finding the right tradeoff between accuracy and overhead). In PaSt, for example, many its properties directly follow from the application (i.e. TCP) requirements: (1) TCP is an adaptive protocol and the purpose of the startup phase is to get a reasonable starting point for the congestion avoidance phase. At the same time, we would like to switch to congestion avoidance mode quickly, so it is important to keep the overhead (number of probing packets) low. Given these two requirements, PaSt cuts off the measurement more quickly than PTR. (2) Since the congestion control tries to track the available bandwidth, it needs the available bandwidth averaged over a short interval. Therefore PaSt uses trains up to a roundtrip time in length. (3) TCP is a two-end protocol. That makes it natural to apply the PTR algorithm. For the ease of deployment, however, PaSt measures the packet train rate at the source (based on ACKs), i.e. it is a one-end implementation.

2.10 Related Work

Bandwidth measurements include path capacity measurements and path available bandwidth measurements. Surveys on bandwidth measurement techniques can be found in [87, 96]. Capacity measurement techniques can be classified into single-packet methods and packet-pair methods. Single-packet methods, like pathchar [64], clink [46] and pchar [79], estimate link capacity by measuring the time difference between the roundtrip times to the two ends of an individual link. This method requires a large numbers of probing packets to filter measurement noise due to factors like queueing delay. Packet-pair path capacity measurement tools include NetDyn probes [30], bprobe [32], nettimer [77], pathrate [45], and CapProbe [71]. In practice, interpreting packet-pair measurements is difficult [92], and accurate measurements generally need to use statistical methods to pick out the packet pairs that correspond to the real path capacity. For example, nettimer uses kernel density estimation to filter measurement noises; it also compares sending rate with receiving rate to pick out good measurement samples. Pathrate explicitly analyzes the multi-modal nature of a packet gap distribution. It first uses a large number of packet-pair measurements to identify all clusters, which generally include the one corresponding to the real capacity. It then uses longer and longer packet trains until the bandwidth distribution becomes unimodal, i.e., converges to the asymptotic dispersion rate. The smallest cluster that is larger than the unimodal cluster then corresponds to the real capacity value.

CapProbe uses one-way delays to identify the packet pairs that are still back-to-back on the bottleneck link. It is based on the observation that packet pairs that are not interfered by competing traffic will have the smallest sum of one-way delays for the two packets in each pair.

Characterizing end-to-end available bandwidth, however, is more difficult than path capacity measurement since path available bandwidth is a dynamic property and depends on many factors. Its dynamic nature means that practical available bandwidth measurements represent an average over some time interval. Therefore, active measurement techniques often use packet trains, i.e., longer sequences of packets. An early example is the PBM (Packet Bunch Mode) method [92]. It extends the packet pair technique by using different-size groups of back-to-back packets. If routers in a network implement fair queueing, bandwidth indicated by back-to-back packet probes is an accurate estimate for the "fair share" of the bottleneck link's bandwidth [72]. Another early example, cprobe [32], sends a short sequence of echo packets between two hosts. By assuming that "almost-fair" queueing occurs during the short packet sequence, cprobe provides an estimate for the available bandwidth along the path between the hosts.

Research on available bandwidth measurement made significant progress since the year 2000, when several effective techniques were proposed, including IGI/PTR. TOPP [85] proposes a theoretical model on how background traffic load changes the transmission rate of probing packets. Our technique—IGI/PTR—leverages on its results. Pathload [65] measures one-way delay of the probing packets in packet trains. If the probing rate is higher than the available bandwidth, delay values of the probing packets will have an increasing trend, and pathload can adjust the probing rate until it is close enough to the real available bandwidth. PathChirp [101] uses packet chirps and also uses one-way delay to identify the packet gap value corresponding to the real available bandwidth. Spruce [113] is the only technique that uses packet pairs instead of packet trains for available bandwidth measurement. Its idea is to use a relatively large number of packet pairs with their packet gap values following a Poisson distribution to capture background traffic throughput. Spruce has better measurement accuracy than Pathload and IGI/PTR [113], but its measurement time, which is around one minutes, is relatively long. For all four techniques (five algorithms), we use Table 2.5 to summarize their differences and commonalities. For differences, we use two criteria to distinguish these techniques:

1. *What to measure*. We can directly measure the transmission *rate* of a packet train to estimate the available bandwidth, as is done in pathChirp, Pathload, and PTR. Alternatively, we can measure the amount of competing traffic on the bottleneck link to indirectly estimate the residual bandwidth. This is done by measuring the changes in the probing packet *gap*, as is done in Spruce and IGI.

2. *How to measure*. All tools use packet pairs, either sent individually or as a train, but they differ in how the packet pair gaps are controlled by the sender. Pathload, IGI, and PTR use packet trains with uniform intervals. In contrast, in pathChirp and

Table 2.5: Comparison of current available bandwidth measurement algorithms

		how to measure			
		non-uniform probing	uniform probing	difference	common
what to measure	a_bw (rate)	**pathChirp**	**Pathload, PTR**	not need B	timer problem
	c_bw (gap)	**Spruce**	**IGI**	need B	two-end control
	difference	long interval	small interval		

(a_bw: available bandwidth; c_bw: background traffic throughput;B: bottleneck link capacity)

Spruce, the packet intervals are statistically constructed, thus the packet train or the sequence of packet pairs is non-uniform.

Different categories have different properties and consequently, they have different advantages and disadvantages:

1. *Assumption.* Techniques that measure background traffic to estimate available bandwidth need to know path capacity. Spruce assumes it is known, while IGI estimates it using existing probing techniques. The problem is that any error in the path capacity estimate directly impacts the available bandwidth measurement accuracy. Rate-based techniques do not have this problem.

2. *Measurement interval.* How the probing trains are constructed affects the averaging interval that is used for the available bandwidth estimate. The uniform probing techniques generally use short packet trains, so they get a relatively short-term snapshot of network performance. Since they measure the available bandwidth averaged over a very short time interval, the estimates will change quickly when the background traffic is very bursty. In contrast, non-uniform probing techniques use statistical sampling over a longer period thus, for example, average out the effects of bursty traffic.

Besides the above differences, all these available bandwidth measurement techniques also share some problems:

1. *System related timer problem.* All techniques rely on the correctness and accuracy of the system timer and the network packet delivery model: any errors that the sending and receiving systems introduce in the timing of the packets will reduce the accuracy of the tools. The timing accuracy becomes more important as the available bandwidth increases. This could be a serious problem on very high speed network, not only because of the limits of timer resolution, but also because they use different packet delivery mechanisms (e.g. batching). Note that techniques that use the timing of individual packet gaps are more sensitive to this effect than techniques that measure packet train rates.

2. *Two-end control.* All current techniques need two-end control, which significantly hinders deployment. Control at the destination is needed to accurately measure the packet gap or packet train rate.

Since available bandwidth is a very dynamic metric, it is often important to know the variance of path available bandwidth. Pathvar [66] is a recently proposed technique to quantify this metric. Its idea is similar to the turning-point in IGI/PTR. That is, it also compares sending rates with arriving rates. For a stationary available-bandwidth process, a packet train with a fixed sending rate has a constant probability of having a destination arriving rate higher than the sending rate. So by sending a large number of trains and then estimating the corresponding probability, the distribution of the original process can be derived. The variance can then be easily calculated based on the distribution. Pathvar has two working modes. The non-parametric mode keeps adjusting probing rate until a specific probability is reached; while in the parametric mode, a Gaussian distribution is assumed, and only two probing rates are needed to infer the mean and variance.

In this chapter, we used a fluid model to illustrate the insight of IGI/PTR design. A more realistic model is presented by Liu et.al.[78], who use a stochastic model to study the properties of packet dispersions on both single-hop networks and multi-hop networks. For single-hope networks, they show that the asymptotic average of the output packet-pair dispersions is a closed-form function of the input dispersion, if assuming cross-traffic stationarity and ASTA sampling. On multi-hop networks, they show that bursty cross traffic can cause negative bias (asymptotic underestimation) to most existing available bandwidth techniques. To mitigate this deviation, a measurement technique should use large probing packet and long probing packet trains.

2.11 Summary

In this chapter, we show that available bandwidth measurement is a solvable problem. We presented and evaluated two measurement algorithms (IGI and PTR), we also demonstrated their applicability. The main results from this chapter are as follows. First, we designed the IGI/PTR tool based on a turning-point idea. That is, using packet-train probing, accurate end-to-end available bandwidth measurement is obtained when its sending rate equals its arriving rate. Second, we showed that, comparing with Pathload, IGI/PTR has a similar measurement accuracy (over 70%), but has a much smaller measurement overhead and uses a much less measurement time. That is, IGI/PTR is both effective and efficient. Third, using packet trains to measure available bandwidth, packet size should not be too small—500-byte to 700-byte packets are empirically best; packet train length should be neither too short nor too long, packet trains with 16-60 packets are appropriate for IGI/PTR. Fourth, on multi-hop network paths, the post-tight-link effect is the major factor that introduces errors in IGI/PTR's measurement. Fifth, PTR can be improved to work reasonably well on Gigabit network paths. Finally, PTR not only can be used as a stand-alone bandwidth measurement tool, it also can be incorporated into existing applications or protocols to improve their performance by providing network performance information.

Chapter 3

Locating Bandwidth Bottlenecks

The IGI/PTR tool shows that we can use the packet-train probing technique to efficiently and effectively measure end-to-end available bandwidth. In this chapter, we show that packet-train probing can also be used to locate bottleneck links. The location of bottleneck links is important diagnostic information for both ISPs and regular end users. For ISP network operators, given the location of a bottleneck link, they can either fix the problem or redirect their customers' traffic to avoid the bottleneck link. Regular end users can use multihoming or overlay routing techniques to avoid the bottleneck link, thus improving the performance of their data transmissions.

However, obtaining bottleneck link location information requires link-level available bandwidth for all links along a network path, which is much harder to obtain than end-to-end available bandwidth. Network operators only have access to the performance information of links on their network, which may not include the bottleneck link. Even if the bottleneck is in their network, they might not be able to catch it since SNMP is often configured to provide 5-minute load average information, so temporary load spikes can be hidden. For end users, it is even harder since they have no access to any link-level information. Several techniques have been proposed to detect the link that has the smallest capacity or available bandwidth, but they either require very long measurement time (e.g., pathchar [64]), or use a large amount of probing packets (e.g., BFind [26]), which significantly limits their usage in practice.

In this chapter, we present an active bottleneck locating technique—Pathneck. It is based on insights obtained from the PTR available bandwidth estimation algorithm. Pathneck allows end users to efficiently and effectively locate bottleneck links on the Internet. *The key idea is to combine measurement packets and load packets in a single probing packet train.* Load packets emulate the behavior of regular data traffic while measurement packets trigger router responses to obtain location information. Pathneck relies on the fact that load packets interleave with competing traffic on the links along the path, thus changing the length of the packet train. By measuring the changes using the measurement packets, the position of congested links can be inferred. Two important characteristics of

Pathneck are that it is extremely light-weight and only requires single-end control. In this chapter, we first describe the Pathneck design (Section 3.1), and then evaluate the technique using both Internet experiments and Emulab testbed emulations (Section 3.2 and 3.3).

3.1 Design of Pathneck

The goal of Pathneck design is to develop a light-weight, single-end-control bottleneck detection tool. In this section, we first present the concept of Recursive Packet Trains and then describe the detailed locating algorithm.

3.1.1 Recursive Packet Train

As defined in Chapter 1, the *bottleneck link* of a network path is the link with the smallest available bandwidth, i.e. the link that determines the end-to-end throughput on the path. Note in this definition, a bottleneck link is not necessarily the one that has the smallest capacity. In this chapter, we will also use the term *choke link*, which refers to any link that has a lower available bandwidth than the partial path from the source to that link. The upstream router for the choke link is called the *choke point* or *choke router*. The formal definitions of choke link and choke point are as follows. Let us assume an end-to-end path from source $S = R_0$ to destination $D = R_n$ through routers $R_1, R_2, ..., R_{n-1}$. Link $L_i = (R_i, R_{i+1})$ has available bandwidth $A_i (0 \leq i < n)$. Using this notation, we define the set of *choke links* as:

$$CHOKE_L = \{L_k | \exists j, 0 \leq j < n, k = argmin_{0 \leq i \leq j} A_i\}$$

and the corresponding set of *choke points* (or *choke routers*) are

$$CHOKE_R = \{R_k | L_k \in CHOKE_L, 0 \leq k < n\}$$

Clearly, choke links will have less available bandwidth as we get closer to the destination, so the last choke link on the path will be the *bottleneck link* or the primary choke link. The second to last choke link is called the *secondary choke link*, and the third to last one is called the *tertiary choke link*, etc.

In order to identify the bottleneck location, we need to measure the train length on *each* link. This information can be obtained with a novel packet train design, called a Recursive Packet Train. Figure 3.1 shows an example of a Recursive Packet Train (RPT); every box is a UDP packet and the number in the box is its TTL value. The probing packet train is composed of two types of packets: measurement packets and load packets. *Measurement packets* are standard traceroute packets, i.e. they are 60 byte UDP packets with properly filled-in payload fields. The figure shows 30 measurement packets at each

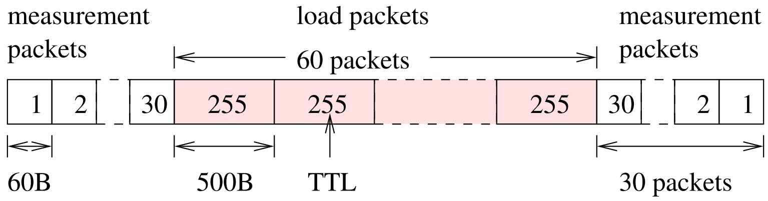


Figure 3.1: Recursive Packet Train (RPT).

end of the packet train, which allows us to measure network paths with up to 30 hops; more measurement packets should be used for longer paths. The TTL values of the measurement packets change linearly, as shown in the figure. *Load packets* are used to generate a packet train with a measurable length. As with the IGI/PTR tool [58], load packets should be large packets that represent an average traffic load. We use 500 byte packets as suggested in [58]. The number of packets in the packet train determines the amount of background traffic that the train can interact with, so it pays off to use a fairly long train. In practice, we set it empirically in the range of 30 to 100.

The probing source sends the RPT packets in a back-to-back fashion. When they arrive at the first router, the first and the last packets of the train expire, since their TTL values are 1. As a result, the packets are dropped and the router sends two ICMP packets back to the source [24]. The other packets in the train are forwarded to the next router, after their TTL values are decremented. Due to the way the TTL values are set in the RPT, the above process is repeated on each subsequent router. The name "recursive" is used to highlight the repetitive nature of this process.

At the source, we can use *the time gap between the two ICMP packets from each router* to estimate the packet train length on the incoming link of that router. The reason is that the ICMP packets are generated when the head and tail packets of the train are dropped. Note that the measurement packets are much smaller than the total length of the train, so the change in packet train length due to the loss of measurement packets can be neglected. For example, in our default configuration, each measurement packet accounts for only 0.2% the packet train length. The time difference between the arrival at the source of the two ICMP packets from the same router is called the *packet gap*.

The `ICMP_TIMESTAMP` Option

The `ICMP_TIMESTAMP` is an option that allows end users to request timestamp from routers. That is, upon receiving an ICMP packets with this option set, a router will reply with an ICMP packet with a timestamp. The timestamp is a 32-bit number representing milliseconds elapsed since midnight of Universal Time. The ICMP protocol (RFC 792) defines three timestamps: the Originate Timestamp, the Receive Timestamp, and the Transmit Timestamp. The Originate Timestamp is the time the sender last touched the packet before sending it, the Receive Timestamp is the time the echoer first touched it on receipt, and the Transmit Timestamp is the time the echoer last touched the packet on sending it. Currently, only one timestamp is used (either the Receive Timestamp or the Transmit Timestamp, depending on the vendor), the other two are set to the same value.

We can use `ICMP_TIMESTAMP` packets as measurement packets to query router timestamps and calculate the gap values. Using this method, gap values are not subject to reverse path congestion anymore, so this approach has the potential to provide more accurate gap measurements. However, we decide *not* to pursue this approach for the following reasons. First, although it has been shown that over 90% of Internet routers respond to this ICMP option [28], on only 37% of end-to-end paths, all routers support this option. Although we can still measure gap values for those routers that do not support this option, so long as the TTL values in the ICMP packets are properly set, this results in two different types of gap values which are hard to compare. This is because unlike the locally measured gap values that are at the microsecond level, the timestamps from routers are at the level of milliseconds, and can hide small gap value changes. Second, the `ICMP_TIMESTAMP` packets need to use router IP addresses as destination IP to query router timestamps, so the forwarding routes used for the measurement packets could be different from those of the load packets due to reasons like ECMP (Equal Cost Multiple Path) routing.

3.1.2 The Pathneck Algorithm

RPT allows us to estimate the probing packet train length on each link along a path. The gap sequences obtained from a set of probing packet trains can then be used to identify the location of the bottleneck link. Pathneck detects the bottleneck link in three steps:

- Step 1: *Labeling of gap sequences*. For each probing train, Pathneck labels the routers where the gap value increases significantly as candidate choke points.
- Step 2: *Averaging across gap sequences*. Routers that are frequently labeled as candidate choke points by the probing trains in the set are identified as actual choke points.
- Step 3: *Ranking choke points*. Pathneck ranks the choke points with respect to their packet train transmission rate.

The remainder of this section describes in detail the algorithms used in each of the three steps.

Labeling of Gap Sequences

Under ideal circumstances, gap values only increase (if the available bandwidth on a link is not sufficient to sustain the rate of the incoming packet train) or stay the same (if the link has enough bandwidth for the incoming packet train), but it should never decrease.

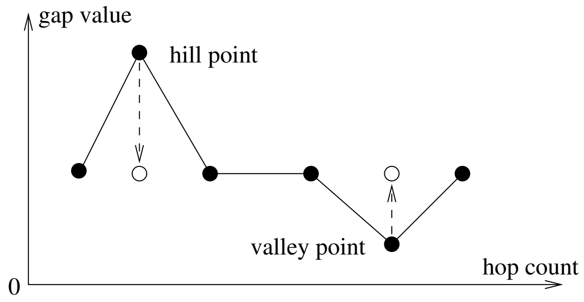


Figure 3.2: Hill and valley points.

In reality, the burstiness of competing traffic and reverse path effects add noise to the gap sequence, so we pre-process the data before identifying candidate choke points. We first remove any data for routers from which we did not receive both ICMP packets. If we miss data for over half the routers, we discard the entire sequence. We then fix the *hill* and *valley* points where the gap value decreases in the gap sequence (Figure 3.2). A hill point is defined as p_2 in a three-point group (p_1, p_2, p_3) with gap values satisfying $g_1 < g_2 > g_3$. A valley point is defined in a similar way with $g_1 > g_2 < g_3$. Since in both cases, the decrease is short-term (one sample), we assume it is caused by noise and we replace g_2 with the closest neighboring gap value.

We now describe the core part of the labeling algorithm. The idea is to match the gap sequence to a step function (Figure 3.3), where each step corresponds to a candidate choke point. Given a gap sequence with len gap values, we want to identify the step function that is the best fit, where "best" is defined as the step function for which the sum of absolute difference between the gap sequence and the step function across all the points is minimal. We require the step function to have clearly defined steps, i.e. all steps must be larger than a threshold ($step$) to filter out measurement noise. We use 100 $microseconds$ (μs) as the threshold. This value is relatively small compared with possible sources of error (to be discussed in Section 3.1.3), but we want to be conservative in identifying candidate choke points.

We use the following dynamic programming algorithm to identify the step function. Assume we have a gap subsequence between hop i and hop j: $g_i, ..., g_j$ $(i \leq j)$, and let us define $avg[i, j] = \sum_{k=i}^{j} g_k/(j - i + 1)$, and the distance sum of the subsequence as $dist\_sum[i, j] = \sum_{k=i}^{j} |avg[i, j] - g_k|$. Let $opt[i, j, l]$ denote the minimal sum of the distance sums for the segments between hop i and j (including hop i and j), given that there are at most l steps. The key observation is that, given the optimal splitting of a subsequence, the splitting of any shorter internal subsequence delimited by two existing splitting points must be an optimal splitting for this internal subsequence. Therefore, $opt[i, j, l]$ can

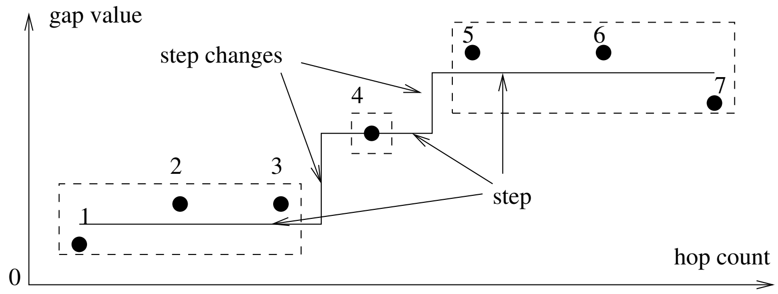


Figure 3.3: Matching the gap sequence to a step function.

be recursively defined as the follows:

$$opt[i,j,l] = \begin{cases} dist\_sum[i,j] & l = 0 \ \& \ i \leq j, \\ min\{opt[i,j,l-1], opt2[i,j,l]\} & l > 0 \ \& \ i \leq j. \end{cases}$$

$$\begin{aligned} opt2[i,j,l] = \ & min\{opt[i,k,l_1] + opt[k+1,j,l-l_1-1] : i \leq k < j, 0 \leq l_1 < l, \\ & |LS[i,k,l_1] - FS[k+1,j,l-l_1-1]| > step\} \end{aligned}$$

Here $LS[i,k,l_1]$ denotes the last step value of the optimal step function fitting the gap subsequence between i and k with at most l_1 steps, and $FS[k+1,j,l-l_1-1]$ denotes the first step value of the optimal step function fitting the gap subsequence between $k+1$ and j with at most $l-l_1-1$ steps.

The algorithm begins with $l = 0$ and then iteratively improves the solution by exploring larger values of l. Every time $opt2[i,j,l]$ is used to assign the value for $opt[i,j,l]$, a new splitting point k is created. The splitting point is recorded in a set $SP[i,j,l]$, which is the set of optimal splitting points for the subsequence between i and j using at most l splitting points. The algorithm returns $SP[0, len-1, len-1]$ as the set of optimal splitting points for the entire gap sequence. The time complexity of this algorithm is $O(len^5)$, which is acceptable considering the small value of len on the Internet. Since our goal is to detect the primary choke point, our implementation only returns the top three choke points with the largest three steps. If the algorithm does not find a valid splitting point, i.e. $SP[0, len-1, len-1] = \emptyset$, it simply returns the source as the candidate choke point.

Averaging Across Gap Sequences

To filter out effects caused by bursty traffic on the forward and reverse paths, Pathneck uses results from multiple probing trains (e.g. 6 to 10 probing trains) to compute *confidence* information for each candidate choke point. To avoid confusion, we will use the term *probing* for a single RPT run and the term *probing set* for a group of probings (generally 10 probings). The outcome of Pathneck is the summary result for a probing set.

For the optimal splitting of a gap sequence, let the sequence of step values be $sv_i (0 \leq i \leq M)$, where M is the total number of candidate choke points. The confidence for a candidate choke point i $(1 \leq i \leq M)$ is computed as

$$conf_i = \left| \frac{1}{sv_i} - \frac{1}{sv_{i-1}} \right| / \frac{1}{sv_{i-1}}$$

Intuitively, the confidence denotes the percentage of available bandwidth change implied by the gap value change. For the special case where the source is returned as the candidate choke point, we set its confidence value to 1.

Next, for each candidate choke point in the probing set we calculate d_rate as the frequency with which the candidate choke point appears in the probing set with $conf \geq 0.1$. Finally, we select those choke points with $d\_rate \geq 0.5$. Therefore, *the final choke points for a path are the candidates that appear with high confidence in at least half of the probings in the probing set*. In Section 3.2.3, we quantify the sensitivity of Pathneck to these parameters.

Ranking Choke Points

For each path, we rank the choke points based on their average gap value in the probing set. The packet train transmission rate R is $R = ts/g$, where ts is the total size for all the packets in the train and g is the gap value. That is, the larger the gap value, the more the packet train was stretched out by the link, suggesting a lower available bandwidth on the corresponding link. As a result, we identify the choke point with the largest gap value as the bottleneck of the path. Note that since we cannot control the packet train structure at each hop, the RPT does <u>not</u> actually measure the available bandwidth on each link, so in some cases, Pathneck could select the wrong choke point as the bottleneck. For example, on a path where the "true" bottleneck is early in the path, the rate of the packet train leaving the bottleneck can be higher than the available bandwidth on the bottleneck link. As a result, a downstream link with slightly higher available bandwidth could also be identified as a choke point and our ranking algorithm will mistakenly select it as the bottleneck.

Note that our method of calculating the packet train transmission rate R is similar to that used by cprobe [32]. The difference is that cprobe estimates available bandwidth, while Pathneck estimates the location of the bottleneck link. Estimating available bandwidth in fact requires careful control of the inter-packet gap for the train [85, 58] which neither tool provides.

While Pathneck does not measure available bandwidth, we can use the average per-hop gap values to provide a rough upper or lower bound for the available bandwidth of each link. We consider three cases:

- *Case 1:* For a choke link, i.e. its gap increases, we know that the available bandwidth is less than the packet train rate. That is, the rate R computed above is an upper bound for the available bandwidth on the link.

- *Case 2:* For links that maintain their gap relative to the previous link, the available bandwidth is higher than the packet train rate R, and we use R as a lower bound for the link available bandwidth.

- *Case 3:* Some links may see a decrease in gap value. This decrease is probably due to temporary queuing caused by traffic burstiness, and according to the packet train model discussed in [58], we cannot say anything about the available bandwidth.

Considering that the data is noisy and that link available bandwidth is a dynamic property, these bounds should be viewed as very rough estimates. We provide a more detailed analysis for the bandwidth bounds on the bottleneck link in Section 3.3.

3.1.3 Pathneck Properties

Pathneck meets the design goals we identified earlier in this section. Pathneck does not need cooperation of the destination, so it can be widely used by regular users. Pathneck also has low overhead. Each measurement typically uses 6 to 10 probing trains of 60 to 100 load packets each. This is a very low overhead compared to existing tools such as pathchar [64] and BFind [26]. Finally, Pathneck is fast. For each probing train, it takes about one roundtrip time to get the result. However, to make sure we receive all the returned ICMP packets, Pathneck generally waits for 3 seconds — the longest roundtrip time we have observed on the Internet — after sending out the probing train, and then exits. Even in this case, a single probing takes less than 5 seconds. In addition, since each packet train probes all links, we get a consistent set of measurements. This, for example, allows Pathneck to identify multiple choke points and rank them. Note however that Pathneck is biased towards early choke points— once a choke point has increased the length of the packet train, Pathneck may no longer be able to "see" downstream links with higher or slightly lower available bandwidth.

A number of factors could influence the accuracy of Pathneck. First, we have to consider the ICMP packet generation time on routers. This time is different for different routers and possibly for different packets on the same router. As a result, the measured gap value for a router will not exactly match the packet train length at that router. Fortunately, measurements in [52] and [28] show that the ICMP packet generation time is pretty small; in most cases it is between $100\mu s$ and $500\mu s$. We will see later that over 95% of the gap changes of detected choke points in our measurements are larger than $500\mu s$. Therefore, while large differences in ICMP generation time can affect individual probings, they are unlikely to significantly affect Pathneck bottleneck results.

Second, as ICMP packets travel to the source, they may experience queueing delay caused by reverse path traffic. Since this delay can be different for different packets, it is a source of measurement error. We are not aware of any work that has quantified reverse path effects. In our algorithm, we try to reduce the impact of this factor by filtering out the measurement outliers. Note that if we had access to the destination, we might be able to estimate the impact of reverse path queueing.

Third, packet loss can reduce Pathneck's effectiveness. Load packet loss can affect RPT's ability to interleave with background traffic thus possibly affecting the correctness of the result. Lost measurement packets are detected by lost gap measurements. Note that it is unlikely that Pathneck would lose significant numbers of load packets without a similar loss of measurement packets. Considering the low probability of packet loss in general [80], we do not believe packet loss will affect Pathneck results.

Fourth, multi-path routing, which is sometimes used for load balancing, could also affect Pathneck. If a router forwards packets in the packet train to different next-hop routers, the gap measurements will become invalid. Pathneck can usually detect such cases by checking the source IP address of the ICMP responses. In our measurements, we do not use the gap values in such cases.

Pathneck also has some deployment limitations. First, the deployment of MPLS can significantly impact Pathneck measurement capabilities since MPLS can hide IP-level routes and make Pathneck only be able to detect AS-level bottlenecks. Second, we discovered that network firewalls often only forward 60 byte UDP packets that strictly conform to the packet payload format used by standard Unix traceroute implementation, while they drop any other UDP probing packets, including the load packets in our RPT. If the sender is behind such a firewall, Pathneck will not work. Similarly, if the destination is behind a firewall, no measurements for links behind the firewall can be obtained by Pathneck. Third, even without any firewalls, Pathneck may not be able to measure the packet train length on the last link, because the ICMP packets sent by the destination host cannot be used. In theory, the destination should generate a "destination port unreachable" ICMP message for each packet in the train. However, due to ICMP rate limiting, the destination network system will typically only generate ICMP packets for some of the probing packets, which often does not include the tail packet. Even if an ICMP packet is generated for both the head and tail packets, the *accumulated* ICMP generation time for the whole packet train makes the returned interval worthless. Of course, if we have the cooperation of the destination, we can get a valid gap measurement for the last hop by using a valid port number, thus avoiding the ICMP responses for the load packets. Below we provide a modification to the packet train to alleviate this problem.

3.1.4 Pathneck-dst—Covering The Last Hop

The last-hop problem of the Pathneck tool significantly impacts its utility, especially on commercial networks where bottlenecks are often on the last hop. To alleviate this problem, we modified the structure of the probing packet train. The idea (suggested by Tom Killian from AT&T Labs–Research) is to use ICMP ECHO packets instead of UDP packets as the measurement packets for the last hop. For example, the packet train in Figure 3.4 is used to measure a 15-hop path. If the destination replies to the ICMP ECHO packet,

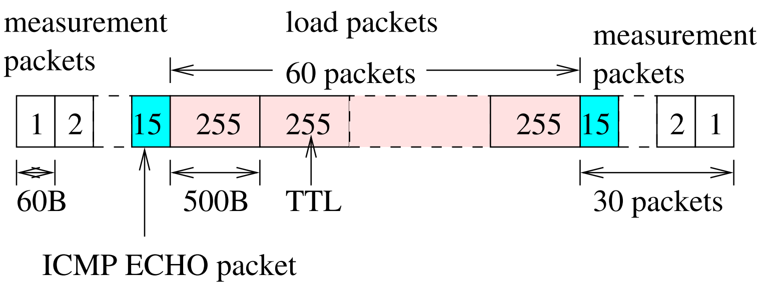


Figure 3.4: The probing packet train used by Pathneck-dst

we can obtain the gap value on the last hop. In order to know where the ICMP packets should be inserted in the probing packet train, Pathneck first uses traceroute to get the path length. Of course, this modification does not work if an end host does not respond to ICMP ECHO packets. In the next chapter, we will show that at least 40% of Internet end nodes in our experiments respond to ICMP ECHO packets, therefore, Pathneck-dst is a significant improvement in terms of measuring the last hop.

To distinguish this modified Pathneck from the original implementation, we will refer this new version as "Pathneck-dst", while using "Pathneck" for the original implementation. Because Pathneck-dst was developed later than Pathneck, some experiments presented later in the dissertation use the original implementation.

3.2 Evaluation of Bottleneck Locating Accuracy

We use both Internet paths and the Emulab testbed [5] to evaluate Pathneck. Internet experiments are necessary to study Pathneck with realistic background traffic, while the Emulab testbed provides a fully controlled environment that allows us to evaluate Pathneck with known traffic loads. Besides the detection accuracy, we also examine the accuracy of the Pathneck bandwidth bounds and the sensitivity of Pathneck to its configuration parameters.

3.2.1 Internet Validation

For a thorough evaluation of Pathneck on Internet paths, we would need to know the actual available bandwidth on all the links of a network path. This information is impossible to obtain for most operational networks. The Abilene backbone, however, publishes its backbone topology and traffic load (5-minute SNMP statistics) [1], so we decided to probe Abilene paths.

We used two sources in the experiment: a host at the University of Utah and a host at Carnegie Mellon University. Based on Abilene's backbone topology, we chose 22 probing

Table 3.1: Bottlenecks detected on Abilene paths.

Probe destination	d_rate (Utah/CMU)	Bottleneck router IP	AS path ($AS1$-$AS2$)†
calren2§	0.71/0.70	137.145.202.126	2150-2150
princeton§	0.64/0.67	198.32.42.209	10466-10466
sox§	0.62/0.56	199.77.194.41	10490-10490
ogig§	0.71/0.72	205.124.237.10 198.32.8.13	210-4600 (Utah) 11537-4600 (CMU)

† $AS1$ is bottleneck router's AS#, $AS2$ is its post-hop router's AS#.
§ calren = `www.calren2.net`, princeton = `www.princeton.edu`,
§ sox = `www.sox.net`, ogig = `www.ogig.net`.

destinations for each probing source, making sure that each of the 11 major routers on the Abilene backbone is included in at least one probing path. From each probing source, every destination is probed 100 times, with a 2-second interval between two consecutive probings. To avoid interference, the experiments conducted at Utah and at CMU were run at different times.

Using $conf \geq 0.1$ and $d\_rate \geq 0.5$, only 5 non-first-hop bottleneck links were detected on the Abilene paths (Table 3.1). This is not surprising since Abilene paths are known to be over-provisioned, and we selected paths with many hops inside the Abilene core. The d_rate values for the 100 probes originating from Utah and CMU are very similar, possibly because they observed similar congestion conditions. By examining the IP addresses, we found that in 3 of the 4 cases (`www.ogig.net` is the exception), both the Utah and CMU based probings are passing through the same bottleneck link close to the destination; an explanation is that these bottlenecks are very stable, possibly because they are constrained by link capacity. Unfortunately, all three bottlenecks are outside Abilene, so we do not have the load data.

For the path to `www.ogig.net`, the bottleneck links appear to be two different peering links going to AS4600. For the path from CMU to `www.ogig.net`, the outgoing link of the bottleneck router 198.32.163.13 is an OC-3 link. Based on the link capacities and SNMP data, we are sure that the OC-3 link is indeed the bottleneck. The SNMP data for the Utah links was not available, so we could not validate the results for the path from Utah to `www.ogig.net`.

3.2.2 Testbed Validation

The detailed properties of Pathneck were studied using the Emulab testbed. Since Pathneck is a path-oriented measurement tool, we used a linear topology (Figure 7.5). Nodes 0 and 9 are the probing source and destination, while nodes 1-8 are intermediate routers. The link delays are roughly set based on a traceroute measurement from a CMU host to

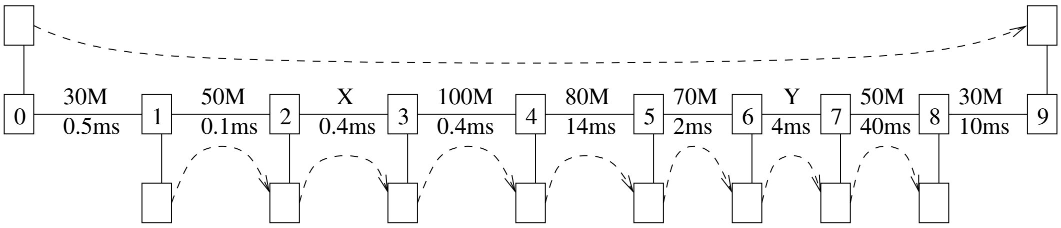


Figure 3.5: Testbed configuration.

Hop 0 is the probing source, hop 9 is the probing destination. Hops 1 - 8 are intermediate routers. The blank boxes are used for background traffic generation. The dashed lines show the default background traffic flow directions. "X" and "Y" are the two links whose capacity will change for different scenarios.

`www.yahoo.com`. The link capacities are configured using the Dummynet [4] package. The capacities for links X and Y depend on the scenarios. Note that all the testbed nodes are PCs, not routers, so their properties such as the ICMP generation time are different from those of routers. As a result, the testbed experiments do not consider some of the router related factors.

The dashed arrows in Figure 7.5 represent background traffic. The background traffic is generated based on two real packet traces, called *light-trace* and *heavy-trace*. The *light-trace* is a sampled trace (using prefix filters on the source and destination IP addresses) collected in front of a corporate network. The traffic load varies from around 500Kbps to 6Mbps, with a median load of 2Mbps. The *heavy-trace* is a sampled trace from an outgoing link of a data center connected to a tier-1 ISP. The traffic load varies from 4Mbps to 36Mbps, with a median load of 8Mbps. We also use a simple UDP traffic generator whose instantaneous load follows an exponential distribution. We will refer to the load from this generator as *exponential-load*. By assigning different traces to different links, we can set up different evaluation scenarios. Since all the background traffic flows used in the testbed evaluation are very bursty, they result in very challenging scenarios.

Table 3.2 lists the configurations of five scenarios that allow us to analyze all the important properties of Pathneck. For each scenario, we use Pathneck to send 100 probing trains. Since these scenario are used for validation, we only use the results for which we received all ICMP packets, so the percentage of valid probing is lower than usual. During the probings, we collected detailed load data on each of the routers allowing us to compare the probing results with the actual link load. We look at Pathneck performance for both probing sets (i.e. result for 10 consecutive probings as reported by Pathneck) and individual probings. For probing sets, we use $conf \geq 0.1$ and $d\_rate \geq 0.5$ to identify choke points. The real background traffic load is computed as the average load for the interval that includes the 10 probes, which is around 60 seconds. For individual probings, we only use $conf \geq 0.1$ for filtering, and the load is computed using a $20ms$ packet trace centered around the probing packets, i.e. we use the instantaneous load.

Table 3.2: The testbed validation experiments

#	X	Y	Trace	Comments
1	50	20	*light-trace* on all	Capacity-determined bottleneck
2	50	50	35Mbps *exponential-load* on Y, *light-trace* otherwise	Load-determined bottleneck
3	20	20	*heavy-trace* on Y, *light-trace* otherwise	Two-bottleneck case
4	20	20	*heavy-trace* on X, *light-trace* otherwise	Two-bottleneck case
5	50	20	30% *exponential-load* on both directions	The impact of reverse traffic

Experiment 1 — Capacity-determined Bottleneck

In this experiment, we set the capacities of X and Y to 50Mbps and 20Mbps, and use *light-trace* on all the links; the starting times within the trace are randomly selected. All 100 probings detect hop 6 (i.e. link Y) as the bottleneck. All other candidate choke points are filtered out because of a low confidence value (i.e. $conf < 0.1$). Obviously, the detection results for the probing sets are also 100% accurate.

This experiment represents the easiest scenario for Pathneck, i.e. the bottleneck is determined by the link capacity, and the background traffic is not heavy enough to affect the bottleneck location. This is however an important scenario on the Internet. A large fraction of the Internet paths fall into this category because only a limited number of link capacities are widely used and the capacity differences tend to be large.

Experiment 2 — Load-determined Bottleneck

Besides capacity, the other factor that affects the bottleneck position is the link load. In this experiment, we set the capacities of both X and Y to 50Mbps. We use the 35Mbps *exponential-load* on Y and the *light-trace* on other links, so the difference in traffic load on X and Y determines the bottleneck. Out of 100 probings, 23 had to be discarded due to ICMP packet loss. Using the remaining 77 cases, the probing sets always correctly identify Y as the bottleneck link. Of the individual probings, 69 probings correctly detect Y as the top choke link, 2 probings pick link $\langle R7, R8\rangle$ (i.e. the link after Y) as the top choke link and Y is detected as the secondary choke link. 6 probings miss the real bottleneck. In summary, the accuracy for individual probings is 89.6%.

Comparing the Impact of Capacity and Load

To better understand the impact of link capacity and load in determining the bottleneck, we conducted two sets of simplified experiments using configurations similar to those used in experiments 1 and 2. Figure 3.6 shows the gap measurements as a function of the hop count (x axis). In the left figure, we fix the capacity of X to 50Mbps and change the capacity of Y from 21Mbps to 30Mbps with a step size of 1Mbps; no background traffic

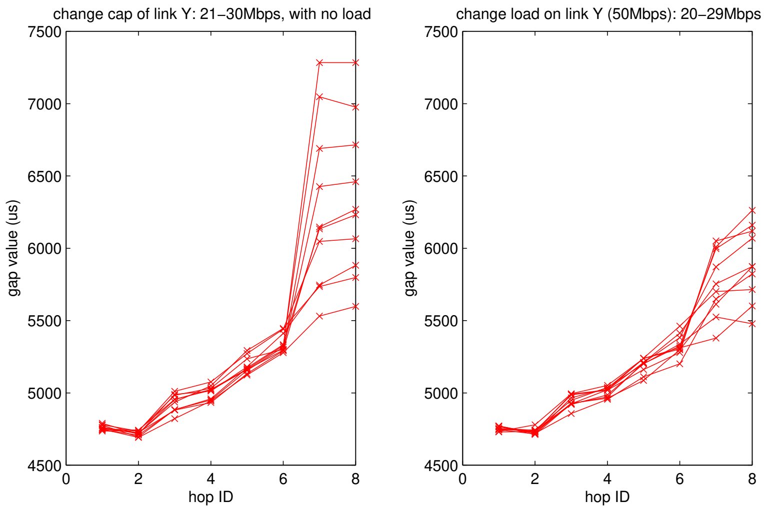


Figure 3.6: Comparing the gap sequences for capacity (left) and load-determined (right) bottlenecks.

is added on any link. In the right figure, we set the capacities of both X and Y to 50Mbps. We apply different CBR loads to Y (ranging from 29Mbps to 20Mbps) while there is no load on the other links. For each configuration, we executed 10 probings. The two figures plot the median gap value for each hop; for most points, the 30-70 percentile interval is under 200μs.

In both configurations, the bottleneck available bandwidth changes in exactly the same way, i.e. it increases from 21Mbps to 30Mbps. However, the gap sequences are quite different. The gap increases in the left figure are regular and match the capacity changes, since the length of the packet train is strictly set by the link capacity. In the right figure, the gaps at the destination are less regular and smaller. Specifically, they do not reflect the available bandwidth on the link (i.e. the packet train rate exceeds the available bandwidth). The reason is that the back-to-back probing packets compete un-fairly with the background traffic and they can miss some of the background traffic that should be captured. This observation is consistent with the principle behind TOPP [85] and IGI/PTR [58], which states that the probing rate should be set properly to accurately measure the available bandwidth. This explains why Pathneck's packet train rate at the destination provides only an upper bound on the available bandwidth. Figure 3.6 shows that the upper bound will be tighter for capacity-determined bottlenecks than for load-determined bottlenecks. The fact that the gap changes in the right figure are less regular than that in the left figure also confirms that capacity-determined bottlenecks are easier to detect than load-determined bottlenecks.

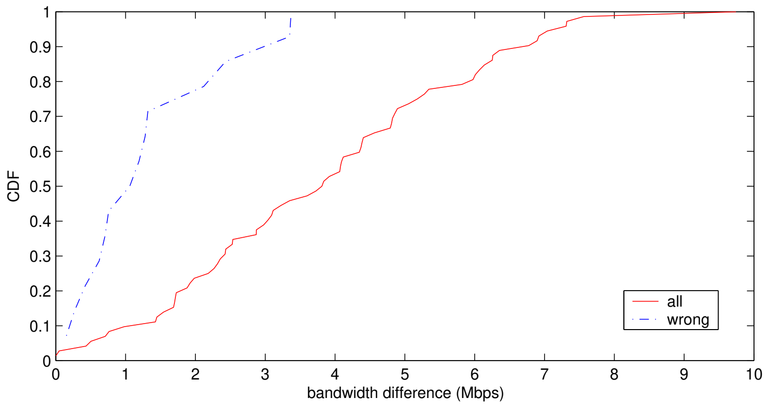


Figure 3.7: Cumulative distribution of bandwidth difference in experiment 3.

Experiments 3 & 4 — Two Bottlenecks

In these two experiments, we set the capacities of both X and Y to 20Mbps, so we have two low capacity links and the bottleneck location will be determined by load. In experiment 3, we use the *heavy-trace* for Y and the *light-trace* for other links. The probing set results are always correct, i.e. Y is detected as the bottleneck. When we look at the 86 valid individual probings, we find that X is the real bottleneck in 7 cases; in each case Pathneck successfully identifies X as the *only* choke link, and thus the bottleneck. In the remaining 79 cases, Y is the real bottleneck. Pathneck correctly identifies Y in 65 probings. In the other 14 probings, Pathneck identifies X as the only choke link, i.e. Pathneck missed the real bottleneck link Y. The raw packet traces show that in these 14 incorrect cases, the bandwidth difference between X and Y is very small. This is confirmed by Figure 3.7, which shows the cumulative distribution of the available bandwidth difference between X and Y for the 14 wrong cases (the dashed curve), and for all 86 cases (the solid curve). The result shows that if two links have similar available bandwidth, Pathneck has a bias towards the first link. This is because the probing packet train has already been stretched by the first choke link X, so the second choke link Y can be hidden.

As a comparison, we apply the *heavy-trace* to both X and Y in experiment 4. 67 out of the 77 valid probings correctly identify X as the bottleneck; 2 probings correctly identify Y as the bottleneck; and 8 probings miss the real bottleneck link Y and identify X as the only bottleneck. Again, if multiple links have similar available bandwidth, we observe the same bias towards the early link.

Experiment 5 — Reverse Path Queuing

To study the effect of reverse path queuing, we set the capacities of X and Y to 50Mbps and 20Mbps, and apply *exponential-load* in both directions on all links (except the two

Table 3.3: The number of times of each hop being a candidate choke point.

Router	1	2	3	4	5	6	7
$conf \geq 0.1$	24	18	5	21	20	75	34
$d\_rate \geq 0.5$	6	0	0	2	0	85	36

edge links). The average load on each link is set to 30% of the link capacity. We had 98 valid probings. The second row in Table 3.3 lists the number of times that each hop is detected as a candidate choke point (i.e. with $conf \geq 0.1$). We observe that each hop becomes a candidate choke point in some probings, so reverse path traffic does affect the detection accuracy of RPTs.

However, the use of probing sets reduces the impact of reverse path traffic. We analyzed the 98 valid probings as 89 sets of 10 consecutive probings each. The last row of Table 3.3 shows how often links are identified as choke points ($d\_rate \geq 0.5$) by a probing set. The real bottleneck, hop 6, is most frequently identified as the actual bottleneck (last choke point), although in some cases, the next hop (i.e. hop 7) is also a choke point and is thus selected as the bottleneck. This is a result of reverse path traffic. Normally, the train length on hop 7 should be the same as on hop 6. However, if reverse path traffic reduces the gap between the hop 6 ICMP packets, or increases the gap between the hop 7 ICMP packets, it will appear as if the train length has increased and hop 7 will be labeled as a choke point. We hope to tune the detection algorithm to reduce the impact of this factor as part of future work.

3.2.3 Impact of Configuration Parameters

The Pathneck algorithms described in Section 3.1.2 use three configuration parameters: the threshold used to pick candidate choke points ($step$ = $100\mu s$), the confidence value ($conf$ = 0.1), and the detection rate (d_rate = 0.5). We now investigate the sensitivity of Pathneck to the value of these parameters.

To show how the $100\mu s$ threshold for the step size affects the algorithm, we calculated the cumulative distribution function for the step sizes for the choke points detected in the "GE" set of Internet measurements (see Table 3.4, we use 3000 destinations selected from BGP tables). Figure 3.8 shows that over 90% of the choke points have gap increases larger than $1000\mu s$, while fewer than 1% of the choke points have gap increases around $100\mu s$. Clearly, changing the step threshold to a larger value (e.g. $500\mu s$) will not change our results significantly.

To understand the impact of $conf$ and d_rate, we re-ran the Pathneck detection algorithm by varying $conf$ from 0.05 to 0.3 and d_rate from 0.5 to 1. Figure 3.9 plots the percentage of paths with at least one choke point that satisfies both the $conf$ and d_rate thresholds. The result shows that, as we increase $conf$ and d_rate, fewer paths have iden-

Table 3.4: Probing sources from PlanetLab (PL) and RON.

ID	Probing Source	AS Number	Location	Upstream Provider(s)	Testbed	BW	GE	CR	OV	MH
1	ashburn	7911	DC	2914	PL		√		√	
2	bkly-cs	25	CA	2150,3356,11423,16631	PL		√	√	√	√
3	columbia	14	NY	6395	PL		√		√	
4	diku	1835	Denmark	2603	PL		√		√	
5	emulab	17055	UT	210	–		√		√	
6	frankfurt	3356	Germany	1239, 7018	PL		√		√	
7	grouse	71	GA	1239, 7018	PL		√		√	
8	gs274	9	PA	5050	–		√		√	
9	bkly-intel	7018	CA	1239	PL		√		√	
10	jhu	5723	MD	7018	PL		√		√	√
11	nbgisp	18473	OR	3356	PL		√		√	
12	princeton	88	NJ	7018	PL		√	√	√	√
13	purdue	17	IN	19782	PL		√	√	√	
14	rpi	91	NY	6395	PL		√		√	√
15	uga	3479	GA	16631	PL		√	√	√	
16	umass	1249	MA	2914	PL		√	√	√	
17	unm	3388	NM	1239	PL		√	√	√	
18	utah	17055	UT	210	PL		√		√	
19	uw-cs	73	WA	101	PL		√	√	√	
20	mit-pl	3	MA	1	PL			√		√
21	cornell	26	NY	6395	PL				√	
22	depaul	20130	CH	6325, 16631	PL					√
23	umd	27	MD	10086	PL					√
24	dartmouth	10755	NH	13674	PL					√
25	kaist	1781	Korea	9318	PL					√
26	cam-uk	786	UK	8918	PL					√
27	ucsc	5739	CA	2152	PL					√
28	ku	2496	KS	11317	PL					√
29	snu-kr	9488	Korea	4766	PL					√
30	bu	111	MA	209	PL					√
31	northwestern	103	CH	6325	PL					√
32	cmu	9	PA	5050	PL					√
33	stanford	32	CA	16631	PL					√
34	wustl	2552	MO	2914	PL					√
35	msu	237	MI	3561	PL					√
36	uky	10437	KY	209	PL					√
37	ac-uk	786	UK	3356	PL					√
38	umich	237	MI	3561	PL					√
39	mazu1	3356	MA	7018	RON				√	
40	aros	6521	UT	701	RON	√	√	√	√	
41	jfk1	3549	NY	1239, 7018	RON	√	√	√	√	
42	nortel	11085	Canada	14177	RON	√	√	√	√	
43	nyu	12	NY	6517, 7018	RON	√	√	√	√	
44	vineyard	10781	MA	209, 6347	RON		√	√	√	
45	intel	7018	CA	1239	RON		√	√	√	
46	cornell	26	NY	6395	RON	√		√		
47	lulea	2831	Sweden	1653	RON	√		√		
48	ana1	3549	CA	1239, 7018	RON	√		√		
49	ccicom	13649	UT	3356, 19092	RON	√		√		
50	ucsd	7377	CA	2152	RON	√		√		
51	gr	3323	Greece	5408	RON			√		
52	utah	17055	UT	210	RON	√		√		

BW: used for bandwidth accuracy analysis in Section 3.3; GE: used for the analysis in Section 3.2.3 and 4.1;
CR: used for correlation analysis in Section 4.4; OV: used for overlay analysis in Section **??**;
MH: used for multihoming analysis in Section **??**; "–": probing hosts obtained privately.

tifiable choke points. This is exactly what we would expect. With higher values for $conf$ and d_rate, it becomes more difficult for a link to be consistently identified as a choke link. The fact that the results are much less sensitive to d_rate than $conf$ shows that most of the choke point locations are fairly stable within a probing set (short time duration).

The available bandwidth of the links on a path and the location of both choke points and the bottleneck are dynamic properties. The Pathneck probing trains effectively sample these properties, but the results are subject to noise. Figure 3.9 shows the tradeoffs involved in using these samples to estimate the choke point locations. Using high values for $conf$ and d_rate will result in a small number of stable choke points, while using lower values will also identify more transient choke points. Clearly the right choice will depend

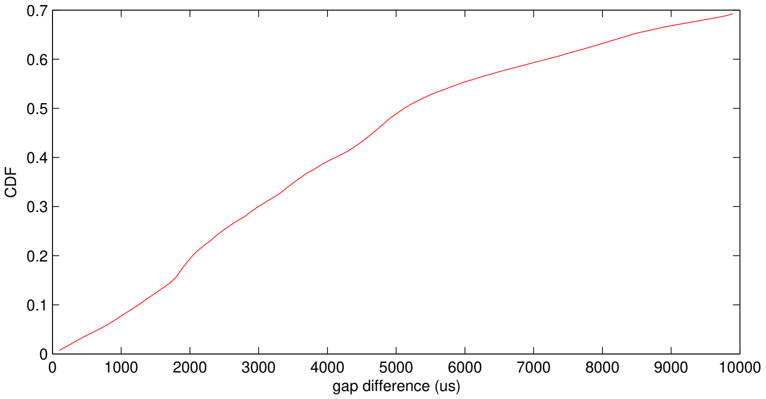


Figure 3.8: Distribution of step size on the choke point.

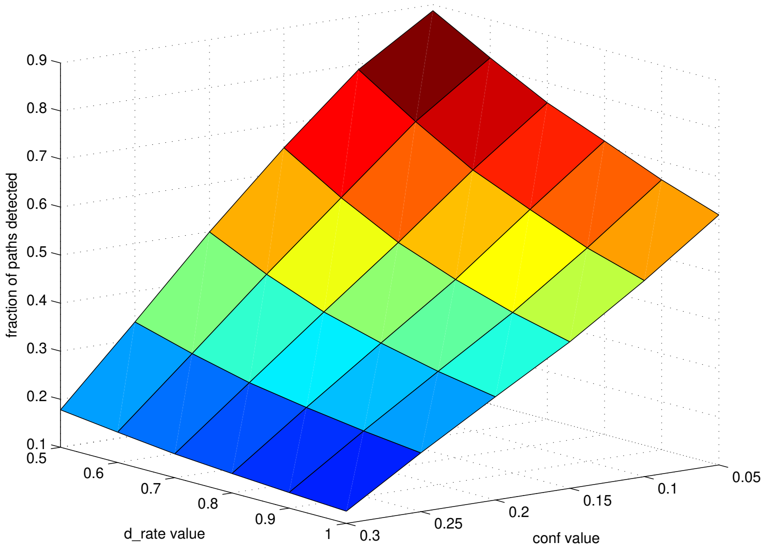


Figure 3.9: Sensitivity of Pathneck to the values of $conf$ and d_rate.

on how the data is used. We see that for our choice of $conf$ and d_rate values, 0.1 and 0.5, Pathneck can clearly identify one or more choke points on almost 80% of the paths we probed. The graph suggests that our selection of thresholds corresponds to a fairly liberal notion of choke point.

3.3 Tightness of Bottleneck-Link Bandwidth Bounds

A number of groups have shown that packet trains can be used to estimate the available bandwidth of a network path [85, 58, 65]. However, the source has to carefully control the inter-packet gap, and since Pathneck sends the probing packets back-to-back, it cannot, in general, measure the available bandwidth of a path. Instead, as described in Section 3.1.2, the packet train rate at the bottleneck link can provide a rough upper bound for the available bandwidth. In this section, we compare the upper bound on available bandwidth on the bottleneck link reported by Pathneck with end-to-end available bandwidth measurements obtained using IGI/PTR [58] and Pathload [65].

Since both IGI/PTR and Pathload need two-end control, we used 10 RON nodes for our experiments — ana1, aros, ccicom, cornell, jfk1, lulea, nortel, nyu, ucsd, utah (see the "BW" column in Table 3.4); this results in 90 network paths for our experiment. On each RON path, we obtain 10 Pathneck probings, 5 IGI/PTR measurements, and 1 Pathload measurement[1]. The estimation for the upper bound in Pathneck was done as follows. If a bottleneck can be detected from the 10 probings, we use the median packet train transmission rate on that bottleneck. Otherwise, we use the largest gap value in each probing to calculate the packet train rate and use the median train rate of the 10 probings as the upper bound.

Figure 3.10 compares the average of the available bandwidth estimates provided by IGI, PTR, and Pathload (x axis) with the upper bound for the available bandwidth provided by Pathneck (y axis). The measurements are roughly clustered in three areas. For low bandwidth paths (bottom left corner), Pathneck provides a fairly tight upper bound for the available bandwidth on the bottleneck link, as measured by IGI, PTR, and Pathload. In the upper left region, there are 9 low bandwidth paths for which the upper bound provided by Pathneck is significantly higher than the available bandwidth measured by IGI, PTR, and Pathload. Analysis shows that the bottleneck link is the last link, which is not visible to Pathneck. Instead, Pathneck identifies an earlier link, which has a higher bandwidth, as the bottleneck.

The third cluster corresponds to high bandwidth paths (upper right corner). Since the current available bandwidth tools have a relative measurement error around 30% [58], we show the two 30% error margins as dotted lines in Figure 3.10. We consider the upper bound for the available bandwidth provided by Pathneck to be *valid* if it falls within these error bounds. We find that most upper bounds are valid. Only 5 data points fall outside of the region defined by the two 30% lines. Further analysis shows that the data point above the region corresponds to a path with a bottleneck on the last link, similar to the cases mentioned above. The four data points below the region belong to paths with the same source node (lulea). We have not been able to determine why the Pathneck bound is too low.

[1]We force Pathload to stop after 10 fleets of probing. If Pathload has not converged, we use the average of the last 3 probings as the available bandwidth estimate.

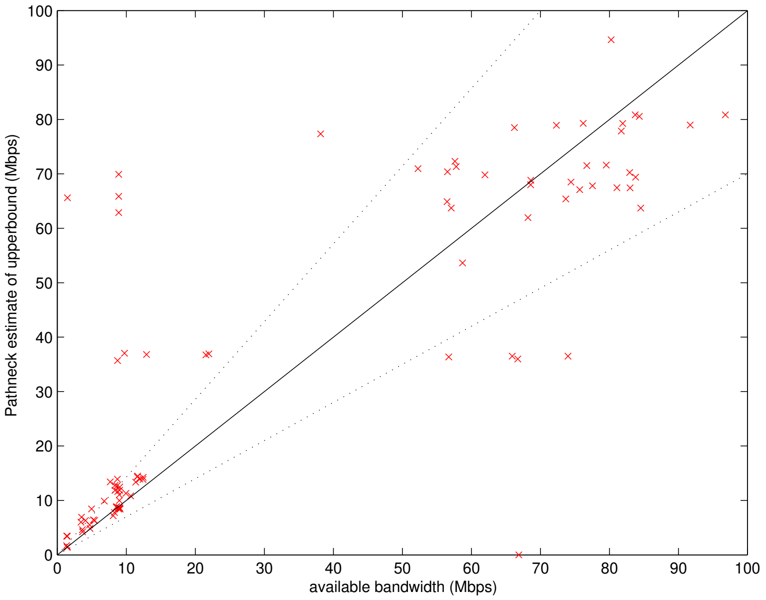


Figure 3.10: Comparison between the bandwidth from Pathneck with the available bandwidth measurement from IGI/PTR and Pathload.

3.4 Related Work

Ideally, if we have SNMP load data for all links along a network path, it is trivial to identify the link that has the smallest available bandwidth. However, such information is seldom publicly accessible. Even if it is, SNMP data is generally only collected for 5-minute intervals, so it may not correctly reflect the situation at smaller time granularities. For this reason, all current bottleneck locating tools use active measurement techniques. These include Cartouche [54], STAB [100], BFind [26], and Pathchar [64]. Among them, Cartouche is the closest to the Pathneck tool. Cartouche uses a packet train that combines packets of different sizes to measure the bandwidth for any segment of the network path. The bottleneck location is deduced from its measurement results. STAB also uses two different sizes probing packets, but instead of letting small size packets expire like Pathneck, it expires the large packets. Both Cartouche and STAB require two-end control, while Pathneck only needs single-end control. Also Pathneck tends to use less probing packets than these two techniques.

BFind only needs single-end control. It detects the bottleneck position by injecting a steady UDP flow into the network path, and by gradually increasing its throughput to amplify the congestion at the bottleneck router. At the same time, it uses traceroute to monitor the RTT changes to all the routers on the path, thus detecting the position of the most congested link. Concerns about the overhead generated by the UDP flow force BFind to only look for bottlenecks with available bandwidths of less than 50Mbps. Moreover, considering its measurement time, its overhead is fairly high, which is undesirable for a general-purpose probing tool.

Pathchar estimates the capacity of each link on a network path. It can be used to locate the link that has the smallest capacity, which may or may not be the bottleneck link (the link that has the smallest available bandwidth) as we defined. The main idea of pathchar is to measure the link-level packet transmission time. This is done by taking the difference between the RTTs from the source to two adjacent routers. To filter out measurement noise due to factors such as queueing delay, pathchar needs to send a large number of probing packets, identifying the smallest RTT values for the final calculation. As a result, pathchar also has a large probing overhead.

3.5 Summary

This chapter presented the Pathneck tool that can locate Internet bottlenecks. Pathneck uses a novel packet train structure—Recursive Packet Train—to associate link available bandwidth with link location information. Its overhead is several magnitudes lower than previously proposed bottleneck locating tools. Using both Internet experiments and Emulab testbed emulations, we showed that Pathneck can accurately identify bottleneck links on 80% of the paths we measured. The paths where Pathneck makes mistakes generally have an ealier link that has an available bandwidth similar to that of the bottleneck link. Using the RON testbed, we also showed that the bottleneck-link available-bandwith upper-bounds provided by Pathneck are fairly tight and can be used for applications that only need rough estimations for path available bandwidth.

Chapter 4

Internet Bottleneck Properties

Equipped with the light-weight Pathneck tool, and in collaboration with our colleagues, we have studied several bottleneck properties at the Internet scale [56, 55, 57]. In [56], we studied bottleneck popularity, bottleneck inference, and how to avoid bottlenecks. In [55], we looked at bottleneck persistence, bottleneck clustering, relationships between bottleneck links and link loss/delay, and the correlation between bottleneck links and link-level performance metrics. In [57], we studied bottleneck location distribution, Internet end-user access-bandwidth distribution, and analyzed how distributed systems like CDN (Content Distribution Network) can be used to avoid bottlenecks and improve end users' access bandwidth. To the best of my knowledge, this is the first bottleneck property study at an Internet scale.

In this chapter, I present the results from the four most insightful studies—the bottleneck link location distribution, Internet end-user access-bandwidth distribution, the persistence of Internet bottlenecks, and the relationship between bottleneck links and link loss/delay. The insights from these studies not only greatly improve our understanding of Internet bottlenecks, they also help us improve the performance of network applications. To demonstrate this point, in the last part of this chapter, I describe how we use bottleneck information to improve the performance of a content distribution system, and to obtain a transmission time distribution for web sites.

4.1 Distribution of Bottleneck Locations

The common intuition about Internet bottlenecks is that most of them are on the Internet edge, but this intuition has not been tested due to the lack of an efficient tool. In this section, we use Pathneck to quantitatively evaluate this assumption. Below we first describe the data sets we collected for our analysis, and then present the location distribution of Internet bottlenecks.

Table 4.1: Measurement source nodes

ID	Location	ID	Location	ID	Location	ID	Location
$S01$	US-NE	$S06$	US-SE	$S11$	US-NM	$S15$	Europe
$S02$	US-SM	$S07$	US-SW	$S12$	US-NE	$S16$	Europe
$S03$	US-SW	$S08$	US-MM	$S13$	US-NM	$S17$	Europe
$S04$	US-MW	$S09$	US-NE	$S14$	US-NE	$S18$	East-Asia
$S05$	US-SM	$S10$	US-NW				

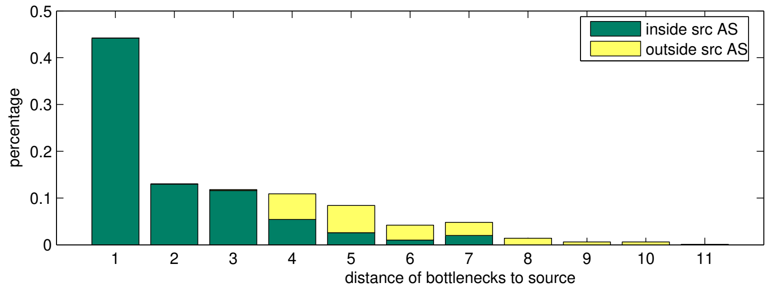
NE: northeast, NW: northwest, SE: southeast, SW: southwest
ME: middle-east, MW: middle-west, MM: middle-middle

4.1.1 Data Collection

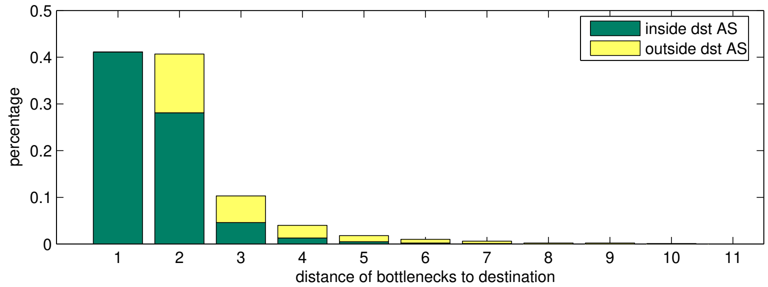
We use three data sets in this chapter. The first is the data set collected in December 2003 using the "GE" source nodes listed in Table 3.4, which consists of 25 nodes from PlanetLab [12] and RON [15]. For this data set, each node probed 3000 destinations which are diversely distributed over a BGP table. For our bottleneck location analysis, we only use the paths where bottlenecks are on the source side, i.e., the first half of a path. We ignore the other paths because the Pathneck tool used to collect this data set could not measure the last hop of Internet paths. This data set is called the *BtSrc* data set.

The second data set was collected in February 2005 using the Pathneck-dst tool, which can measure the last hop of Internet paths. To achieve an Internet scale, we select one IP address as the measurement destination from each of the 165K prefixes extracted from a BGP table. Ideally, the destination should correspond to an online host that replies to ping packets. However it is difficult to identify online hosts without probing them. In our study, we partially alleviate this problem by picking IP addresses from three pools of existing data sets collected by a Tier-1 ISP: Netflow traces, client IP addresses of some Web sites, and the IP addresses of a large set of local DNS servers. That is, for each prefix, whenever possible, we use one of the IP addresses from those three sources in that prefix; otherwise, we randomly pick an IP address from that prefix. In this way, we were able to find reachable destination in 67,983 of the total 165K prefixes. We used a single source node at CMU to probe these reachable destinations. Since the measurements share a source, we only use those paths where bottlenecks are on the destination side, i.e., where bottleneck is at the later half of a path. This data set is called the *BtDst* data set.

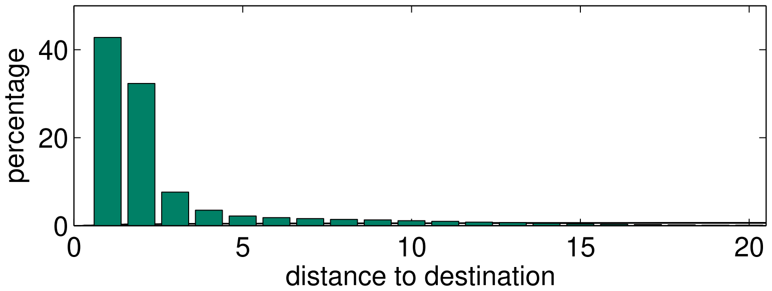
The third data set was collected in September 2004 using 18 measurement sources (see Table 4.1) within a single Tier-1 ISP. Fourteen of these sources are in the US, three are in Europe, and one is in East-Asia. All the sources directly connect to a large Tier-1 ISP via 100Mbps Ethernet links. Since these 18 measurement sources are diversely distributed, compared with the *BtDst* data set, they provide a broader view of bottleneck locations and bandwidth distributions. The measurement destinations for this data set are selected using a method similar to that used for the *BtDst* data set. The difference is that we use a



(a) source-side bottleneck distribution in the *BtSrc* data set



(b) destination-side bottleneck distribution in the *BtDst* data set



(c) bottleneck location distribution in the *ISP* data set

Figure 4.1: Bottleneck location Distribution

different BGP table, which gave us 164,130 IP addresses of which 67,271 are reachable. In this data set, we consider the bottlenecks on all the 67K complete paths. Since we know the measurement sources are well provisioned and the source-side links are very unlikely to be bottlenecks, this data set also captures the destination-side bottleneck distribution. We call this data set the *ISP* data set.

4.1.2 Bottleneck Location Distribution

Figure 4.1 plots the bottleneck link distributions from all three data sets. In each figure, the x-axis is the hop distance from bottleneck links to the end nodes (source nodes in the *BtSrc* data set, destination nodes in the other two), while the y-axis is the percentage of bottlenecks for each hop distance. For the *BtSrc* and the *BtDst* data sets, we also distinguish bottleneck links inside the end ASes (indicated using the dark bars) from those that are not (indicated using the lighter bars).

Figure 4.1(a) shows that 79% of the source-side bottlenecks are within the first four hops, and 74% are within source local ASes. However, note this analysis is limited to 25 source nodes. Figure 4.1(b) shows that 96% of destination-side bottlenecks are on the last four hops and 75% are inside local destination ASes. Since the *BtDst* data set includes measurements from over 67K paths, we believe the conclusion from this figure are representative for the Internet. Figure 4.1(c) confirms the results from the *BtDst* data set: in the *ISP* data set, 86% of destinations have bottlenecks on the last 4 hops, 75% on the last two hops, and 42.7% on the last hop. Although the numbers from different data sets are not exactly the same, they are consistent in the sense that most bottlenecks are on Internet edge. That is, they confirm people's intuition that most of Internet bottlenecks are on Internet edge. For simplicity, since the *ISP* data set has the largest scale, in the rest of this dissertation, we only refer to the results from this data set, e.g., "86% of bottlenecks are within 4 hops from end nodes".

Note that this analysis did not look at how often bottlenecks are on peering links, which is claimed to be another common location for bottlenecks [26]. The reason is mainly the difficulty in identifying the exact locations of inter-AS links, as explained in [56].

4.2 Access-Link Bandwidth Distribution

Given that most Internet bottlenecks are on Internet edge, the bandwidth distribution from these bottlenecks reflects the condition of Internet end-user access speed, which is another important Internet property un-revealed so far. We use the *ISP* data set to study this property. Note that an analysis based on this data set can be biased since we probed one destination per prefix. An implied assumption is that different prefixes have a similar density of real end hosts, which is probably not true. However, given the large number of hosts in the Internet, this is the best approach we can think of.

In the *ISP* data set, since each destination is measured by 18 different sources, we select a bandwidth measurement for each destination that is most representative among those from all 18 sources to use in the following study. This is done by splitting the 18 bandwidth measurements into several groups, and by taking the median value of the largest group as the representative bandwidth. The group is defined as the follows. Let G be a group, and x be a bandwidth value, $x \in G$ iff $\exists y, y \in G, |(x-y)/y| < 0.3$.

Figure 4.2 plots the distribution of the representative available bandwidths for the 67,271 destinations for which Pathneck can measure the last hop. We observe that 40% of destinations have bottleneck bandwidth less than 2.2Mbps, 50% are less than 4.2Mbps, and 63% are less than 10Mbps. These results show that small-capacity links still dominate Internet end-user connections, which are very likely due to small-capacity last-mile links

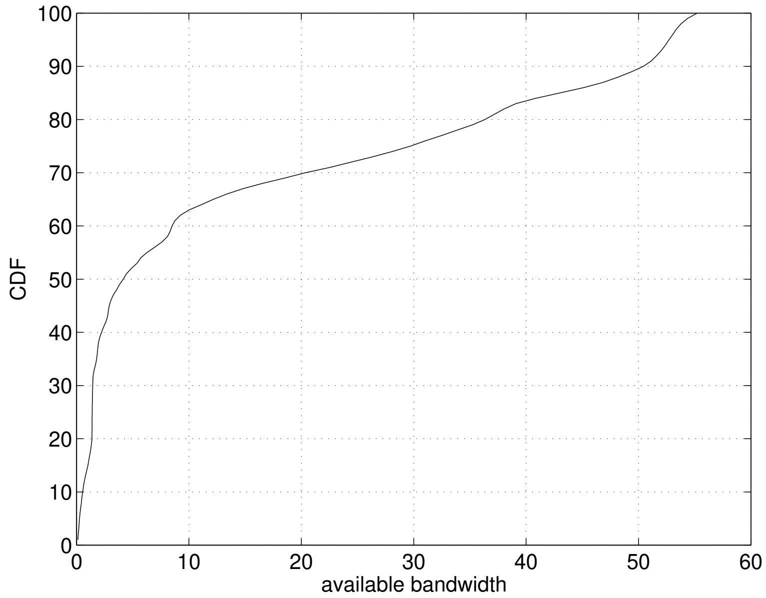


Figure 4.2: Distribution of Internet end-user access bandwidth

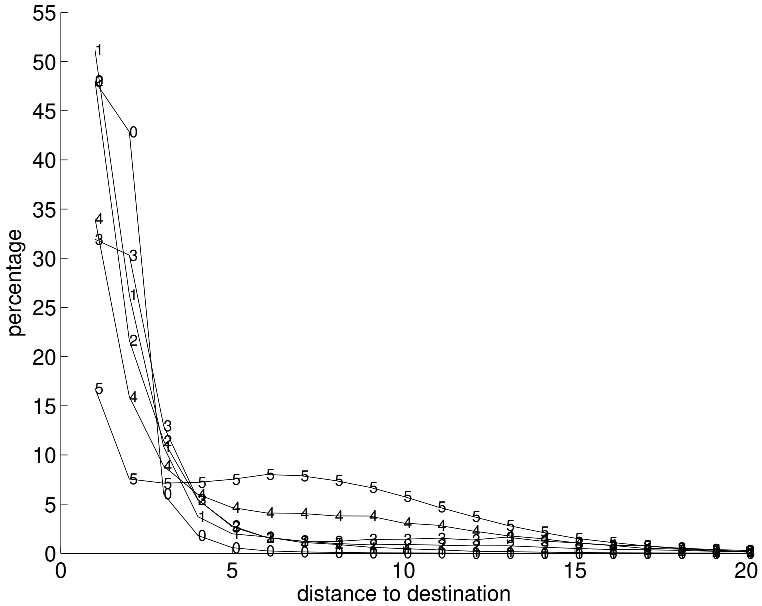


Figure 4.3: Bottleneck distribution for destinations with different available bandwidth

such as DSL, cable-modem, and dial-up links.

For the destinations with bottleneck bandwidth larger than 10Mbps, the bottleneck bandwidth is almost uniformly distribute in the range of [10Mbps, 50Mbps]. The distribution curve in Figure 4.2 ends at 55.2Mbps. This is a bias introduced by our measurement infrastructure. The sending rates from our measurement sources are generally less than 60Mbps, which determines the maximum bottleneck bandwidth that we can detect. For destinations with bottleneck bandwidth higher than 10Mbps, high-bandwidth bottlenecks are more likely to be determined by link load instead of by link capacity, and bottlenecks more frequently appear in the middle of paths. This observation is illustrated in Figure 4.3, where we split the 67,271 destinations into different groups based on bottleneck bandwidth, and plot the distribution of bottleneck locations for each group. The curve marked with "i" represents the group of destinations which have bottleneck bandwidth in the range of $[i * 10Mbps, (i+1) * 10Mbps)$. We can see that while groups $0 \sim 3$ have distributions very similar with that shown in Figure 4.1(c), groups 4 and 5 are clearly different. For example, 62% of destinations in group 5 have bottlenecks that are over 4 hops away from the destinations, where different measurement sources have a higher probability of having different routes and thus different bottlenecks.

4.3 Persistence of Bottleneck Links

By "bottleneck persistence", we mean the fraction of time that the bottleneck is on the same link. Because the bottleneck location of a path is closely tied with its route, in this section, we first look at route persistences before discussing bottleneck persistences. Below we first describe our experimental methodology and how we collect the data.

4.3.1 Experimental Methodology and Data Collection

We study bottleneck persistence from both spatial and temporal perspectives. For the spatial analysis, we conducted *1-day periodic probing*. That is, we selected a set of 960 destinations and probed each of them once per day from a CMU host for 38 days. That provides us 38 sets of probing results for each destination. Here the number of destinations–960– is determined by the length of the probing period (1 day) and the measurement time of Pathneck (90 seconds per destination). This set of data is used throughout this section.

For the temporal analysis, we conducted two more experiments: (1) *4-hour periodic probing*, where we select a set of 160 destinations from those used in the 1-day periodic probing and probe each of them from a CMU host every four hours for 148 hours, obtaining 37 sets of probing results for each destination; and (2) *1-hour periodic probing*, where we select a set of 40 destinations from those used in the 4-hour periodic probing and probe each of them from a CMU host every hour for 30 hours, thus obtaining 30 sets of probing results for each destination. These two data sets are only used in Section 4.3.4.

Table 4.2: Determining co-located routers

Heuristic	*# IP pairs*
Same DNS name	42
Alias	53
CMU or PSC	16
Same location in DNS name	572
Digits in DNS name filtered	190
Real change	1722

4.3.2 Route Persistence

In the 1-day periodic probing data set, we observe quite a few IP level route changes: among the 6,868 unique IP addresses observed in this data set, 2,361 of them are associated with hops whose IP address changes, i.e., the route appears to change. This shows that we must consider route persistence in the bottleneck persistence analysis. Intuitively, Internet routes have different persistence properties at different granularity, so in the following, we investigate route persistence at both the location level and the AS level. At the *location level*, we consider hops with IP addresses that belong to the same router or co-located routers as the same hop. We will explain what we mean by the "same router" or "co-located router" below. Location-level analysis can help us reduce the impact of "false" route changes. At the *AS level*, we consider all hops in the same AS as the same AS-level hop; this is done by mapping the IP address of each hop to its AS number using the mapping provided by [81].

Location-Level Route

At the location level, the IP addresses associated with the same router are identified using two heuristics. First, we check the DNS names. That is, we resolve each IP address into its DNS name and compare the DNS names. If two IP addresses *(a)* have the same hop position *(b)* for the same source-destination pair and *(c)* are resolved to the same DNS name, they are considered to be associated with the same router. We found that 5,410 out of the 6,868 IP addresses could be resolved to DNS names, and 42 pairs of IP addresses resolve to identical DNS names (refer to Table 4.2). Second, we look for IP aliases. For the unresolved IP addresses, we use Ally [109] to detect router aliases. We found that 53 IP pairs are aliases.

The IP addresses associated with co-located routers are identified by applying the following heuristics sequentially.

1. *CMU or PSC*. Because all our measurements are conducted from a CMU host, they always pass through PSC (`www.psc.edu`) before entering other networks, so we consider all those routers within CMU or PSC as co-located.

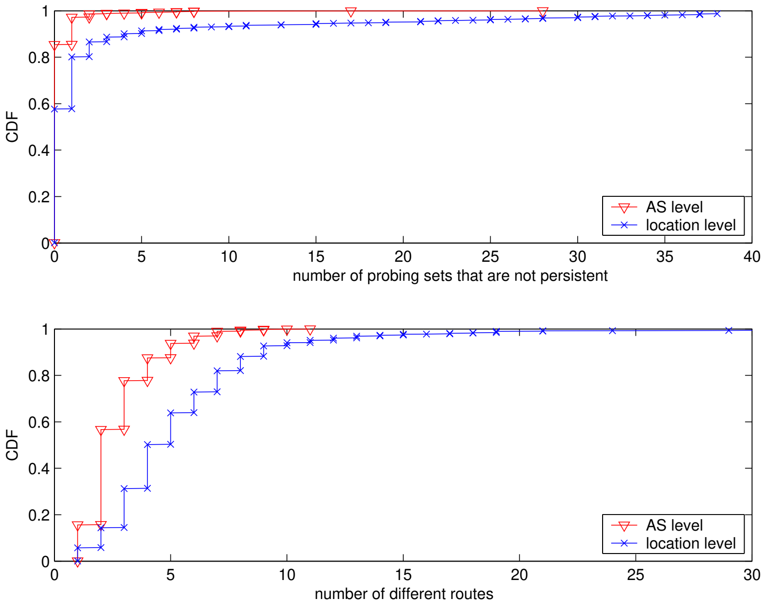


Figure 4.4: Route persistence at the location level and AS level

2. *Same location in DNS name.* As pointed out in [121], the DNS names used by some ISPs (e.g., the `*.ip.att.net` for AT&T and the `*.sprintlink.net` for Sprint) include location information, which allows us to identify those routers that are at the same geographical position.

3. *Digits in DNS name filtered.* We remove the digits from DNS names. If the remaining portion of the DNS names become identical, we consider them to be co-located.

These three heuristics allow us to identify 16, 572, and 190 pairs of co-located routers, respectively. Note that heuristics (2) and (3) are not perfect: stale information in DNS can cause mistakes in heuristic (2), while heuristic (3) is completely based on our limited knowledge of how ISPs assign DNS names to their IP addresses. Although we think the impact from these errors is small, better tools are needed to identify co-located IP addresses.

At the location level, we consider a route change only when the corresponding hops do not belong to the same or a co-located routers. Table 4.2 shows that 1,722 pairs of IP addresses are associated with hops that experience route changes. Given this definition for location-level route change, we define a *persistent probing set* as a probing set where the route remains the same during the 10 probings.

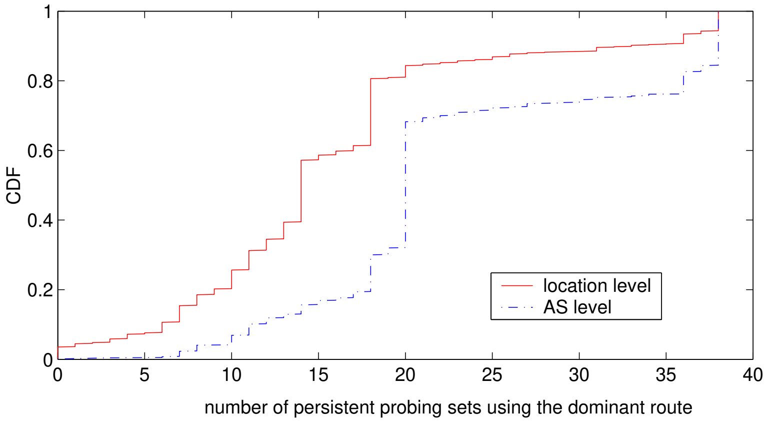


Figure 4.5: Frequency of the dominant route

Route Persistence Results on Both Location Level and AS Level

Figure 4.4 shows the route persistence results for the 1-day periodic probing, at both the location and AS level. The top graph plots the cumulative distribution of the number of probing sets that are not persistent. As expected, AS-level routes are more persistent than location-level routes. Some location-level routes change fairly frequently. For example, about 5% of the source-destination pairs have more than 15 (out of 38) probing sets that are not persistent at the location level. However overall, the vast majority of the routes are fairly persistent in the short term: at the location level, 57% of the source-destination pairs have perfect persistence (i.e., all probing sets are persistent), while 80% have at most one probing set that is not persistent. The corresponding figures for AS level are 85% and 97%, respectively.

The bottom graph in Figure 4.4 illustrates long-term route persistence by plotting the distribution of the number of different location-level and AS-level routes that a source-destination pair uses. We observe that only about 6% of the source-destination pairs use one location-level route, while about 6% of the source-destination pairs have more than 10 location-level routes (for 380 probings). The long-term route persistence at the location level is quite poor. However, at the AS level, not surprisingly, the routes are much more persistent: 94% of the source-destination pairs have fewer than 5 different AS-level routes.

We have seen that most of the source-destination pairs use more than one route. For our bottleneck persistence analysis, we need to know if there is a dominant route for a source-destination pair. Here, the *dominant route* is defined as the route that is used by the highest number of persistent probing sets in all 38 probing sets for the same source-destination pair. Figure 4.5 shows the distribution of the dominant route for each source-destination pair, i.e., the number of persistent probing sets that use the dominant route. We can see that, at the location level, only around 15% of the source-destination pairs have a route with a frequency of 20 or more (out of 38), i.e., the “dominant” routes are usually not very dominant. At the AS level, for about 30% of the source-destination pairs, the dominant route is used by less than 20 (out of 38) probing sets. This is consistent with the observation in [121] that a total of about 1/3 of Internet routes are short lived (i.e., exist for less than one day).

4.3.3 Bottleneck Spatial Persistence

We study spatial bottleneck persistence from two points of view: the route view and the end-to-end view. The route-view analysis provides the bottleneck persistence results excluding the effect of route changes, while end-to-end view can tell us the bottleneck persistence seen by a user, including the effect of route changes. The comparison between these two views will also illustrate the impact of route changes. In each view, the analysis is conducted at both the location and the AS level. A bottleneck is persistent at the location level if the bottleneck routers on different routes for the same source-destination pair are the same or co-located. A bottleneck is persistent at the AS level if the bottleneck routers on different routes for the same source-destination pair belong to the same AS.

Route View

In the route view, bottleneck persistence is computed as follows. We first classify all persistent probing sets to the same destination into different groups based on the route that each probing set follows. In each group, for every bottleneck router detected, we count the number of persistent probing sets in which it appears (cnt), and the number of persistent probing sets in which it appears as a bottleneck (bot). Then the bottleneck persistence is defined as bot/cnt. To avoid the bias due to small cnt, we only consider those bottlenecks where $cnt \geq 10$. The number "10" is selected based on Figure 4.5, which shows that over 80% (95%) of the source-destination pairs have a dominant route at the location level (AS level) with a frequency higher than 10; also, picking a larger number will quickly reduce the number of source-destination pairs that can be used in our analysis. Therefore, 10 is a good trade-off between a reasonably large cnt and having a large percentage of source-destination pairs to be used in the analysis.

In Figure 4.6, the two bottom curves (labeled with "route view") plot the cumulative distribution of the bottleneck persistence. We can see that, at both the location level and AS level, around 50% of bottlenecks have persistence larger than 0.7, and over 25% of them have perfect persistence. This shows that most of the bottlenecks are reasonably persistent in the route view. Note that the location-level curve and the AS-level curve are almost identical. This seems to contradict the intuition that bottlenecks should be more persistent at the AS level. Note however that for a source-destination pair, cnt in the AS level can be larger than that for the location level, so we cannot directly compare the persistence at these two levels.

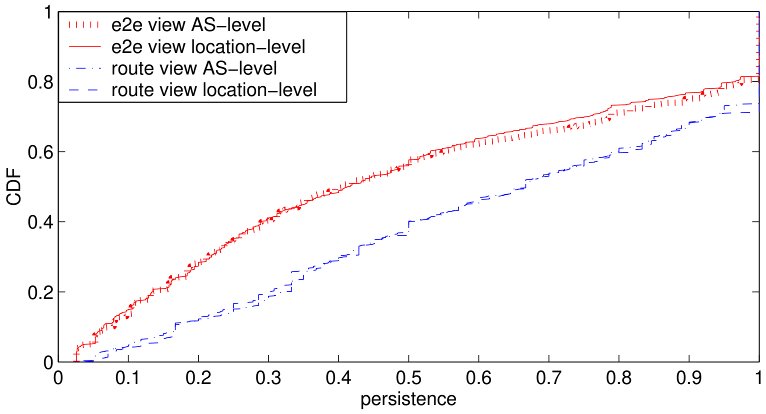


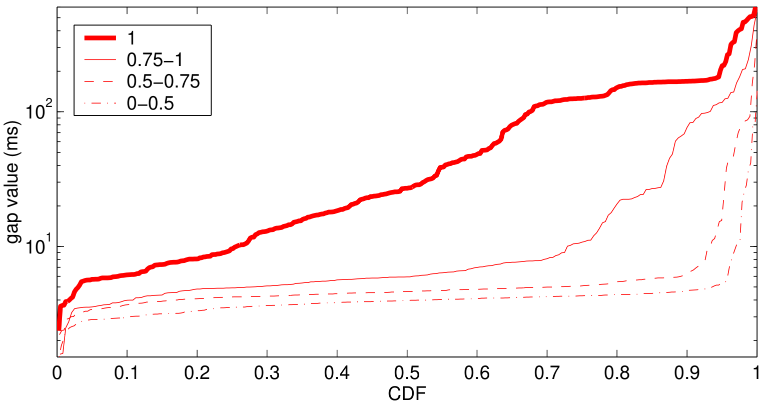
Figure 4.6: Persistence of bottlenecks.

End-To-End View

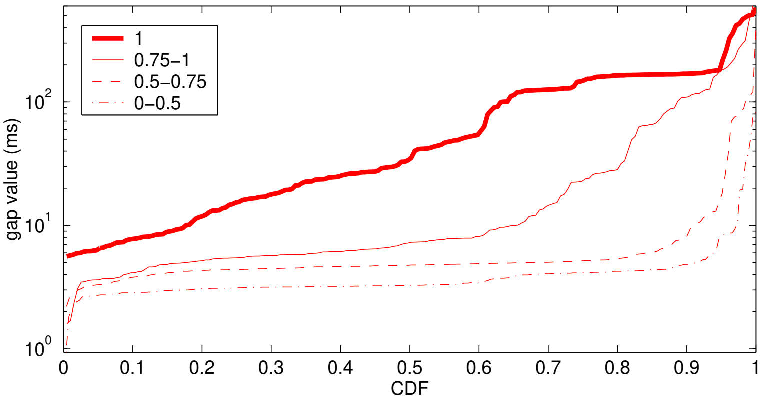
In this view, we consider bottleneck persistence in terms of source-destination pairs, regardless of the route taken. We compute bottleneck persistence of end-to-end view in a way similar with that of route view. The two top curves (labeled with "e2e view") in Figure 4.6 show the results for end-to-end bottleneck persistence. Again, the results for location level and AS level are very similar. However, the persistence in the end-to-end view is much lower than that in the route view – only 30% of bottlenecks have persistence larger than 0.7. This degradation from that in the route view illustrates the impact of route changes on bottleneck persistence.

Relationship With Gap Values

For those bottlenecks with high persistence, we find that they tend to have large gap values in the Pathneck measurements. This is confirmed in Figure 4.7, where we plot the relationship between the bottleneck gap values and their persistence values in both the route view and end-to-end view. We split the bottlenecks that are included in Figure 4.6 into 4 groups based on their persistence value: 1, [0.75, 1), [0.5, 0.75), and [0, 0.5), and then plot the cumulative distribution for the average bottleneck gap values in each group. We observe a clear relationship between large gap values and high persistence in both the route view (top figure) and end-to-end view (bottom figure). The reason is, as discussed in [56], that a larger gap value corresponds to smaller available bandwidth, and the smaller the available bandwidth, the less likely it is that there will be a hop with a similar level of available bandwidth on the path between a source-destination pair, so the bottleneck is more persistent.



(a) Route view



(b) End-to-end view

Figure 4.7: Bottleneck gap value vs. bottleneck persistence

4.3.4 Bottleneck Temporal Persistence

So far our analysis has focused on the 1-day periodic probing results, which provide only a coarse-grained view of bottleneck persistence. The 4-hour and 1-hour periodic probings described early in this section allow us to investigate short-term bottleneck persistence. Although these two sets of experiments only cover a small number of source-destination pairs, it is interesting to compare their results with those in the 1-day periodic probings.

Figure 4.8 compares location-level route persistence over 1- hour, 4-hour, and 1-day time periods. In the top graph, the x-axis for the 1-hour and 4-hour curves are scaled by 38/30 and 38/37 to get a fair comparison with the 1 day curve. For the 4-hour and 1-day periodic probings, the number of probing sets that are not persistent are very similar, while those for 1-hour periodic probing show a slightly higher percentage of probing sets that are not persistent. This seems to imply that there are a quite a few short-term route changes that can be caught by 1-hour periodic probings but not by 4-hour periodic probings. The bottom figure shows that the number of different routes for 1-day periodic probings is

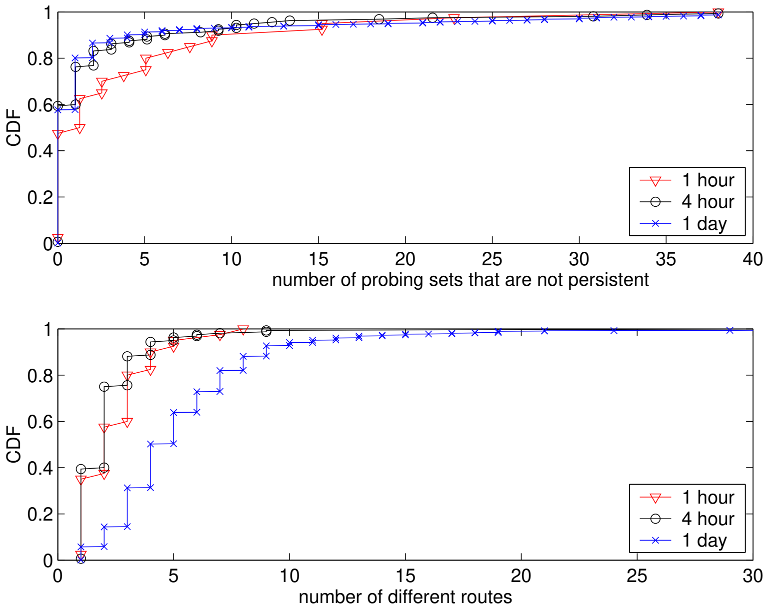


Figure 4.8: Location-level route persistence.

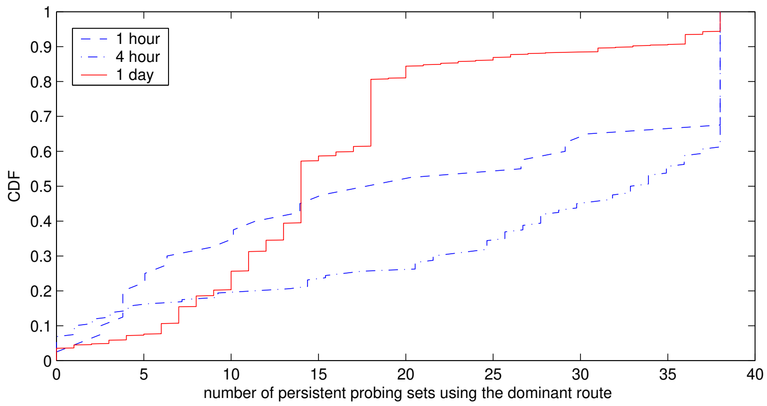


Figure 4.9: Distribution of the dominant route at the location level.

significantly larger than those for 4-hour and 1-hour periodic probings. We think this is mainly because the 1-day periodic probings cover a much longer period.

Figure 4.9 plots the distribution of the dominant route at the location level. Clearly, in the 4-hour and 1-hour periodic probings, the dominant routes cover more persistent probing sets than for the 1-day periodic probings — in the 4-hour and 1-hour periodic probings, 75% and 45% of the source-destination pairs have over 20 persistent probing sets that use the dominant routes, while only around 20% of the source-destination pairs in the 1-day periodic probings use the dominant routes. Note that the 4-hour periodic

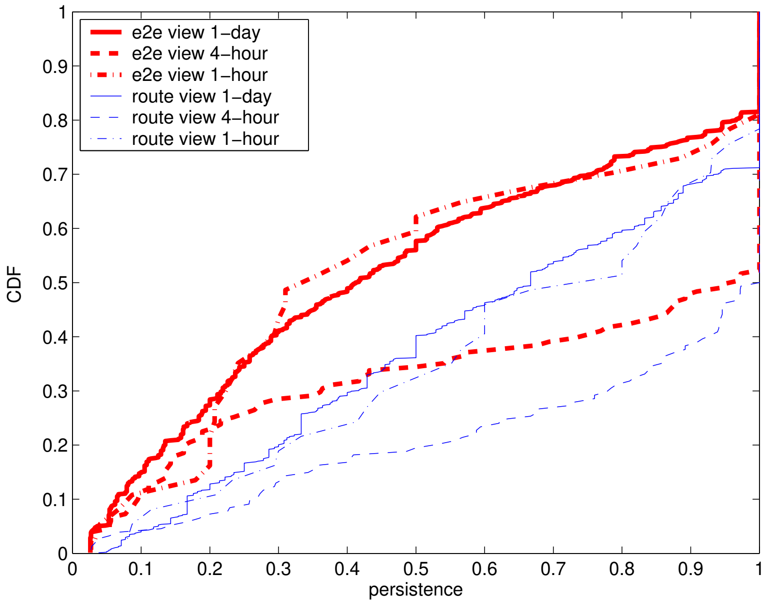


Figure 4.10: Persistence of the bottlenecks with different measurement periods at the location level.

probing results have the largest dominant route coverage. A possible reason is that the 1-day periodic probings last much longer and allow us to observe more route changes, while the 1-hour periodic probings can catch more short-term route changes. The same explanation can also explain the difference in bottleneck persistence plotted in Figure 4.10, which compares the location-level bottleneck persistence for different probing periods. Again, we see that the 1-day and 1-hour curves are closer to each other in both the route view and the end-to-end view, while the 4-hour curves stand out distinctly, with higher persistence. This is because the 4-hour periodic probings have the best dominant route coverage, so route changes have the least impact.

4.3.5 Discussion

The analysis in this section shows that 20% – 30% of the bottlenecks have perfect persistence. As expected, bottlenecks at the AS level are more persistent than bottlenecks at the location level. Long-term Internet routes are not very persistent, which has a significant impact on the bottleneck persistence. That is, people will reach different conclusions about bottleneck persistence depending on whether or not route changes are taken into account. We also confirm that bottlenecks with small available bandwidth tend to be more persistent. Finally, we show that bottleneck persistence is also sensitive to the length of the time period over which it is defined, and the worst persistence results seem to occur for medium time periods. Note that these results are based on measurements from 960 or

Table 4.3: Different types of paths in the 954 paths probed

	No loss	Loss	Total
No bottleneck	139	121	260
Bottleneck	312	382	694
Total	451	503	954

fewer paths, which is a relatively small number of paths compared to the number of paths that are used in the previous study. In future work, we hope to alleviate this limitation by using more measurement machines.

4.4 Relationship with Link Loss and Delay

In this section, we investigate whether there is a clear relationship between bottleneck and link loss and delay. Since network traffic congestion may cause queueing and packet loss, we expect to see that bottleneck links are more likely to experience packet loss and queueing delay. On the other hand, capacity determined bottlenecks may not experience packet loss. Therefore, the relationship between bottleneck position and loss position may help us to distinguish load-determined and capacity-determined bottlenecks.

4.4.1 Data Collection

In this study, we use Tulip [80] to detect the packet loss position and to estimate the link queueing delay. We probed 954 destinations from a CMU host. For each destination, we did one set of Pathneck probings, i.e., 10 RPT probing trains, followed by a Tulip loss probing *and* a Tulip queuing probing. Both types of Tulip probings are configured to conduct 500 measurements for each router along the path [22]. For each router along the path, Tulip provides both the round trip loss rate and forward path loss rate. Because Pathneck can only measure forward path bottlenecks, we only consider the forward path loss rate. Table 4.3 classifies the paths based on whether or not we can detect loss and bottleneck points on a path.

4.4.2 Relationship with Link Loss

Let us first look at how the positions of the bottleneck and loss points relate to each other. In Figure 4.11, we plot the distances between loss and bottleneck points for the 382 paths where we observe both a bottleneck and loss points. In the top figure, the x-axis is the normalized position of a bottleneck point — the normalized position of a hop is defined to be the ratio between the hop index (the source node has index 1) and the length of the whole path. The y-axis is the relative distance from the closest loss point to that bottleneck point.

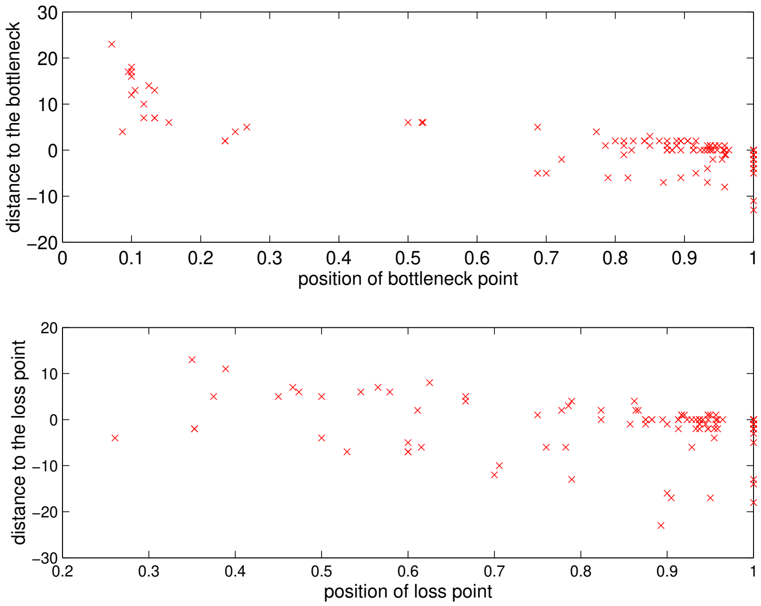


Figure 4.11: Distances between loss and bottleneck points.

If there is a loss point with equal distance on each side, we plot both, one with a positive distance, and the other with a negative distance. Positive distance means that the loss point has a larger hop index, i.e., it is downstream from the bottleneck point; negative distance means that the loss point is earlier in the path than the bottleneck point. The bottom figure presents the data from the loss point of view, and the distance is computed from the closest bottleneck point. Figure 4.11 clearly shows that there are fewer bottleneck points in the middle of the path, while a fair number of loss points appear within the normalized hop range [0.3, 0.9]. On the other hand, there are fewer loss points in the beginning of the path.

Figure 4.12 shows the cumulative distribution of the distance from the closest loss point to each bottleneck points, using the same method as that used in the top graph of Figure 4.11. We observe that over 30% of bottleneck points also have packet loss, while around 60% of bottleneck points have a loss point no more than 2 hops away. This distance distribution skews to the positive side due to the bottleneck clustering at the beginning of the path, as shown in Figure 4.11.

4.4.3 Relationship with Link Delay

Besides packet loss, queueing delay is another metric that is frequently used as an indication of congestion. Tulip provides queueing delay measurements as the difference between the median RTT and the minimum RTT from the probing source to a router. Note that the queueing delay computed this way corresponds to the cumulative queueing delay from the probing source to a router, including delay in both the forward and return path. The 500

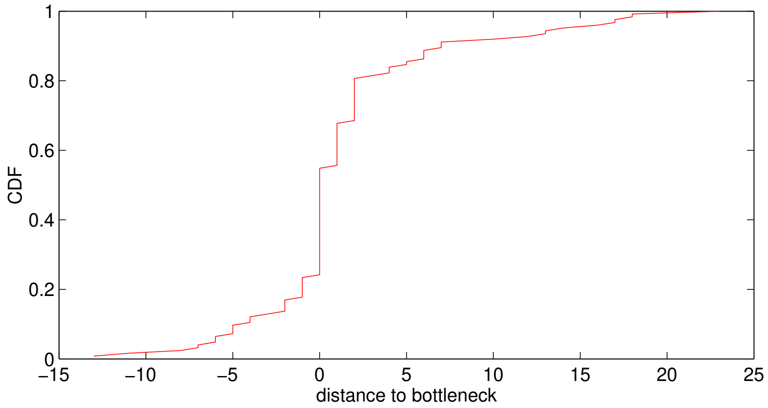


Figure 4.12: Distance from closest loss point to each bottleneck points

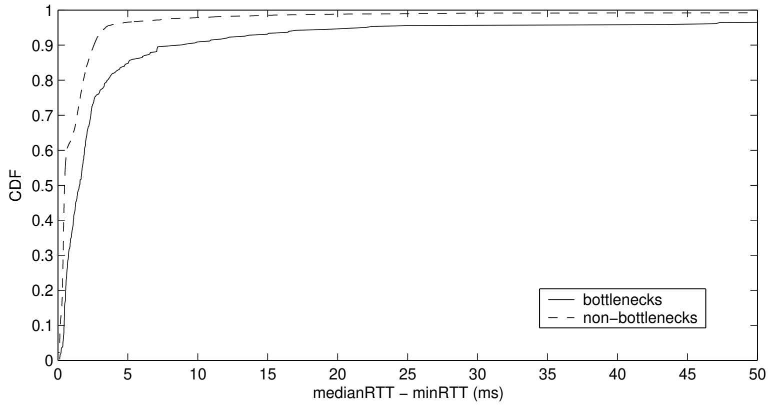


Figure 4.13: Bottlenecks vs. queueing delay

measurements for each router in our experiment can provide a reasonable estimate for this queueing delay. Based on these measurements, we look at the relationship between the bottlenecks and the corresponding queueing delays.

Figure 4.13 shows the cumulative distribution of the queueing delays for bottleneck and non-bottleneck links. In our experiment, we observe queueing delays as large as $900ms$, but we only plot up to $50ms$ in the figure. As expected, we tend to observe longer queueing delays at bottleneck points than at non-bottleneck points: fewer than 5% of the non-bottleneck links have a queue delay larger than $5ms$, while around 15% of the bottleneck links have a queue delay larger than $5ms$. We also observe the same relationship between the loss points and their queueing delays, i.e., we tend to observe longer queueing delay at the loss points. Note that these results are still preliminary since it is unclear to what degree the tulip delay measurements (in terms of RTT delay variance) can be used to quantify link queueing delay.

4.5 Applications of Bottleneck Properties

The studies presented in the previous four sections significantly improved our understanding about Internet bottlenecks, the insights also help us improve the performance of some important applications. First, the results in Figure 4.3 suggest that an end node may experience different bottlenecks for paths connecting with different peer nodes, so a distributed system like a CDN may be able to help the end node avoid the worst bottleneck. Second, with the end-user access-bandwidth distribution, a web site can obtain its transmission-time distribution. In this section, I will present detailed analyses about these two applications.

4.5.1 Improving End-User Access Bandwidth Using CDN

In this section, we study how Internet bottleneck properties can be used to improve the performance of content distribution networks (CDNs). The idea of a content distribution network is to replicate data in a set of servers distributed diversely over the Internet so that a client request can be redirected to the server that is closest to the client and hopefully can achieve the best performance. During the redirection procedure, most CDNs focus on reducing the network delay; few have looked at bandwidth performance due to the difficulty of obtaining bandwidth estimation. Therefore, there is not a good understanding on how well a CDN can improve bandwidth performance of end users. On the other hand, the results in Figure 4.3 suggest that an end-node may experience different bottlenecks for paths connecting with different servers. This implies that it is possible for a CDN to also improve clients' bandwidth performance. In this section, we quantify the extent of such improvement by viewing the system composed of the measurement nodes listed in Table 4.1 as a CDN. In the following, we first describe the redirection algorithm, we then present the analysis results. The analysis is based on the *ISP* data set described in Section 4.1.1.

The Greedy Algorithm

To optimize client performance using replicas, we need to know how many replicas we should use and where they should be deployed. The goal is that the selected set of replicas should have performance very close to that achieved by using all replicas. A naive way is to consider all the possible combinations of replicates when selecting the optimal one. Given that there are $2^{18} - 1$ (i.e., 262,143) different combinations for 18 replicas, and each replica measures over 160K destinations, the time of evaluating all combinations is prohibitively high. Therefore, we use a greedy algorithm, which is based on a similar idea as the greedy algorithm used by Qiu et.al. [98]. This algorithm only needs polynomial processing time, but can find a sub-optimal solution very close to the optimal one. We now explain how to apply this algorithm on the *ISP* data set.

The intuition behind the greedy algorithm is to always pick the best replica among the available replicas. In this algorithm, the "best" is quantified using a metric called *marginal utility*. This metric is computed as follows. Assume that at some point, some replicas have already been selected. The best bandwidth from among the selected replica to each destination is given by $\{bw_i | 0 \leq i < N\}$, where $N \approx 160K$ is the number of destinations, and bw_i is the largest bandwidth that is measured by one of the replicas already selected to destination i. Let $\{sbw_i | 0 \leq i < N\}$ be the sorted version of $\{bw_i\}$. We can now compute bw_sum as:

$$bw\_sum = \sum_{k=0}^{99} sbw_{index(k)}$$

where

$$index(k) = \frac{N \times (k+1)}{101} - 1$$

There are two details that need to be explained in the above computation. First, we cannot simply add all the individual measurements when calculating bw_sum. This is because by definition, a destination is not necessarily reached by all the measurement sources, so introducing a new replica could bring in measurements to new destinations, thus changing the value of N. Therefore, we cannot simply add all bw_i since the results would be incomparable. In our analysis, we add 100 values that are evenly distributed on the CDF curve. Here, the number "100" is empirically selected as a reasonably large value to split the curve. Second, we split the curve into 101 segments using 100 splitting points, and only use the values of these 100 splitting points. That is, we do not use the two end values—sbw_0 and sbw_{N-1}, whose small/large values are very probably due to measurement error.

Suppose a new replica A is added, many bw_i can change, and we compute a new bw_sum_A. The marginal utility of A is then computed as:

$$marginal\_utility = \frac{|bw\_sum - bw\_sum_A|}{bw\_sum}$$

With this definition, the replica selected in each step is the one that has the largest marginal utility. For the first step, the replica selected is simply the one that has the largest bw_sum. In the end, the algorithm generates a *replica selection sequence*:

$$v_1, v_2, ..., v_{18}$$

where $v_i \in \{S01, S02, ..., S18\}$. To select $m(< 18)$ replicas, we can simply use the first m replicas in this sequence. This greedy algorithm has polynomial processing time, but only gives a sub-optimal solution, although it is very close to optimal [57].

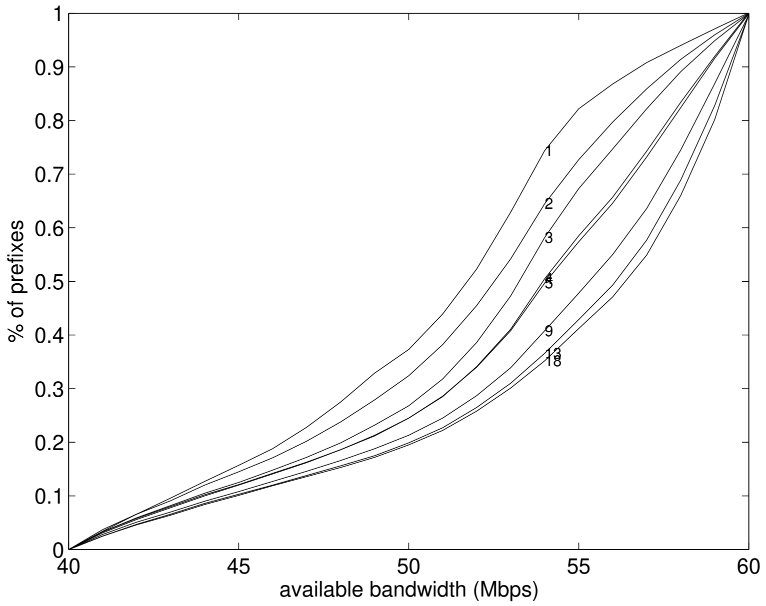


Figure 4.14: Bandwidth optimization for well provisioned destinations.

Bandwidth Improvement Using CDN

Using the greedy algorithm, we did two studies based on the ISP data set. In the first study, we only use the 23,447 destinations where Pathneck can measure the last hop, and the bottleneck bandwidth is higher than 40Mbps. In the second, we increase the scale by including more measurement results.

The results of the first study are shown in Figure 4.14, the numbers marked on the curves are the the numbers of replicas. It shows the cumulative distribution of path bandwidth upper-bounds with varying numbers of replicas. We can see that the bandwidth improvement from replicas is indeed significant. For example, with a single replica, there are only 26% paths that have bandwidth higher than 54Mbps, while with all 18 replicas, the percentage increases to 65%.

An obvious problem for the first study is that it only covers around 16% of the paths that we measured, thus the results could be biased. Therefore we did the second and more general study, where we use join nodes instead of real destination in our study. Join nodes are defined as follows. For those unreachable destinations, we only have partial path information. Ideally, the partial paths would connect the measurement sources to a common *join node* that is the last traced hop from each source, so the join node stands in for the destination in a consistent way. Unfortunately, such a join node may not exist, either because there may not be a node shared by all the 18 measurement sources to a given destination or the shared hop may not be the last traced hop. For this reason, we relax the constraints for the join node: the join node only needs to be shared by 14 measurement

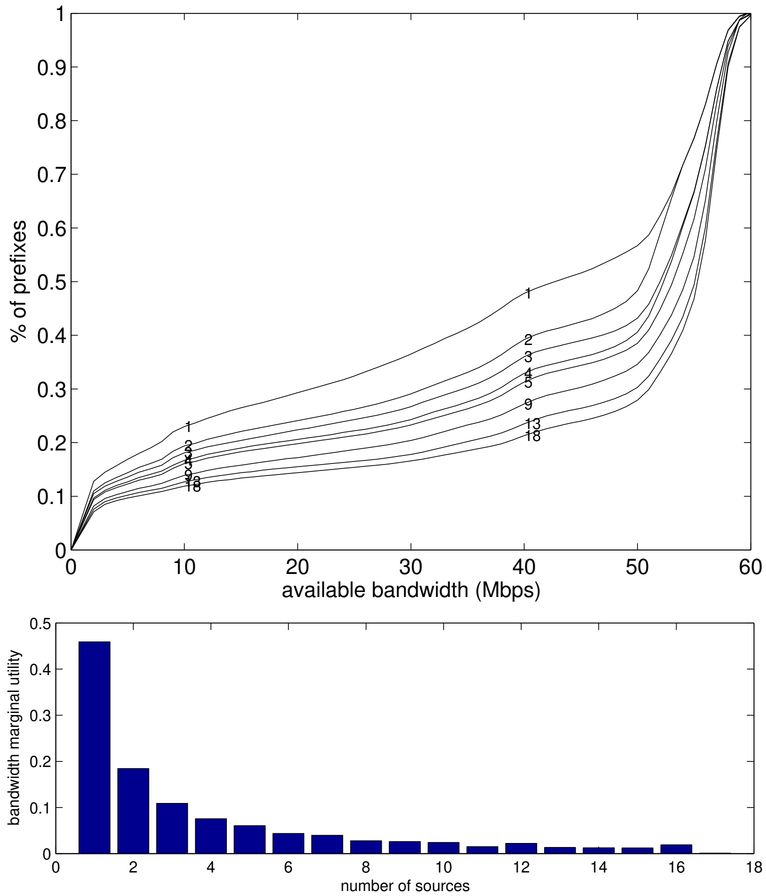


Figure 4.15: Bandwidth optimization for all valid destinations.

sources, and it need only be within three hops from the last traced hop. Here "14" is simply a reasonably large number that we choose. This allows us to make the best of the available data while still having comparable data. Specifically, this allows us to include over 141K destinations (86% of the total) in our analysis, among which 47% are real destinations, and 53% are join nodes. We refer to these 141K destinations as *valid destinations*.

Now we apply the greedy algorithm on all the valid destinations, but exclude last-mile bottlenecks. In other words, we do a "what if" experiment to study what would happen if last-mile links were upgraded. That is, for those destinations where Pathneck can measure the last hop, we remove the last two hops; this is based on the observation in Figure 4.1(c) that 75% paths have bottlenecks on the last two hops. For the others, we use the raw measurement results. Figure 4.15 includes the results from the greedy algorithm when considering all destinations. Table 4.4 lists the replica selection sequence from the greedy algorithm, and the marginal utility from each replica. We can see that the bandwidth improvement spreads almost uniformly in the large range [5Mbps, 50Mbps]. If using

Table 4.4: Replica selection sequence

select seq.	1	2	3	4	5	6	7	8	9
node id	S03	S17	S04	S16	S12	S15	S01	S05	S18
location	SW	Eu	MW	Eu	NE	Eu	NE	SM	As
select seq.	10	11	12	13	14	15	16	17	18
node id	S07	S13	S10	S08	S14	S11	S02	S09	S06
location	SW	NM	NW	MM	NE	NM	SM	NE	SE

NE: northeast, NW: northwest, SE: southeast, SW: southwest
ME: middle-east, MW: middle-west, MM: middle-middle
Eu: Europe, As: East-Asia

5% as the threshold for marginal utility, only the first two replicas selected significantly contribute to the bandwidth performance improvement. Also geographic diversity does not play an important role.

4.5.2 Distribution of Web Data Transmission Time

Pathneck measurements provide both bandwidth and delay information. That makes it possible to study the distribution of data transmission times of network services like web services. This is because end-to-end data transmission times are determined by delay, available bandwidth, and data size altogether, and a web server can easily know their data size distribution. In this section, we use the bandwidth and the delay information provided by Pathneck to study how well replicated hosting can be used to improve a web server's data transmission times. Below we first provide a simplified TCP model to characterize data transmission time as a function of available bandwidth, delay, and data size. We then look at the transmission-time distribution for different data sizes with different number of replicas.

Simplified TCP Throughput Model

Simply speaking, TCP data transmission includes two phases: slow-start and congestion avoidance [63]. During slow-start, the sending rate doubles every roundtrip time, because the congestion window exponentially increases. During congestion avoidance, the sending rate and the congestion window only increase linearly. These two algorithms, together with the packet loss rate, can be used to derive an accurate TCP throughput model [89]. However, we can not use this model since we do not know the packet loss rate, which is very expensive to measure. Instead, we build a simplified TCP throughput model that only uses bandwidth and RTT information.

Our TCP throughput model is as follows. Let the path under considered have available bandwidth abw (Bps) and roundtrip time rtt (second). Assume that the sender's TCP congestion window starts from 2 packets and that each packet is 1,500 bytes. The congestion window doubles every RTT until it is about to exceed the bandwidth-delay product of the path. After that, the sender sends data at a constant rate of abw. This transmission algorithm is sketched in the code segment shown below. It computes the total transmission time (t_{total}) for x bytes of data along a path.

```
cwin = 2 * 1500;
t_ss = 0;  t_ca = 0;

while (x > cwin && cwin < abw * rtt) {
    x -= cwin;
    cwin *= 2;
    t_ss += rtt;
}
if (x > 0) t_ca = x / abw;

t_total = t_ss + t_ca;
```

where t_ss and t_ca are the transmission time spent in the slow-start phase and the congestion avoidance phase, respectively. We say the data transmission is *rtt-determined* if $t\_ca = 0$. We can easily derive the maximum rtt-determined data size as

$$2^{\lfloor log_2(abw*rtt/1500)\rfloor+1} * 1500(byte)$$

In the following, we call this size the *slow-start size*. Clearly, when the data is less than the slow-start size, it is rtt-determined.

This model ignores many important parameters that can affect TCP throughput, including packet loss rate and TCP timeout. However, the purpose of this analysis is not to compute an exact number, but rather to provide a guideline on the range of data sizes where RTT should be used as the optimization metric in replica hosting.

Slow-Start Sizes and Transmission Times

Using the above model, we compute the slow-start sizes for the 67,271 destinations in the *ISP* data set for which Pathneck can obtain complete measurements. Figure 4.16 plots the distributions of slow-start sizes for the paths starting from each replica. Different replicas have fairly different performance; differences are as large as 30%. Overall, at least 70% of paths have slow-start sizes larger than 10KB, 40% larger than 100KB, and around 10% larger 1MB. Given that web pages are generally less than 10KB, it is clear that their transmission performance is dominant by RTT and replica placement should minimize RTT. For data sizes larger than 1MB, replica deployment should focus on improving bandwidth.

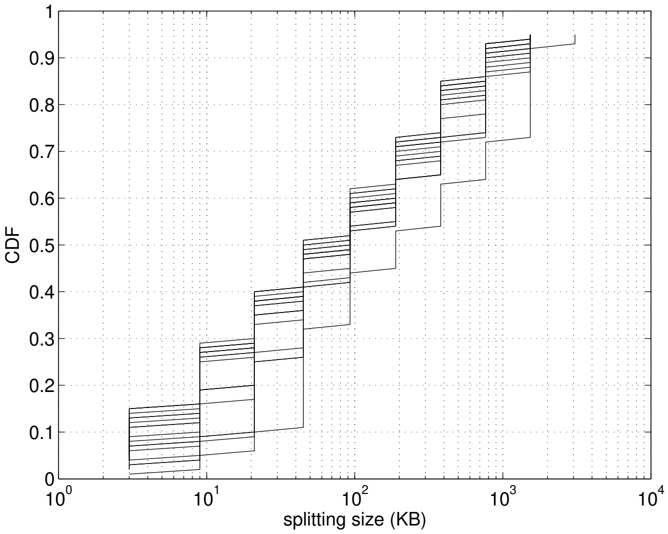


Figure 4.16: Cumulative distribution of the slow-start sizes

To obtain concrete transmission times for different data sizes, we use our TCP throughput model to compute data transmission time on each path for four data sizes: 10KB, 100KB, 1MB, and 10MB. We then use the greedy algorithm to optimize data transmission times. The four subgraphs in Figure 4.17 illustrate the transmission-time distributions for each data size using different number of replicas. These figures are plotted the same way as that used in Figure 4.14. If we focus on the 80 percentile values when all 18 replicas are used, we can see the transmission times for 10KB, 100KB, 1MB and 10MB are 0.4 second, 1.1 second, 6.4 second, and 59.2 second, respectively. These results are very useful for Internet-scale network applications to obtain an intuitive understanding about their data transmission performance.

4.5.3 More Applications

Besides the applications on CDNs and web data transmissions, as a collaboration with Zhuoqing Morley Mao, we also studied how well overlay routing and multihoming can be used to avoid bottlenecks and improve end-user data transmission performance [56]. We found that both methods are effective in avoiding bottlenecks. In our experiments, 53% of overlay paths and 78% of multihoming paths were able to improve end-user bandwidth performance.

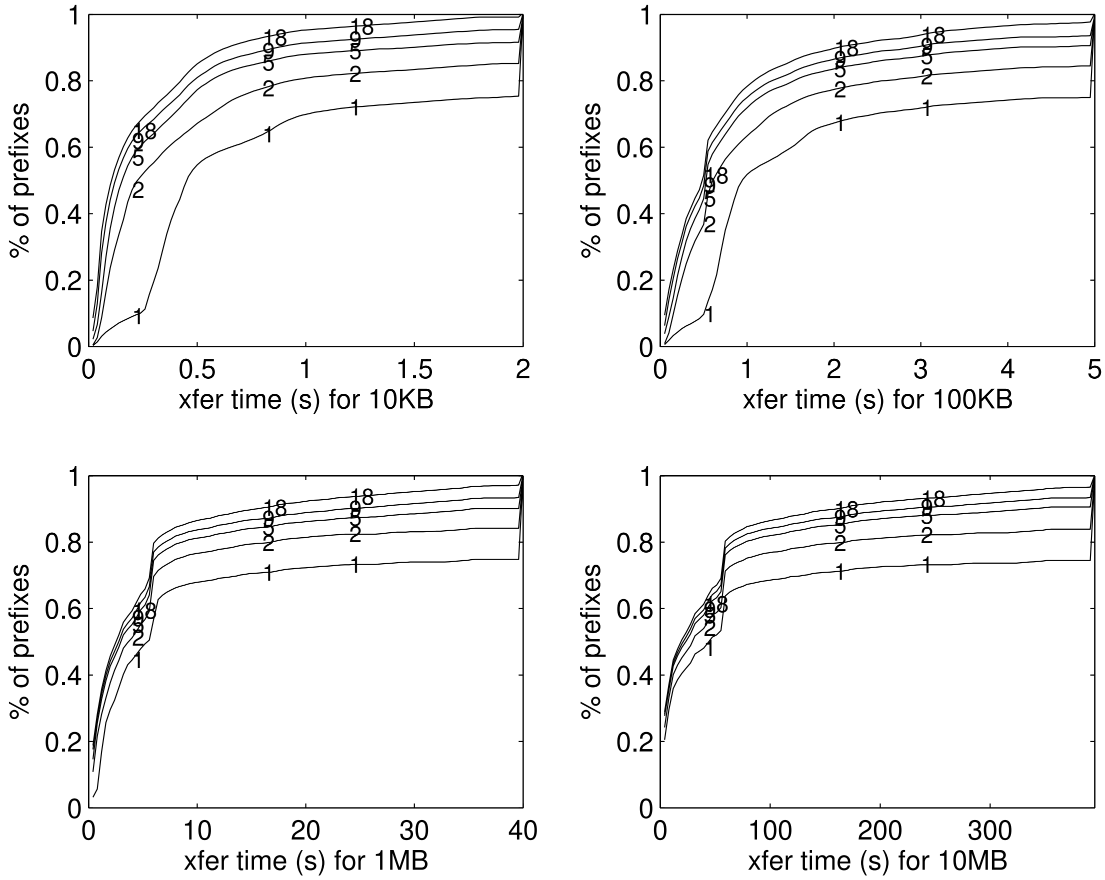


Figure 4.17: Transmission times for different data sizes

4.6 Related Work

Due to lack of efficient bottleneck locating techniques, there has not been a lot of work on characterizing Internet bottlenecks. To the best of our knowledge, the only previous study is from [26], which shows that Internet bottlenecks are mostly on edge and peering links. That conclusion, however, is limited by the scale of its experiments. Our Pathneck based study overcomes this limitation by looking at bottleneck properties at an Internet scale. Furthermore, our study is more extensive by looking at more important bottleneck properties.

The other group of related work is on the persistence of Internet path properties, such as route, delay, loss rate, and throughput. Labovitz et.al. [74, 75, 76] showed that a large fraction of IP prefixes had persistent routes from many observation points, despite the large volume of BGP updates. Rexford et.al. [99] discovered that the small number of popular destinations responsible for the bulk of Internet traffic had very persistent BGP routes. Zhang et.al. [121, 120] showed that Internet routes appear to be very persistent although some routes exhibited substantially more changes than others; packet loss rate and end-to-end throughput were considerably less stationary. Although none of these studies are directly on bottleneck link persistence, their insights are helpful in our persistence analysis.

4.7 Summary

This chapter presented an Internet-scale measurement study on bottleneck properties. We showed that (1) over 86% of Internet bottlenecks are within 4 hops from end nodes, i.e., they are mostly on network edges; (2) low-speed links still dominate Internet end-user access links, with 40% of end users having access bandwidth less than 2.2Mbps; (3) bottlenecks are not very persistent—only 20%–30% of the source-destination pairs in our experiments have perfect bottleneck persistence; (4) bottlenecks have close relationship with packet loss—60% of the bottlenecks in our measurements can be correlated with a lossy link no more than 2 hops away; the relationship between bottlenecks and link queueing delay is much weaker, with only 14% of correlation. We also demonstrated that Pathneck can help improve the performance of popular applications like CDN, and help web servers obtain their transmission-time distribution.

Chapter 5

Source and Sink Trees

IGI/PTR and Pathneck provide the techniques to measure end-to-end available bandwidth and to locate path bottleneck links. A common characteristic of these two techniques is their relatively small measurement overhead. However, compared with ping or traceroute, they are still expensive. For example, IGI/PTR is one of the end-to-end available bandwidth measurement tools that have the smallest overhead, but it still uses around 100KB probing packets per path. Consequently, for large-scale available bandwidth monitoring, i.e., measuring the N^2 paths in a N-node system when N is very large (hundreds or thousands), overhead remains a problem. For example, measuring all the end-to-end paths in an 150-node system will require over 2GB of probing packets, which is a significant network load.

To address this problem, we use a key insight from our Internet bottleneck property study—over 86% of Internet bottlenecks are on Internet edge. Given this fact, if we can measure the edge bandwidth of a network system, we can cover a large percentage of the bottlenecks that determine the path available bandwidth of any network system. The remaining problem is then how to efficiently measure and represent bandwidth information for the Internet edge. We use a novel structure of end-user routes—*source and sink trees*—to achieve that goal.

It is well known that the Internet is composed of a large number of networks controlled by different ISPs. Internet routes are determined by each ISP independently, and the overall connectivity topology looks much like a random graph with no simple pattern. However, if we only focus on the routes used by a single end user, they have a very clear tree structure. We call them the source and sink trees of the end user. This structure not only makes the notion of "Internet edge" concrete, it also provides valuable information on path sharing. This is because each tree branch is used by the end user to communicate with a large number of other nodes, and the corresponding paths are all affected by the performance of that tree branch. This type of sharing is the key that we will take advantage of to reduce the overhead for large-scale available bandwidth measurement.

In this chapter, we first define the source and sink trees at both IP-level (Section 5.1) and AS-level (Section 5.2), showing that AS-level source and sink trees closely approximate real tree structures (Section 5.3). We also show that the size of most trees are relatively small, and can thus be measured efficiently (Section 5.4). We then study how we can group destination nodes that share a tree branch using the RSIM metric (Section 5.5).

5.1 IP-Level Source and Sink Trees

The IP-level source tree of an end node refers to the graph composed by all the upstream routes it uses, and its IP-level sink tree refers to the graph composed by all the downstream routes it uses. Due to the scale of Internet, it is not necessary to consider a complete tree that is composed by the full routes, since our goal is to capture network-edge information. Therefore, we only consider the first E (for upstream routes) and the last N (for downstream routes) links of a complete IP-level path, and we call these two partial paths the *source-segment* and *sink-segment*, respectively. We also use the term *end-segment* to indicate either a source-segment or sink-segment.

The exact values of E and N are determined by applications, and different branches can have different E or N values. For end-to-end available bandwidth inference, there is a clear tradeoff between increasing the number of bottlenecks covered by using a large E or N value and increasing tree-measurement overhead. In Section 4.1, we have seen that around 86% of bottlenecks are within four hops from the end nodes. That implies that we need to focus on the top-four-layer trees, since increasing the tree depth will not significantly increase the number of bottlenecks covered. Therefore, we will use $E = 4$ and $N = 4$ for end-to-end available bandwidth inference. With this configuration, the terms source-segment and sink-segment are defined as follows. Let the path from node s to d be $Path(s, d) = (r_0 = s, r_1, r_2, ..., r_n = d)$, here $r_i (1 \leq i \leq n-1)$ are routers on the path. Then the source-segment of $Path(s, d)$ is

$$srcSgmt(s, d) = (r_0, r_1, r_2, r_3, r_4)$$

and the sink-segment of $Path(s, d)$ is

$$sinkSgmt(d, s) = (r_{n-4}, r_{n-3}, r_{n-2}, r_{n-1}, r_n)$$

Figure 5.1 illustrates the IP-level source trees for end nodes a_0, b_0 and the IP-level sink trees for end nodes c_0, d_0. The dashed lines indicate the omitted central parts of the paths. Here we have $srcSgmt(a_0, c_0) = (a_0, a_1, a_3, a_6, a_{10})$ and $sinkSgmt(c_0, a_0) = (c_8, c_4, c_2, c_1, c_0)$, etc.

Given the IP-level source and sink trees, the intuition of reducing large-scale available bandwidth measurement overhead is as follows. If bottlenecks are all on end-segments, we only need to consider the available bandwidth of source-segments and sink-segments, while ignoring links within the "Internet Core" as illustrated in Figure 5.1. This is an

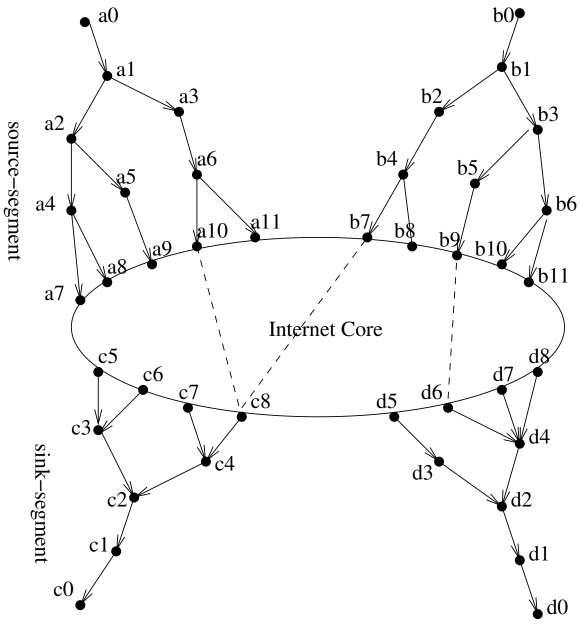


Figure 5.1: IP-level source and sink trees

important observation, since for a large system, many paths will share a same end-segment. For example, $Path(a_0, c_0)$ and $Path(b_0, c_0)$ share sink-segment $(c_8, c_4, c_2, c_1, c_0)$. This means that the measurement overhead is proportional to the number of end-segments, not the number of paths. As we will discuss later in this this chapter, Internet end nodes have on average only about ten end-segments, so the overhead is linear to the number of system nodes.

5.2 AS-Level Source and Sink Trees

IP-level source and sink trees can efficiently capture Internet edge route and bandwidth information, but to fully take advantage of the tree structures, we also need AS-level information. This is because end-to-end routes are determined together by intra-AS routes and inter-AS routes. Intra-AS routes are generally some type of shortest-path routes within the corresponding AS, while inter-AS routes are determined by the BGP protocol, which is based on AS. Therefore, to integrate the AS-level route information, we also define source and sink trees at the AS-level. In practice, AS-level source and sink trees have three advantages over IP-level trees. First, AS-level routes have been shown to have some useful properties like valley-free, shortest-path routing, and tiering structure. These properties can simplify the design of tree-based algorithms, as demonstrated in the BRoute system. Second, AS-level routes hide many factors that affect IP-level routes, like load balance routing which generally happens within ASes. Therefore AS-level routes tend to be more

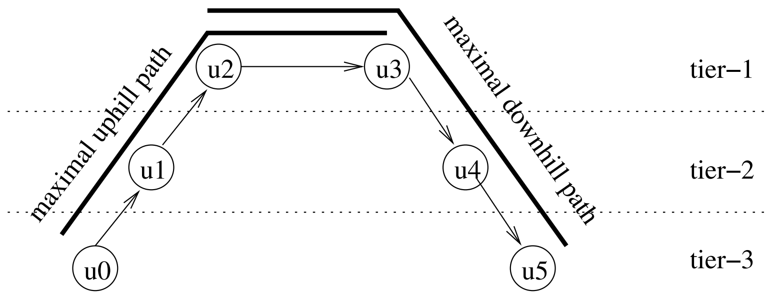


Figure 5.2: Maximal uphill/downhill path

stable than IP-level routes, and require less monitoring effort in the long term. Finally, and perhaps most importantly, as we will demonstrate in Section 5.3, AS-level source and sink trees also have a very clear tree-like structure. That makes AS-level trees a perfect abstraction for IP-level trees, thus effectively reducing the tree size to cover the same scope of route.

AS-level source and sink trees are defined based on two important properties of Internet ASes: the valley-free property of AS paths and the five-tier classification of Internet ASes. On today's Internet, depending on how BGP (Border Gateway Protocol) routes are exchanged between two ASes, AS relationships can be classified into four types: customer-to-provider, provider-to-customer, peering, and sibling relationships (see Gao [50] for details). The first two relationships are asymmetric, while the latter two are symmetric. An important property of AS relationships is the *valley-free* rule: after traversing a provider-to-customer or peering edge, an AS path can not traverse a customer-to-provider or peering edge. Consequently, an AS path can be split into an uphill and a downhill path. An uphill path is a sequence of edges that are not provider-to-customer edges, while a downhill path is a sequence of edges that are not customer-to-provider edges. The *maximal uphill/downhill path* is the longest uphill/downhill path in an AS path. Note that, unlike the definition from Gao [50], we allow uphill/downhill paths to include peering edges. This increases the chances that two trees have a common-AS.

A number of algorithms [50, 114, 29, 82] have been proposed to infer AS relationships using BGP tables. In particular, Subramanian et.al. [114] classifies ASes into five tiers. Tier-1 includes ASes belonging to global ISPs, while tier-5 includes ASes from local ISPs. Intuitively, if two connected ASes belong to different tiers, they are supposed to have a provider-to-customer or customer-to-provider relationship; otherwise, they should have a peering or sibling relationship. To be consistent with the valley-free rule, we say that an AS with a smaller (larger) tier number is in a *higher* (*lower*) tier than an AS with a larger (smaller) tier number. An end-to-end path needs to first go uphill from low-tier ASes to high-tier ASes, then downhill until reaching the destination (Figure 5.2).

Formally, let $Tier(u_i)$ denote the tier number of AS u_i. then an AS path $(u_0, u_1, ..., u_n)$

is said to be valley-free iff there exist $i, j(0 \leq i \leq j \leq n)$ satisfying:

$$Tier(u_0) \geq ... \geq Tier(u_{i-1}) > Tier(u_i) = ... = Tier(u_j) < Tier(u_{j+1}) \leq ... \leq Tier(u_n)$$

The maximal uphill path is then $(u_0, u_1, ..., u_j)$, and the maximal downhill path is $(u_i, u_{i+1}, ..., u_n)$. The AS(es) in the highest tier $\{u_i, ..., u_j\}$ are called *top-AS(es)*. In Figure 5.2, (u_0, u_1, u_2, u_3) is the maximal uphill path, (u_2, u_3, u_4, u_5) is the maximal downhill path, and $\{u_2, u_3\}$ are the top-ASes. The above formula allows an AS path to include multiple peering links. While this rarely happens, it allows us to resolve one type of error from the *AS-Hierarchy* data set (described in the next section), where two ASes with customer-to-provider or provider-to-customer relationship may have a same tier number.

AS-level source and sink trees are defined using the maximal uphill and downhill paths. Specifically, the *AS-level source tree* for a node s refers to the graph $srcTree(s) = (V, E)$, where $V = \{u_i\}$ includes all the ASes that appear in one of the maximal uphill paths starting from s, and $E = \{(u_i, u_j)|u_i \in V, u_j \in V\}$ includes the directional links among the ASes in V, i.e. $(u_i, u_j) \in E$ iff it appears in one of the maximal uphill paths starting from s. *AS-level sink tree* is defined in the same way, except using maximal downhill paths. Below we show that AS-level source and sink trees closely approximate tree structures in practice.

5.3 Tree Structure

In this section, we look at how closely the source and sink trees approximate real tree structures. This is an important property because it allows us to clearly group end nodes based on the tree branches they use. As shown in the next chapter on the BRoute system, the tree property also helps simplify some algorithm design. Below we first describe the data sets that are used in our analysis.

5.3.1 Route Data Sets

We use five data sets in our analysis. The *BGP* data set includes the BGP routing tables downloaded from the following sites on 01/04/2005: University of Oregon Route Views Project [23], RIPE RIS (Routing Information Service) Project [16] [1], and the public route servers listed in Table 5.1 which are available from [18]. These BGP tables include views from 190 vantage points, which allow us to conduct a relatively general study of AS-level source/sink tree properties. Note that the sources for the above BGP tables are often peered with multiple ASes, so they include views from all these ASes. We separate the BGP tables based on peering addresses because an end node generally only has the view from one AS.

[1] Specifically, we use ripe00.net – ripe12.net, except ripe08.net and ripe09.net, which were not available at the time of our downloading.

Table 5.1: Route servers

route-server.as5388.net	route-server.ip.tiscali.net
route-server.as6667.net	route-server.mainz-kom.net
route-server.belwue.de	route-server.opentransit.net
route-server.colt.net	route-server.savvis.net
route-server.east.allstream.net	routeserver.sunrise.ch
route-server.eu.gblx.net	route-server.wcg.net
route-server.gblx.net	route-views.ab.bb.telus.com
route-server.gt.ca	route-views.bmcag.net
route-server.he.net	route-views.on.bb.telus.com
route-server.host.net	route-views.optus.net.au
route-server.ip.att.net	tpr-route-server.saix.net

The *Rocketfuel* data set is mainly used for IP-level analysis of end-segments. We use the traceroute data collected on 12/20/2002 by the Rocketfuel project [109, 17], where 30 Planetlab nodes are used to probe over 120K destinations.[2] This destination set is derived from the prefixes in a BGP table that day. Since it covers the entire Internet, the inferred AS-level source trees from this data set are complete.

The *Planetlab* data set is collected by the authors using 160 Planetlab nodes, each from a different physical location. It includes traceroute result from each node to all the other nodes and it is used to characterize AS-level sink tree properties. This data set is part of the measurements conducted for the analysis in Section 6.4.

The *AS-Hierarchy* data set is from [2]. We downloaded two snapshots to match our route data sets: one on 01/09/2003[3], which is used for mapping *Rocketfuel* data set; the other on 02/10/2004, which is the latest snapshot available, and it is used for mapping *BGP* and *Planetlab* data sets. These data uses the heuristic proposed by Subramanian et.al. [114] to assign tier numbers to all the ASes in the BGP tables used in the computation. As an example, the 02/10/2004 data set identifies 22 tier-1 ASes, 215 tier-2 ASes, 1391 tier-3 ASes, 1421 tier-4 ASes, and 13872 tier-5 ASes.

The *IP-to-AS* data set is downloaded from [7]. Its IP-to-AS mapping is obtained using a dynamic algorithm, which is shown to be better than results obtained directly from BGP tables [81].

Three perspectives of these data sets need to clarify. First, while neither the *AS-Hierarchy* nor the *IP-to-AS* data set are perfect, they do provide very good coverage. For example, Figure 5.3 shows the results for the *Rocketfuel* data set. In this figure, "no tier" indicates the percentage of paths for which at least one hop could not be mapped onto a tier

[2] The original data set covers three days and uses 33 nodes. We only use the data from one of the days, and we discarded the data for 3 nodes because they could not get valid traceroute results for a large portion of destinations.

[3] Among the available data sets, it is the closest to the *Rocketfuel* data set in terms of measurement time.

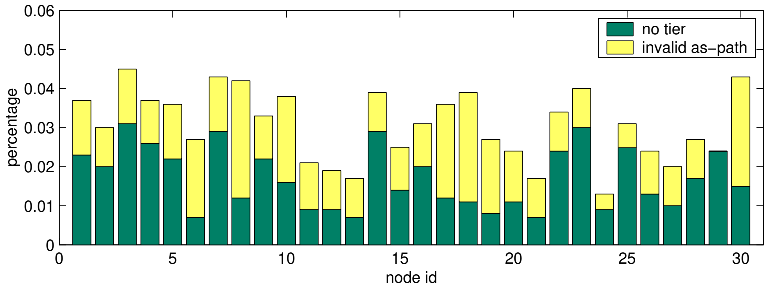


Figure 5.3: Classification of the AS paths in the *Rocketfuel* data set

Figure 5.4: Example of multi-tier AS relationship

number, either because its IP address could not be mapped onto an AS, or because its AS does not have a tier number. The "invalid as-path" bars show the percentage of AS paths that are not valley-free. On average, only 3% of the paths fall in these categories, while for each node fewer than 5% of the paths belong to these two categories. We will exclude these paths from our analysis, which we believe should not change our conclusions.

Second, these data sets were collected at different times. This is not an issue if they are used independently. Actually, evaluations using data sets collected at different times can make our analysis results more general. However, the time difference could introduce errors when we need to use the mapping data sets (*AS-Hierarchy* and *IP-to-AS*) with the route data sets (*BGP*, *Rocketfuel* and *Planetlab*) together. We are currently unable to determine whether these errors have positive or negative impact on our analyses, due to the lack of synchronized data sets.

Finally, we only have complete route information for 30 end nodes (from the *Rocketfuel* data set), while we have complete AS-level route information for 190 vantage points. Since tree-property analysis requires a complete set of route information for a end node, the following analysis will mainly focus on AS-level source and sink trees.

5.3.2 Tree Proximity Metric

An AS-level source or sink tree is not necessarily a real tree due to two reasons. First, ASes in the same tier can have a peering or a sibling relationship, where data can flow in either direction; that can result in a loop in the AS-level source/sink tree. Second, customer-to-provider or provider-to-customer relationship can cross multiple tiers. An example from the *Rocketfuel* data set is shown in Figure 5.4; we see that tier-3 AS11537 can reach tier-1

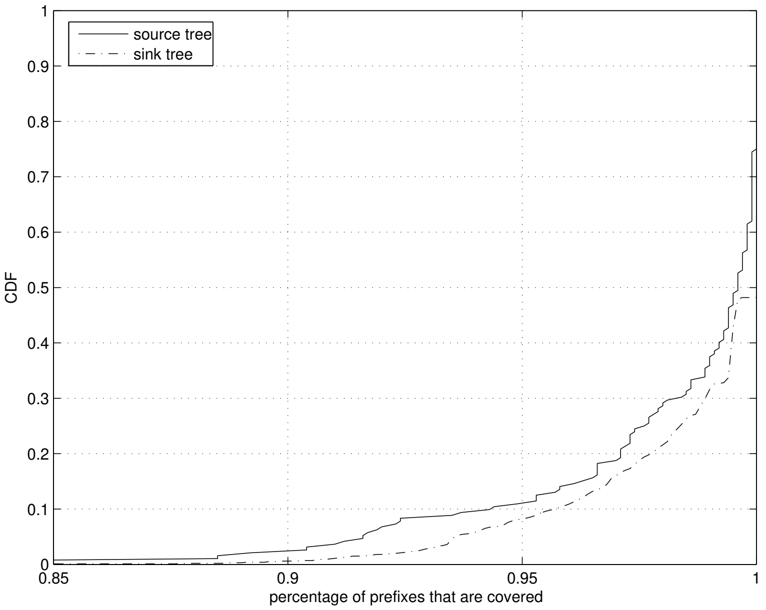


Figure 5.5: The tree proximities of the AS-level source/sink trees from the *BGP* data set

AS7018 either directly or through tier-2 AS201. Obviously, the corresponding AS-level source tree is not a tree since AS7018 has two parents.

In the following, we use the *tree-proximity metric* to study how closely AS-level source/sink trees approximate real tree structures. For AS-level source trees it is defined as follows, the definition for AS-level sink trees is similar. For both the *BGP* and the *Rocketfuel* data set, we first extract all maximal uphill paths for each view point. A view point is either a peering point (in *BGP* data set) or a measurement source node (in *Rocketfuel* data set). We count the number of prefixes covered by each maximal uphill path, and use that number as the popularity measure of the corresponding maximal uphill path. We then construct a tree by adding the maximal uphill paths sequentially, starting from the most popular one. If adding a new maximal uphill path introduces non-tree links, i.e., gives a node a second parent, we discard that maximal uphill path. As a result, the prefixes covered by that discarded maximal uphill path will not be covered by the resulting tree. The *tree proximity* of the corresponding AS-level source tree is defined as the percentage of prefixes covered by the resulting tree. While this greedy method does not guarantee that we cover the largest number of prefixes, we believe it provides a reasonable estimate on how well an AS-level source tree approximates a real tree.

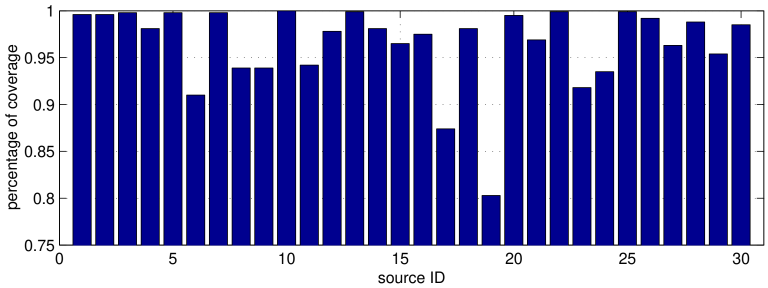


Figure 5.6: The tree proximities of the AS-level source trees from the *Rocketfuel* data set

5.3.3 Analysis Results

Using the *BGP* data set, we can build an AS-level source tree for each of the 190 view points. The distribution of the tree proximities is shown as the solid curve in Figure 5.5. About 90% of the points have a proximity over 0.95, and over 99% are above 0.88. This shows that the AS-level source trees are indeed very close to real tree structures. This conclusion is consistent with the finding by Battista et.al. [29], who noticed that a set of AS relationships can be found to perfectly match the partial view of BGP routes from a single vantage point.

We also built AS-level sink trees using the *BGP* data set. We identified the prefixes that are covered by at least 150 view points, i.e., for which we can get over 150 maximal downhill paths. We picked the number "150" because it can give us a large number of trees. We tried a number of different values and they lead to the same conclusion. The dashed curve in Figure 5.5 shows the distribution of the tree proximities for the 87,877 AS-level sink trees which have over 150 maximal downhill paths. Again, we see the results support the argument that AS-level sink trees closely approximate real trees. The results are in fact slightly better than for source trees, which could be a result of the limited number of downstream routes used for the AS-level sink-tree construction.

We repeated the AS-level source trees analysis for the *Rocketfuel* data set.[4] Figure 5.6 plots the tree proximities for the 30 AS-level source trees. We see that 2 trees are below 0.9, 7 are falling between 0.9 and 0.95, and all the other 23 AS-level source trees have tree proximities over 0.95. While fairly good, these results are worse than the results from the *BGP* data set. We identify several reasons for this. First of all, multihomed hosts can easily create violations in the tree structure because they can use arbitrary policies to select an interface. We manually confirm that node 6, 17, 19, 23, and 24 are all multihomed hosts. Second, traceroute results are subject to measurement noise since the measurements were conducted over 6-hour period [109], in which routes could change, thus introducing inconsistency such as an extra parent for an AS in the AS-level source tree. The *BGP* data

[4]The analysis on AS-level sink trees is not repeated since we only have data from 30 nodes, i.e., at most 30 downstream routes for each destination.

set does not have this problem since BGP table provides a pure AS-level route snapshot, Furthermore, Internet exchange points can introduce extra ASes that are not present in BGP routes. These exchange points could easily become an extra parent for some ASes.

5.3.4 Discussion

We have shown that both AS-level source and sink trees closely approximate real tree structures. Earlier in this section, we identified two possible causes for violations of the tree property. We found that the second cause, i.e. the creation of multiple paths to reach a higher tier AS, was by far the most common reason for discarding a maximum uphill path during tree construction. We speculate that these violations are caused by load-balancing-related routing policies such as MOAS (Multiple Origin AS), SA (Selected Announced Prefixes), etc. The fact that the first cause seldom occurs, i.e., there are few loops in the AS-level source/sink tree, implies that the ASes in the same tier can be "ranked", as implied by Battista et.al. [29]. That is, if two peering ASes are in the same tier, although data can flow in both directions, the data transmission is always in one direction for a specific end node. That is, from the view of a single end node, a peering or sibling relationship actually behaves like a customer-to-provider or provider-to-customer relationship.

An important implication of this observation is that, once we have obtained the maximal uphill paths, the tier numbers from [2] are no longer needed. The AS-level source/sink tree constructed using the maximal uphill/downhill paths determines the relationships for the ASes in the tree. As a result, errors from the *AS-Hierarchy* data set, if any, only affect the AS-level source/sink tree construction when it cannot correctly determine the top-AS(es) for a path. Given that over 90% of Internet paths have top-AS(es) at tier-1 or tier-2 (see Section 5.4), which are relatively stable, we believe that the estimation error introduced by this data set is very limited.

5.4 Tree Sampling

We have seen two ways of constructing AS-level source/sink trees—using BGP tables and using traceroute—as we demonstrated using the *BGP* and the *Rocketfuel* data sets, respectively. While using BGP is very easy, this type of data is generally not readily available to end nodes; moreover, it cannot be used for AS-level sink trees. On the other hand, having each system node use traceroute to probe the entire Internet, as done in the *Rocketfuel* project, is expensive. Fortunately, this is not necessary. In this section, we first show that 70% of IP-level sink trees have less than 10 branches. We first show that AS-level source trees are in general small. We cannot do a similar study on AS-level sink trees because we have insufficient data to build complete AS-level sink trees, but we speculate the conclusions from AS-level source trees also apply to AS-level sink trees. We then present some preliminary results on how to place traceroute landmarks that will support

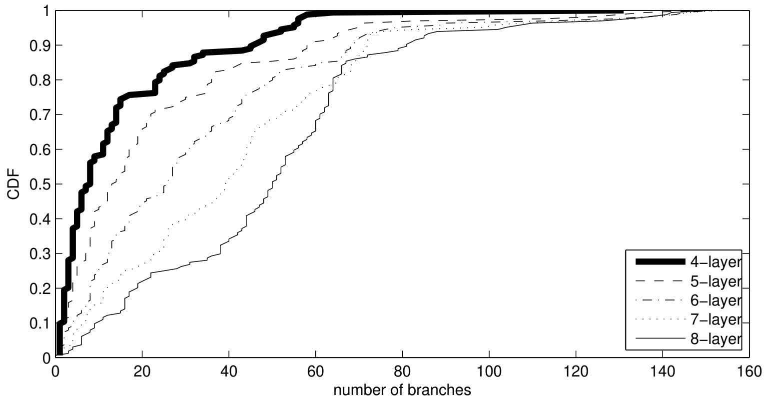


Figure 5.7: IP-level sink-tree size distribution in the *Planetlab* data set.

efficient sampling of the AS-level source tree.

Figure 5.7 shows that the distribution of the number of tree branches for the 210 IP-level sink trees measured using 210 *Planetlab* nodes. The bold solid line shows the distribution for the 4-layer trees, which is our standard definition. We can see, 70% of sink trees have less than 10 branches, which shows that most of IP-level sink trees indeed have very limited size. There are a few trees that have over 100 different branches. This is due to the set of Internet2 Planetlab nodes which are deployed very closely to Internet2 peering link. In practice, these types of nodes are much less likely to be encountered by an end user or application. This figure also illustrates the changes when the tree include more layers of nodes. Clearly, with more layers included in a tree, there will be more branches. For example, for 8-layer sink trees, over 70% of them have more than 40 branches.

Figure 5.8 plots the size distribution of AS-level source trees in the *BGP* data set, measured as the number of distinct ASes in the tree. We separate the trees into three groups based on the tier number of their root ASes. Tier-4 and tier-5 are grouped with tier-3 because there are only 6 and 5 AS-level source trees with root AS in tier-4 and tier-5. Intuitively, the higher the tier (lower tier number) of the root AS, the smaller the size of the AS-level source tree. This is confirmed by Figure 5.8: the AS-level source trees with root AS in tier-3/4/5 are significantly larger than those in tier-1 or tier-2. Even for tier-3/4/5, however, the largest AS-level source tree has fewer than 400 ASes and some are as small as 50 ASes, which can happen, for example, if the root AS is directly connected to a tier-1 AS. We conclude that AS-level source trees are indeed quite small. This observation was confirmed using the *Rocketfuel* data set, where the tree size varies from 50 to 200.

The limited AS-level source tree size gives us the opportunity to measure it using a small number of traceroutes. Intuitively, if routes to different prefixes could share the same maximal uphill path, only one of these routes needs to be measured to cover all these prefixes. Since the size of AS-level source tree is limited, a small number of traceroutes should suffice. A good algorithm for the selection of landmark locations is left for

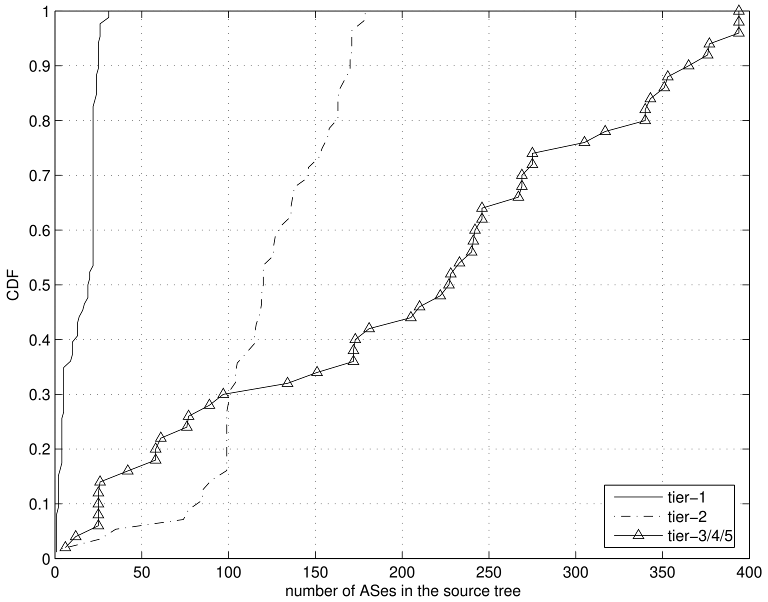


Figure 5.8: Tree size distribution from different tiers of sources, using the *BGP* data set.

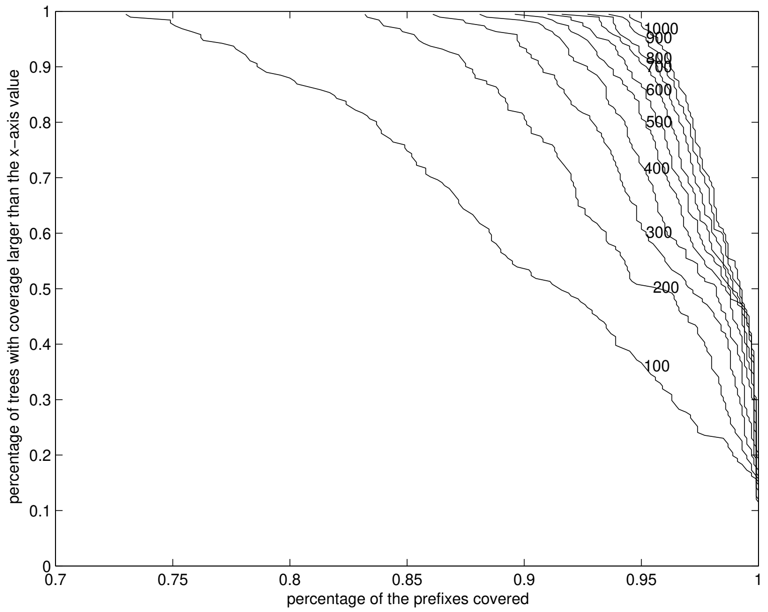


Figure 5.9: Random sampling of the AS-level source tree. The number of AS paths selected are marked on the curves.

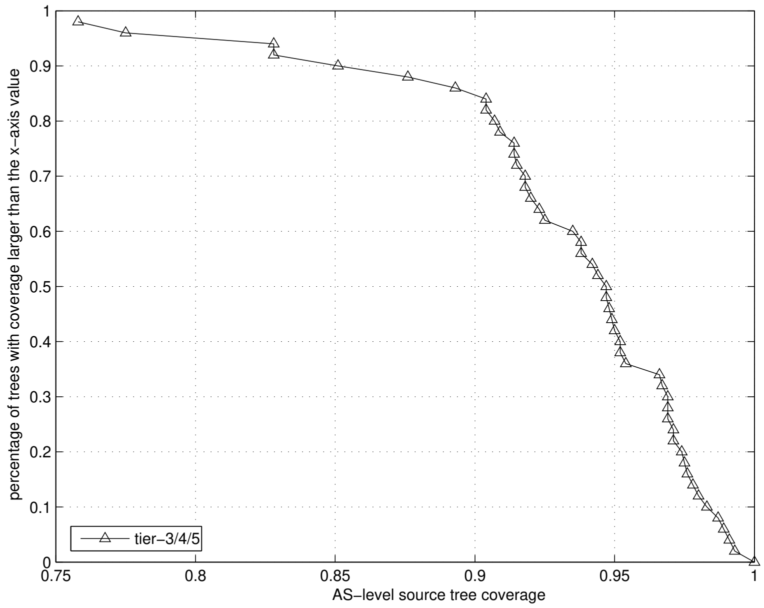


Figure 5.10: Coverage of using only tier-1 and tier-2 landmarks

future work. Instead, we present some measurements that provide insight in the number of traceroute landmarks that are needed to provide good coverage of the AS-level source tree. We do this by presenting the results for two simple selection algorithms: random and tier-based.

In our first study, we randomly chose 100-1000 AS paths from each BGP table to construct a sampled AS-level source tree. Figure 5.9 shows the coverage for different numbers of paths used. In this figure, the x-axis is the percentage prefixes covered by the sampled tree, while the y-axis shows the number of trees that have at least that much coverage. We see that 100 traceroutes can cover at least 70% of the prefixes for all trees, while 300 traceroutes will allow most of trees cover over 90% of the prefixes.

Another approach is to place landmarks strategically. In the *Rocketfuel* data set, we find that over 90% paths have top-AS(es) at tier-1 or tier-2. More importantly, there are only 22 tier-1 ASes and 215 tier-2 ASes in the 02/10/2004 *AS-Hierarchy* data set. The implication is that if we deploy one traceroute landmark in each of these 237 ASes, the measured tree can cover at least 90% of the original AS-level source tree. In practice, one AS can map to multiple different prefixes; in that case we randomly pick one of the paths with the shortest prefix length, hoping that this prefix is the most stable one.

Figure 5.10 plots the sampling performance for the *BGP* data set, using the 237 traceroute landmarks selected above as traceroute destinations. We only plot the AS-level source trees with root ASes in tier-3/4/5. Those from tier-1/2 are ignored because their AS-level source trees are completely covered. We can see that, among the 49 sampled trees, only 2 have coverage less than 80%, 5 between 80% and 90%, the other 42 all cover over 90% of the trees. This shows the effectiveness of the tier-based selection of traceroute landmarks. These results suggest that by extending this algorithm to include a set of carefully selected tier-3 and tier-4 landmarks, it should be possible to get very good coverage of the AS-level trees with a reasonable number of landmarks, e.g. less than 1000.

5.5 Tree-Branch Inference

The tree structure of end-user routes implies that we can split Internet destinations into different groups, with the destinations in each group sharing a tree branch. To take advantage of this sharing, we need a mechanism to identify *which* destinations belong to a same group. The RSIM metric that we will present in this section is one of the possible mechanisms. RSIM is a metric that quantifies route similarities of end nodes. Here route similarity is defined as the overlap of two end-to-end routes between two nodes and an arbitrary third node. That is, we regard routes as a property of end nodes, and route similarity captures the similarity of this property for different end nodes. Ideally, we expect to see that any two end nodes with a RSIM value larger than certain threshold TH_{RSIM} share a large portion of their routes with any other end node (say A), making it sufficient for them to also share one of A's source or sink tree branches.

In the rest of this section, we first define the RSIM metric, and discuss RSIM's properties including destination sensitivity, measurability, and symmetry. We then demonstrate how to use the RSIM metric to group end nodes.

5.5.1 The RSIM Metric

In the remainder of this section, the term "route similarity" refers specifically to RSIM, and is defined as follows. Let $P(s, d)$ denote the IP level route from node s to node d; $L(s, d)$ denote the number of links on $P(s, d)$; $Total(s_1, s_2, d) = L(s_1, d) + L(s_2, d)$; and $Common(s_1, s_2, d)$ denote the total number of links that are shared by $P(s_1, d)$ and $P(s_2, d)$. Let SET denote a set of Internet destinations (see Figure 5.11), then the route similarity between s_1 and s_2 *relative* to SET is defined as:

$$RSIM(s_1, s_2, SET) = \frac{\sum_{d \in SET} 2 * Common(s_1, s_2, d)}{\sum_{d \in SET} Total(s_1, s_2, d)} \tag{5.1}$$

Note this definition uses upstream routes from s_1 and s_2, RSIM can be similarly defined using downstream routes. Intuitively, this definition captures the percentage of links shared by the two routes $P(s_1, d)$ and $P(s_2, d)$. In this definition, when SET is obvious, we simplify $RSIM(s_1, s_2, SET)$ as $RSIM(s_1, s_2)$. It is easy to see that $RSIM(s_1, s_2, SET) \in [0, 1]$ for any s_1, s_2, and SET. The larger $RSIM(s_1, s_2, SET)$ is, the more similar the routes of s_1 and s_2 are. Figure 5.12 shows an example an RSIM computation. Here

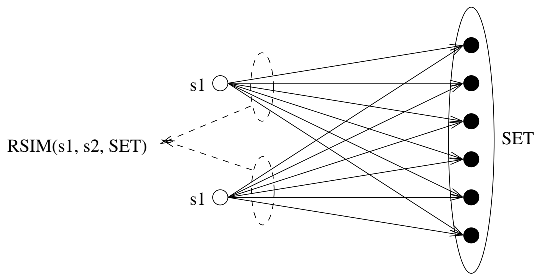


Figure 5.11: $RSIM(s_1, s_2, SET)$

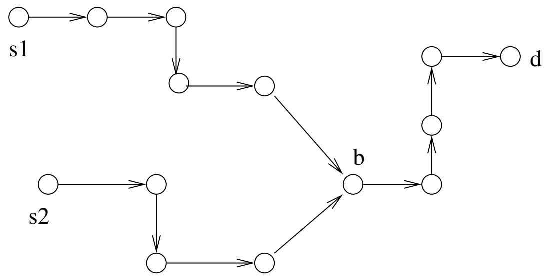


Figure 5.12: $RSIM(s_1, s_2, \{d\}) = 8/17$

we have $SET = \{d\}, Common(s_1, s_2, \{d\}) = 4, Total(s_1, s_2, \{d\}) = 9 + 8 = 17$, so $RSIM(s_1, s_2, \{d\}) = 2 * 4/17 = 8/17$.

5.5.2 RSIM Properties

We analyze the following properties of RSIM to show that it is a useful metric:

- *Destination Sensitivity:* how sensitive is the RSIM value to the choice of destinations;
- *Measurability:* how many measurements are required to compute the value of RSIM;
- *Symmetry:* what is the difference between the upstream route similarity and downstream route similarity?

We will use two data sets described in Section 5.3 in these analyses. One is the *Rocketfuel* data set. In this data set, because the destination IP addresses are randomly selected, they do not necessarily correspond to online hosts, so for many of the destinations the traceroute measurements are not complete. To avoid the impact of incomplete routes on our analysis, we only consider the 5386 destinations which can be reached by at least 28 source nodes using traceroute. The second data set is the *Planetlab* data set, which we collected using 160 Planetlab nodes, each from a different site. This data set provides us a route matrix, where we have routes in both directions for all pairs of nodes. This is the largest route matrix we are aware of, and it is very useful for characterizing the symmetry property of RSIM.

Destination Sensitivity

Since RSIM is a function of the destination set SET, the value of RSIM can be different for different destination sets. However, for many applications it is preferable that RSIM is largely independent of the SET parameter, i.e., it is a fundamental property that only needs to be measured once. This property is very important for reducing measurement overhead.

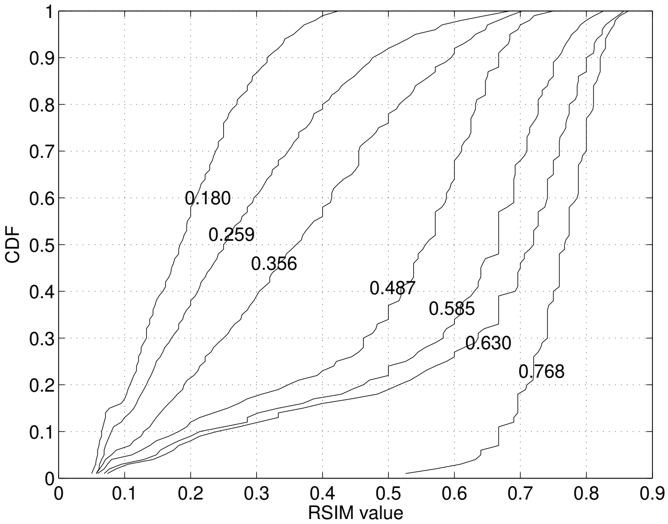


Figure 5.13: Destination-sensitivity of RSIM

In this section, we first focus on the case where SET includes only a single destination. The case where SET includes multiple destinations are discussed in Section 5.5.2.

We use the *Rocketfuel* data set to study destination sensitivity. We first set SET to include the 5386 reachable destinations in the *Rocketfuel* data set, and compute RSIM values for all the 435 ($= 30 * 29/2$) pairs of source nodes. We will use these RSIM values as benchmark values since they are from the largest destination set possible. The distribution of these RSIM values has a sharp peak around 0.7. Specifically, 85% of the 435 pairs have RSIM values between 0.65 and 0.8. This confirms earlier observations that in 2002, most Planetlab nodes had very little diversity in how they connected to the Internet (most used Abilene). However, there is some diversity: the RSIM values range from 0.1 to 0.8.

To study the destination sensitivity of RSIM for node pairs with different RSIM values, we selected seven source-node pairs with RSIM values roughly evenly distributed in the range [0.1, 0.8]. We calculate their RSIM values for *each* of the 5386 individual destinations for which they have complete route data. These similarity values are plotted in Figure 5.13. Each curve plots the cumulative distribution for the RSIM values of one source-node pair relative to each individual destination. The numbers marked on the curves are the benchmark RSIM values. The seven curves can be classified into three groups:

1. The first group only includes the rightmost curve. This curve corresponds to a pair of source nodes with the highest benchmark RSIM value (0.768) among the seven pairs, i.e., their routes are very similar. The similarity values of this source-node pair for individual destinations are distributed in a fairly small region—90% of them are

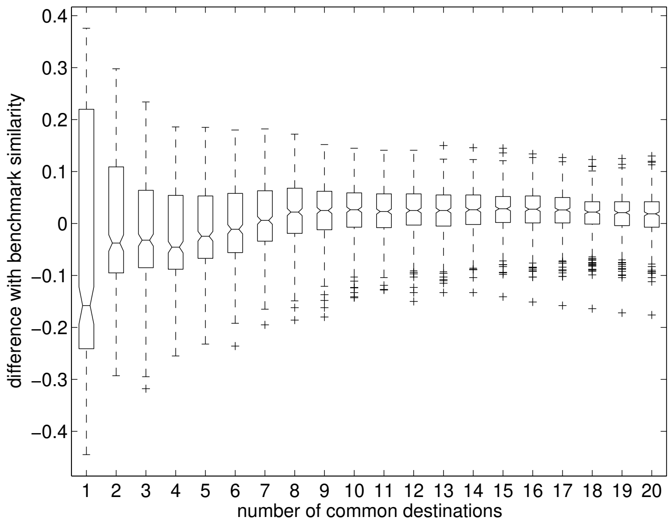


Figure 5.14: Measurability of RSIM

in [0.65, 0.85]. That shows that the similarity between this pair of source nodes is not very sensitive to the destination selected.

2. The middle three curves make up the second group. Their benchmark RSIM values are 0.467, 0.585, and 0.630, respectively, and they represent source-node pairs with average similarity values. Clearly, the RSIM values for individual destinations in this group are more diverse than in the first group. The lowest 30% of similarity values are significantly lower than the other 70% of the values. However, the highest 70% of similarity values cluster within a small region with 0.2 width.

3. The three leftmost curves represent the third group, where the node pairs have low similarity—0.180, 0.259, and 0.356. The similarity values of these source-node pairs with respect to individual destinations is almost evenly distribute in a large range with 0.4-0.6 width. That means that their RSIM values are quite sensitive to the SET parameter.

The above results show that the larger the benchmark RSIM value is, the less sensitive the RSIM values are to the chosen destination. Specifically, for node pairs with properties similar to the rightmost curve, the RSIM values are not sensitive to the specific single-destination set.

Measurability

In this subsection, we show that measuring route similarity only needs a small number of traceroute measurements. We again use the *Rocketfuel* data set to demonstrate this

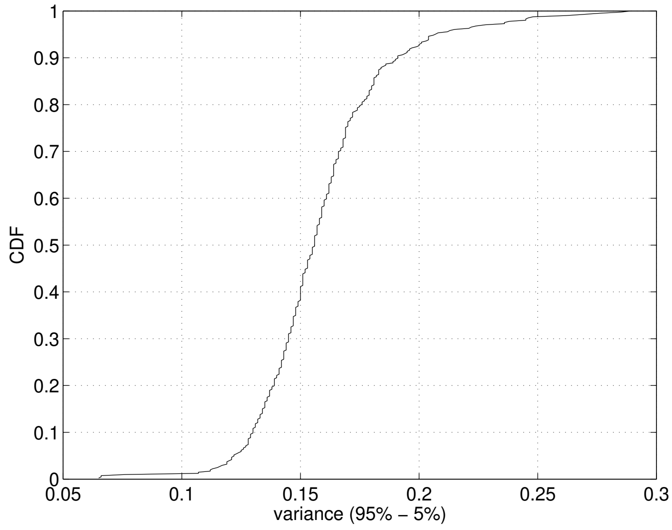


Figure 5.15: Impact of randomness

property. In this data set, we use different numbers (x) of randomly (uniform) selected destinations to compute route similarity ($RSIM_x$) for each of the 435 source-node pairs. We then compare them with their benchmark values ($RSIM_b$) which is based on all 5386 reachable destinations as discussed in Section 5.5.2, and compute the relative difference as $(RSIM_x - RSIM_b)/RSIM_b$. Figure 5.14 plots the distributions of the relative differences from all 435 source-node pairs. The x-axis is the number of routes used in computing $RSIM_x$. The bars are plotted using the `boxplot` function of Matlab, where each bar corresponds to one distribution of the relative difference for all 435 source-node pairs. The middle boxes have three lines corresponding to the lower quartile, median, and upper quartile values, and the whispers are lines extending from each end of the box to show the extent of the rest of the data. We can see that the relative difference between $RSIM_x$ and $RSIM_b$ quickly drops as more destinations are used. Once $x \geq 10$, the median difference stays roughly constant at about 5%, although the variance decreases. This result shows that only 10-20 routes are needed to compute the value of RSIM for the *Rocketfuel* data set.

In the above analysis, for each source-node pair and each number of destinations, the similarity value is calculated using one instance of a randomly selected destination set. These values may be different for different randomly selected destination sets. To quantify the impact of this randomness, for each source-node pair, we select 1000 different random 10-destination sets and compare their RSIM values. For each source-node pair, we then record the difference between the 95 percentile similarity value and the 5 percentile similarity value, and use that as a variance measure for different random 10-destination sets. Figure 5.15 plots the cumulative distribution of this variance for all 435 source-node pairs.

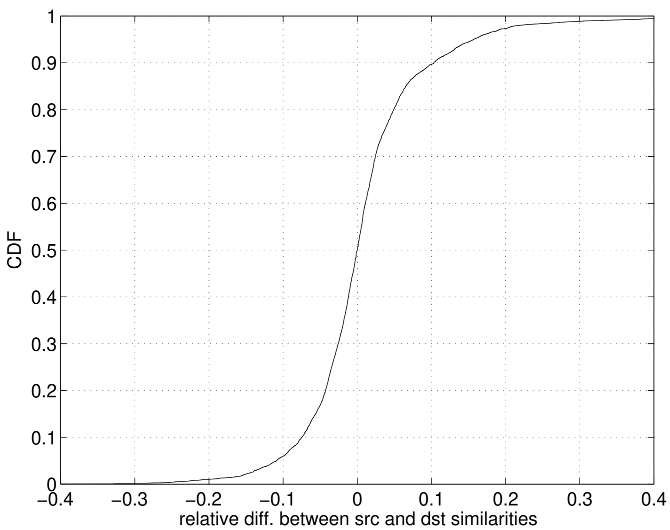


Figure 5.16: Symmetry of RSIM

We see that the variance for 93% of pairs is less than 0.2, which is small. This tells us that the choice of the randomly selected destination sets does not have a significant impact on the RSIM values.

An important implication of the results in Figure 5.14 and 5.15 is that RSIM is not sensitive to the choice of destination set when SET includes ten or more "random" destinations. The reason is as follows. Although RSIM can be very different for different individual destinations, as illustrated in Figure 5.13, ten or more different destinations can fairly well "cover" the distribution curve by including most important points. Therefore, even with different destination sets, since they are from the same distribution, which is determined by the node pair, they should converge to the same value.

So far, the results in this section were obtained using routes between 30 source nodes and 5386 destination nodes. We have done a similar analysis using all 120K destinations, i.e., including the incomplete route data. The results we obtained are similar. Even so, the data set used here only covers a limited fraction of nodes on the Internet, and whether or not our conclusion in this section can be extended to the whole Internet should be validated using larger and more diverse data sets.

Symmetry

It is well known that Internet routes are asymmetric [82], but we find that RSIM values computed using upstream routes and those using downstream routes are very similar. In this sense, RSIM is symmetric, i.e., it captures the similarity of both upstream and downstream routes. In this section, we use the *Planetlab* data set to show this property.

For each node pair among the 160 nodes, we calculate their route similarity using both upstream routes ($RSIM_{up}$) and downstream routes ($RSIM_{down}$). We then compute the difference as ($RSIM_{down} - RSIM_{up}$). Figure 5.16 plots the distribution of this difference for the 14,412 pairs which have at least 10 complete traceroute results to compute both $RSIM_{up}$ and $RSIM_{down}$. We can see that 84% of the pairs have difference within a small range of [-0.1, 0.1], which shows that $RSIM_{up}$ and $RSIM_{down}$ are indeed very similar. That means that, if two nodes have a high probability sharing a large portion of their upstream routes toward a destination, they will also have a high probability sharing a large portion of their downstream routes from that destination.

The implication of this property is two-fold. First, RSIM measurements do not necessarily need upstream routes. If only downstream routes are available, we can still compute RSIM. This property will be used in Section 5.5.3 to get a large sample set for our analysis. Second and more importantly, this property allows us to infer both source and sink segments. For example, if s_1 and s_2 have similar routes, they will both have similar sink segments for their routes towards a third node, and have similar source segments for those routes from a third node. The details are discussed in the next section.

5.5.3 Tree-Branch Inference

The RSIM metric can be used in at least two scenarios. On one hand, it can be used to group end nodes that have similar routes. Such groups not only can be used to identify tree-branch sharing as discussed at the beginning of this section, it also can be used by web sites or peer-to-peer systems for performance optimization. Based on the clustering information, for example, a web site can optimize for each cluster instead of each client, thus significantly reducing management overhead. On the other hand, RSIM also can be used to select a set of end nodes whose routes are very different with each other. End nodes selected in this way can serve as vantage points for measurement systems like the traceroute landmarks that we will discuss in the next chapter.

In this section, we focus on the first type of applications, i.e., grouping end nodes that have similar routes. We study this problem in the context of end-segment inference. That is, we want to group end nodes based on the probability that two end nodes in the group share end-segment for a large number of sources and the destinations. Below we first look at if there exists a RSIM value that allows us to group end nodes. We then present a case study on how often end nodes can be grouped in real network systems.

RSIM Threshold

Ideally, we would like to have the following two claims:

Claim 1: $\exists\, TH_{RSIM}, \forall\, s_1, s_2$, their upstream routes towards any node d share sink segment iff $RSIM(s_1, s_2) > TH_{RSIM}$.

Claim 2: $\exists\, TH_{RSIM}, \forall\, d_1, d_2$, their downstream routes from any node s share source segment iff $RSIM(d_1, d_2) > TH_{RSIM}$.

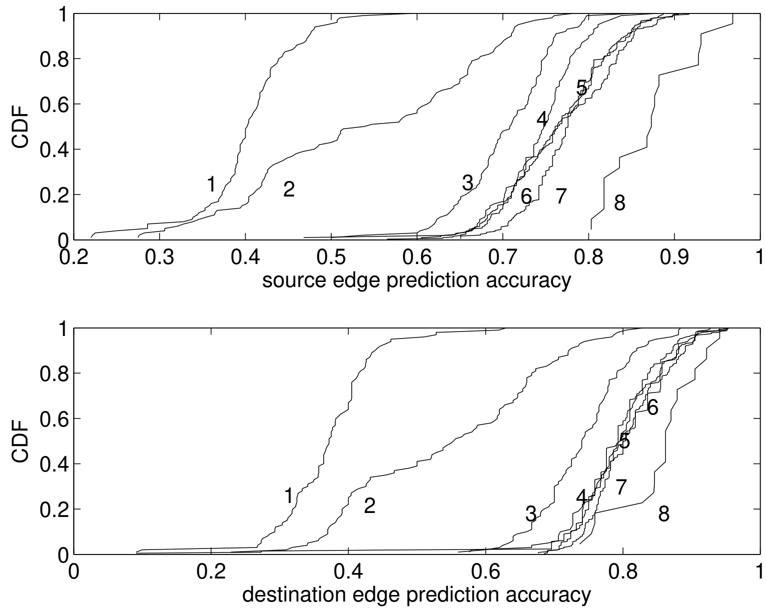


Figure 5.17: End-segment sharing probability.

We call a pair of nodes *neighbors* if their RSIM value is larger than TH_{RSIM}. Intuitively, if a pair of nodes s_1 and s_2 are neighbors, and the bottlenecks of $P(s_1, d)$ and $P(s_2, d)$ are on their sink segments, we then can use the bandwidth measured from $P(s_1, d)$ as that of $P(s_2, d)$, because they are very likely to share the bottleneck.

Of course, it is unrealistic to expect that we will be able to find a threshold TH_{RSIM} that gives 100% sharing of the remote end segments. Instead, we now look at whether a threshold TH_{RSIM} exists that indicates a high probability of end-segment sharing. We use the *Planetlab* data set for this study. We first take the 14,412 node pairs which have at least 20 complete traceroute results, compute their RSIM values using these routes, then group them into nine groups ($g_i, i = 1..9$) based on their RSIM values: $g_i = \{(s, d)|i * 0.1 \leq RSIM(s, d) < (i + 1) * 0.1\}(1 \leq i \leq 9)$. For each node pair in each group, we calculate the probability of sharing source segments and sink segments. Figure 5.17 plots the cumulative distribution of source/sink-segment sharing probabilities for each group. In this figure, g_9 is grouped into g_8 because there are only 6 pairs in g_9. The top graph plots the sharing probability for source segments, and the bottom graph plots the results for the sink segment. g_8 stands out distinctively with the best prediction accuracy—almost all node pairs in this group have a sharing probability higher than 0.8. Although not perfect, we think we can use the value 0.8 for TH_{RSIM}. That shows that RSIM can be used for end-segment inference with a high inference accuracy.

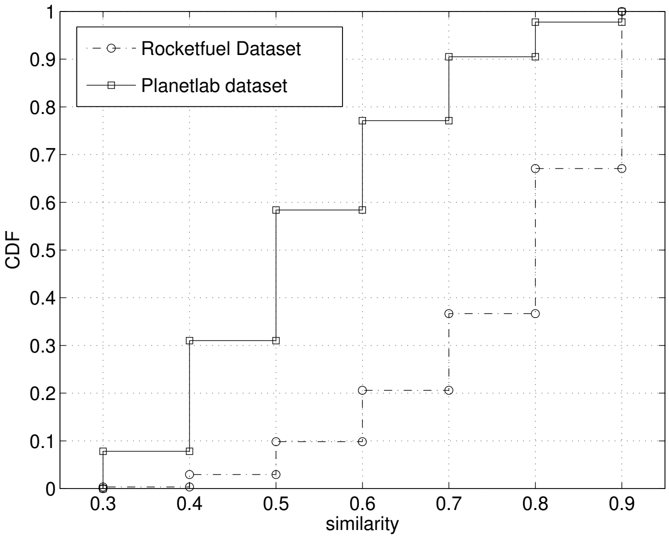


Figure 5.18: Probability of having a neighbor.

Probability of Sharing End Segments

The above results suggest that it will be useful for neighbors to share bandwidth information. Whether this is feasible depends on how likely it is that a node can find a neighbor. This is a very difficult question, but we can use the *Rocketfuel* and *Planetlab* data set to gain some insights. We use the *Rocketfuel* data set as an example of a system that includes nodes from all over the Internet, while we use the *Planetlab* data set to get the view for a real deployed system.

For the *Rocketfuel* data set, we use downstream routes to compute route similarities for the 5386 reachable destinations. The reason that we use destinations instead of sources is to obtain a large scale analysis. The symmetry property of RSIM demonstrated in Section 5.5.2 allows us to compute RSIM values using downstream routes. For the *Planetlab* data set, route similarities are computed for the 160 nodes using upstream routes. Figure 5.18 plots the distribution of node pairs in each group. In this figure, node pairs are again grouped as we did in Figure 5.17 (except that g_1 and g_2 are combined with g_3); the x-axis is the smallest RSIM value in each group. The dashed curve shows the result for the *Rocketfuel* data set. We see that 63% of end nodes can find at least one neighbor, i.e., their RSIM value is larger than 0.8. The *Planetlab* data set has significantly fewer end nodes, so only 10% of end nodes can find a neighbor.

It is worthwhile to mention that in both of these analyses, destinations are selected from different prefixes, while in reality, many system nodes can come from common prefixes. In this sense, the results presented in Figure 5.18 provide a pessimistic view.

5.5.4 Discussion

We are not aware of any metrics designed to quantify route similarity. Related work such as [33] and [53] has studied how to use relay nodes to increase route diversity for link or router fault tolerance, and their route diversity results may be used to measure route similarity. However, our work has a completely different focus—we are interested in segment route similarity, which can not be directly quantified using route diversity. Sometimes IP prefixes are also used to estimate route similarity. This is based on the observation that if two nodes have IP addresses from a common prefix, they often share routes. This approach has three limitations. First, a common IP prefix is not a sufficient condition for route similarity. Nodes from the same prefix, especially those from a large prefix, do not necessary share routes. Second, a common IP prefix is not a necessary condition for similar route, either. For example, from the *Rocketfuel* data set used in [109], we find that node `planet2.cs.ucsb.edu` and node `planetlab1.cs.ucla.edu` have very similar routes although they belong to completely different prefixes–131.179.0.0/16 (AS52) and 128.111.0.0/16 (AS131). Finally, the IP prefix does not *quantify* route similarity, thus it is hard to compare the similarities of different pairs of nodes.

One interesting point is that the definition of RSIM remotely resembles the synthetic coordinate systems proposed by GNP [88] or Vivaldi [41]. For example, in RSIM, the 10-20 traceroute destinations can be regarded as the landmarks; the routes between the landmarks and an end node can be looked as its coordinates; and the formula (5.1) listed in Section 5.5.1 can be used as the distance formula. However, RSIM is not a real coordinate system because its coordinates are not distance values. An intriguing piece of future work is to explore whether we can extend the RSIM metric to construct a real coordinate system for route similarity.

RSIM can be used for end-segment inference which is closely related to the goal of the BRoute system that will be described in the next chapter. In BRoute, each node collects AS-level source tree and sink tree information to infer end segments, which are further used to infer path bottlenecks and available bandwidths. The difference between RSIM and BRoute is that RSIM is a general metric that can be used by different applications, while BRoute focuses on path bandwidth inference by only characterizing routes of each individual node, i.e., BRoute does not directly quantify the similarity of routes from two different nodes. However, BRoute is a much more structured solution that is not sensitive to system node distribution, while using RSIM for end-segment inference depends on the chances that system nodes can find neighbors.

5.6 Summary

In this chapter, we studied source and sink tree structures at both the AS-level and the IP-level. We showed that AS-level source and sink trees closely approximate real tree structures—around 90% of them have tree-approximity over 0.95. These trees also have limited size: AS-level trees in our study have 50-400 tree nodes, and IP-level trees have 10-20 different branches. That shows the source and sink tree structure is an effective method to capture network-edge information. To group destination nodes that have a high probability of sharing tree branches, we proposed the RSIM metric. We showed that RSIM can be measured using a small number of random traceroutes, and it captures the similarity of both upstream routes and downstream routes. When using RSIM for end-segment inference, we showed that if a pair of nodes have an RSIM value larger than 0.8, they have a high probability of sharing the edges of their routes with an arbitrary third node.

Chapter 6

Large-Scale Available Bandwidth Estimation

We have seen that RSIM can be used to infer the end-segment used by a path and to reduce the overhead of large-scale available bandwidth monitoring. The problem with RSIM, however, is that it is an un-structured solution and its effectiveness is closely tied to the specific end-node composition of a system, which determines the probability that an end node can find a neighbor. In this chapter, we propose the BRoute system that provides a more structured solution and does not have this limitation. The key idea is to use AS-level source and sink trees to infer the two edges of an arbitrary end-to-end path, by leveraging previous work on AS relationships. In the rest of this chapter, we first introduce the BRoute design (Section 6.1). We then show how AS-level source and sink trees can be used to identify end segments (Section 6.2), and how to measure end-segment available bandwidth (Section 6.3). The overall path available-bandwidth inference accuracy is evaluated in Section 6.4. We discuss system overhead and security issues in Section 6.5.

6.1 BRoute System Design

6.1.1 Motivation

The design of BRoute is motivated by GNP [88] and similar coordinate systems [41, 40, 105, 95] that use geometrical spaces to model Internet end-to-end propagation delays. These systems assign a set of coordinates from a geometrical space to each node in a system, and use the Euclidean distance between two nodes as an estimate for their Internet delay. The coordinates are calculated based on measurements of the end-to-end delay from end nodes to a small set of "landmarks". Two intriguing properties distinguish such systems: (1) scalability—the system overhead is linear with the number of nodes in the system, and (2) since any node can estimate the latency between two nodes based on their coordinates, only minimal interaction between nodes is required. These two properties are exactly the goals that BRoute achieves for bandwidth estimation.

Using coordinate models for estimating latency is intuitively easy to understand. Internet propagation delay is determined by the physical length of data transmission links, which are laid on the surface of the earth. Since the earth surface is a geometrical space, it is not hard to understand why latency can fit into a geometrical space. However, this argument does not work for bandwidth. Internet path available bandwidth is determined by the bottleneck link, and there appears to be no reason why it would fit a geometrical space. In fact, since path available bandwidth is determined by one link, we would expect it to be harder to predict than latency and potentially very sensitive to even small changes in the route. This is somewhat similar to the loss monitoring problem discussed by Chen et.al. [34]. As a result, while routing may not be important in coordinate-based latency estimation, we believe it must be considered when estimating bandwidth.

6.1.2 BRoute Intuition

The BRoute system uses the source and sink trees that we discussed in the previous chapter, based on two important observations. First, most bottlenecks are on end-segments, and we only need to obtain available bandwidth information for both end-segments of a path to estimate path available bandwidth. Second, the size of source and sink trees is small for the first 4 layers. That is, relatively few routes exist near the source and destination compared with the core of the Internet, thus simplifying the problem of determining which end-segments a path takes, and which bottleneck it encounters. These two observations lead to the two key operations in BRoute. First, each node collects both routing and bottleneck information for the source and sink trees to which it is attached, using traceroute and Pathneck [56], respectively. This information can be published, similar to a set of coordinates. Second, in order to estimate the available bandwidth between a source node and a sink node, a third node would collect the tree information for the source and sink and use it to determine the route taken by the end segments, and the likely bottleneck location and available bandwidth.

Besides the source and sink trees, a key problem that BRoute needs to solve is matching the source-segment and the sink-segment of a path without direct measurement, i.e. identifying the dashed lines in the left graph of Figure 6.1. BRoute does this using AS-level path information. Intuitively, for a pair of nodes s and d, if we know all the upstream AS paths from s (called the AS-level source tree or $srcTree(s)$) and all the downstream AS paths toward d (called the AS-level sink tree or $sinkTree(d)$), then $Path(s, d)$ should pass one of their shared ASes. For example, the right graph of Figure 6.1 illustrates the upstream AS paths from a_0, and the downstream AS paths toward c_0. Assume that $A7$ is the only shared AS then this means that path $Path(a_0, c_0)$ must pass through $A7$, and we can use $A7$ to identify $srcSgmt(a_0, c_0)$ and $sinkSgmt(c_0, a_0)$. We will call the AS that is shared and on the actual path the common-AS. Of course, there will typically be multiple

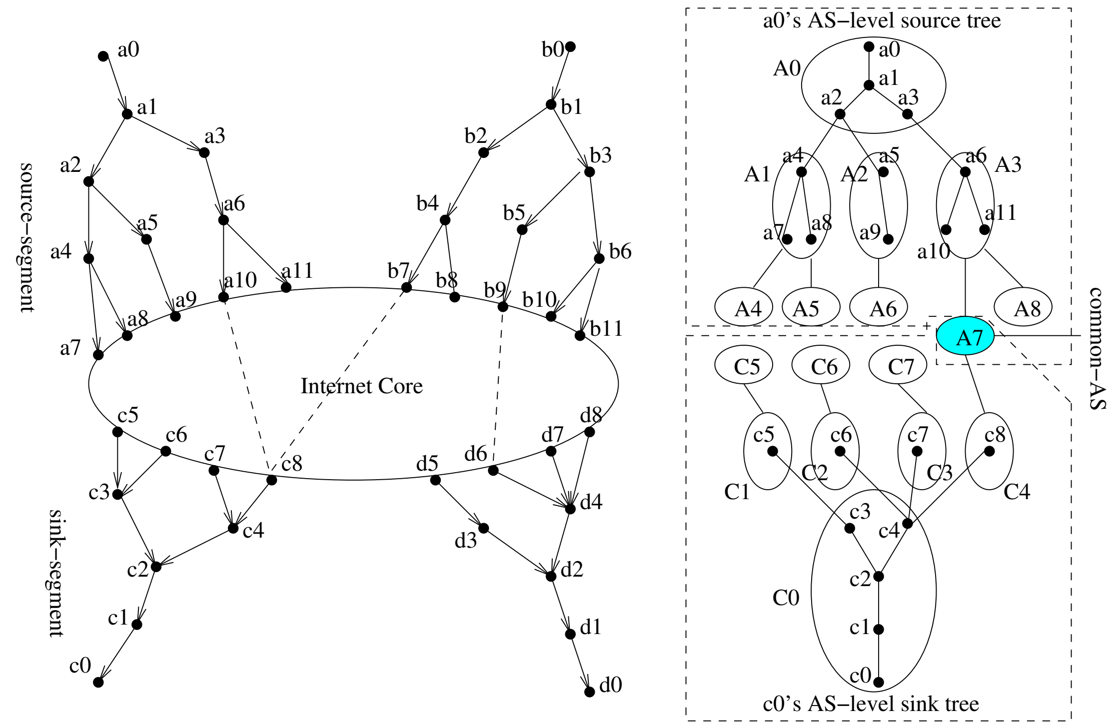


Figure 6.1: Source/sink-segments and AS level source/sink trees

shared ASes between $srcTree(s)$ and $sinkTree(d)$, we will discuss in Section 6.2.1 how to uniquely determine the common-AS.

6.1.3 BRoute Architecture

As a system, BRoute includes three components: system nodes, traceroute landmarks, and information exchange point. System nodes are Internet nodes for which we want to estimate available bandwidth; they are responsible for collecting their AS-level source/sink trees, and end-segment available bandwidth information. Traceroute landmarks are a set of nodes deployed in specific ASes; they are used by system nodes to build AS-level source/sink trees and to infer end-segments. An information exchange point collects measurement data from system nodes and carries out bandwidth estimation operations. It could be a simple central server, or a more sophisticated distributed publish-subscribe system. For simplicity we will assume it is a central server, and call it the *central manager*.

BRoute leverages two existing techniques: bottleneck detection [56] and AS relationship inference [50, 114, 29, 82]. Bottleneck detection is used to measure end-segment bandwidth, and AS relationship information is used to infer the end-segments of a path. Operations of BRoute can be split into the pre-processing stage and the query stage, as illustrated in Figure 6.2:

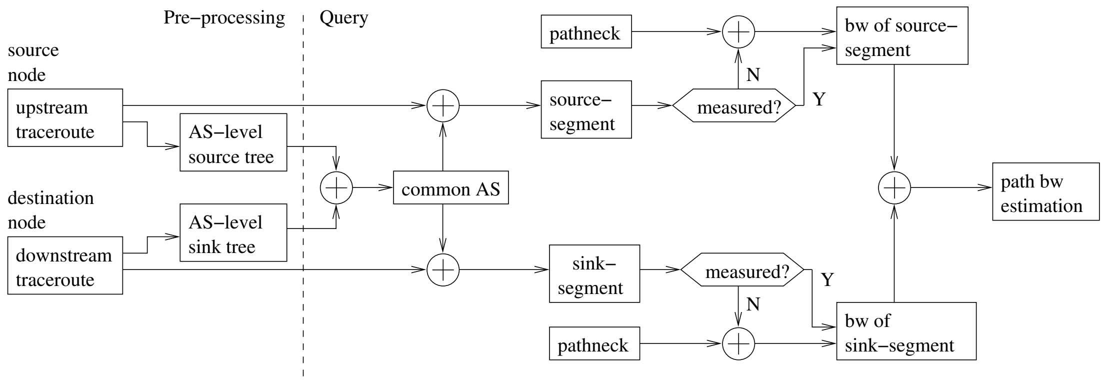


Figure 6.2: BRoute System Architecture

- **Pre-processing:** In this stage, each system node conducts a set of traceroute measurements to the traceroute landmarks. At the same time, traceroute landmarks also conduct traceroutes toward system nodes. The system node then uses the traceroute information to construct AS-level source and sink trees. Next the system node identifies its source-segments and sink-segments and uses Pathneck to collect bandwidth information for each end-segment. This information is reported to the central manager.

- **Query:** Any node can query BRoute for an estimate of the available bandwidth between two system nodes—s to d. The central manager will first identify the common-AS between $srcTree(s)$ and $sinkTree(d)$. The common-AS is used to identify the end-segments $srcSgmt(s, d)$ and $sinkSgmt(d, s)$ of $Path(s, d)$, and the central manager then returns the smaller of the available bandwidths for $srcSgmt(s, d)$ and $sinkSgmt(d, s)$ as the response to the query.

A distinguishing characteristic of BRoute is that it uses AS-level source/sink tree computation to replace most of end-node network measurements, thus *shifting the system overhead from expensive network measurement to cheap and scalable local computation*.

6.2 End-Segment Inference

In this section, we explain the two key operations of BRoute: how to pick the common-AS, and how to use the common-AS to identify the source-segment and sink-segment of a path. For each operation, we first define the algorithm, and then evaluate its performance. We keep using the five data sets described in Section 5.2 for the evaluations.

6.2.1 Selecting the Common-AS

Algorithm 1 Common_AS

1: **if** $AS(s) = AS(d)$ **then**
2:     return $AS(s)$ as the common-AS;
3: **end if**
4: $SET \leftarrow srcTree(s) \cap sinkTree(d)$;
5: **if** $SET = NULL$ **then**
6:     $SET \leftarrow srcTree(s) \cup sinkTree(d)$;
7: **end if**
8: **if** $\{AS(s), AS(d)\} \subseteq SET$ **then**
9:     return both $AS(s)$ and $AS(d)$ as the common-AS;
10: **else if** $AS(s) \in SET$ or $AS(d) \in SET$ **then**
11:     return $AS(s)$ or $AS(d)$ as the common-AS;
12: **else**
13:     remove all the ASes which have at least one ancestor AS (in either $srcTree(s)$ or $sinkTree(d)$) also in SET;
14:     **for all** $A \in SET$ **do**
15:         $p(A) \leftarrow cnt(A, s)/total(s) + cnt(A, d)/total(d)$;
16:     **end for**
17:     return the A which has the largest $p(A)$;
18: **end if**

Algorithm

Typically, an AS-level source tree and an AS-level sink tree share multiple ASes, and we need to choose one of them as the common-AS. Our selection method is shown in Algorithm 1. s and d denote the two end nodes, and $AS(s)$ and $AS(d)$ denote the ASes s and d belong to, respectively. On line 15, $cnt(A, s)$ denotes the number of upstream/downstream AS paths from s that pass AS A in $srcTree(s)$, while $total(s)$ denotes the total number of AS paths in $srcTree(s)$. $cnt(A, d)$ and $total(d)$ are defined similarly.

The heart of the algorithm is on lines 13-17. The first step is based on the fact that most AS level routes follow the shortest AS path [82]. As a result, the algorithm searches for the shared ASes that are closest to the root ASes in both $srcTree(s)$ and $sinkTree(d)$ (line 13). It does this by eliminating from the set of shared ASes (SET) all ASes which have a shared AS on the same AS-level source or sink tree branch closer to the root AS. In other words, if in an AS-level source/sink tree, both AS A and its child AS B are in SET, the AS path should only pass A since that produces a shorter AS path, so B is dropped. In the second step, if the resulting set of shared ASes still has multiple candidates, we pick the one that has the highest probability to appear on $Path(s, d)$ based on the $p(.)$ value (line 14-17).

There are several border cases where one or both root ASes are a shared AS. They include: (1) node s and d are in a same AS (line 1-3); (2) both $AS(s)$ and $AS(d)$ are shared ASes (line 8-9); and (3) either $AS(s)$ or $AS(d)$, but not both, is a shared AS (line 10-11). For these cases, the algorithm return the root AS(es) as the common-AS. In particular, case (2) returns two ASes as the common-ASes.

The last border case is when SET is empty (line 5). In practice, BRoute uses measurements for a limited number of traceroute landmarks to construct the AS-level source/sink trees. Although we will show that this method can cover most ASes in an AS-level source/sink tree, some unpopular ASes could be missing, and as a result, we may not find a shared AS. For this case, we consider all the ASes in both trees as shared (line 6). Similarly, $cnt(.)$ and $total(.)$ are in practice computed based on measurement, so their absolute values may differ from those derived from a complete tree. We quantify the impact of this sampling in Section 6.4.

Evaluation

Given the data we have, we can use two methods to validate Algorithm 1. The first method is to build both AS-level source and sink trees as described in Section 5.3 and to apply Algorithm 1. This is the most direct method and we will use it in our case study in Section 6.4. This method however has the drawback that it is based on limited downstream data, the AS-level sink trees can be incomplete. In this section we use a different method: we evaluate the algorithm using the $srcTree(d)$ to replace the incomplete $sinkTree(d)$. The basis for this method is the observation that the AS-level trees are only used to determine the end-segments of a path (we do not need the AS-level path itself), and the AS-level source tree may be a good enough approximation of the AS-level sink tree for this restricted goal. This in fact turns out to be the case, as we show below. This approach has the advantage that we have much larger data set to work with.

Using the *BGP* data set, we first construct an AS-level source tree for each vantage point, we then infer the common-AS for each pair of vantage points, and we finally compare the result with the actual AS paths in BGP tables. To make sure we have the correct path, we exclude those AS paths whose last AS is not the AS of the destination vantage point. For the resulting 15,383 valid AS paths, the common-AS algorithm selects the wrong common-AS for only 514 paths, i.e. the success rate is 97%. For the 14,869 correctly inferred common-ASes, only 15 are not top AS, which confirms our intuition that the common-AS inferred by Algorithm 1 is typically a top-AS where the maximal uphill and downhill paths meet.

Top-AS Symmetry

The high accuracy of common-AS inference shows that we can indeed replace the AS-level sink tree of the destination with its AS-level source tree in Algorithm 1. This has an interesting implication: it does not matter which node is the source or the destination, or, in other words, the common-AS is the same in both directions. Since the common-AS is almost always a top-AS, it is also "symmetric": for two nodes s and d, $Path(s, d)$ and $Path(d, s)$ share at least one top-AS.

This observation can indeed be confirmed by our data. In the set of 15,383 AS paths we picked above, there are 3503 pairs of AS paths that connect two nodes in both directions. Among them, 2693 (77%) pairs have symmetric AS paths, which obviously also share top-ASes. In the remaining 810 pairs, 486 of them share at least one top-AS. Overall, the top-AS symmetry property applies to 91% pairs. A similar analysis of the *Planetlab* data set yields very similar resuls: 67% of the node pairs have symmetric AS paths, while 92% of the pairs have a symmetric top-AS. Note that this result does not contradict the observation made by Mao et.al. [82] that a large percentage of AS paths are asymmetric, since we focus on the top-AS only.

It is important to point out that while the top-AS symmetry property provides another method for identifying common-AS, AS-level sink tree information is still needed by BRoute to obtain sink-segment information.

6.2.2 End-Segment Mapping

Given that the AS-level source and sink trees closely follow a tree structure, the common-AS can be easily used to identify a unique branch in both the AS-level source and sink trees. We now look at how well this tree branch can be used to determine IP-level end-segment of the path. Our analysis focuses on source-segments because we have complete data for them.

Ideally, for any AS $A \in srcTree(s)$, we would like to see that all upstream paths from s that pass A share a same source-segment e. If this is the case, we say A is *mapped* onto e, and every time A is identified as the common-AS, we know the source-segment of the path is e. In practice, upstream paths from s that pass A could go through many different source-segments, due to reasons such as load-balance routing or multihoming. To quantify the differences among the source-segments that an AS can map onto, we define the *coverage* of source-segments as follows. Suppose AS A is mapped to $k (k \geq 1)$ source-segments $e_1, e_2, ..., e_k$, each of which covers $n(e_i)(1 \leq i \leq k)$ paths that pass A. The coverage of e_i is then defined as $n(e_i)/\sum_{i=1}^{k} n(e_i)$. If we have $n(e_1) > n(e_2) > ... > n(e_k)$, then e_1 is called the top-1 source-segment, e_1 and e_2 are called the top-2 source-segments, etc. In BRoute, we use 0.9 as our target coverage, i.e., if the top-1 source-segment e_1 has coverage over 0.9, we say A is mapped onto e_1.

We use the *Rocketfuel* data set to analyze how many end-segments are needed to achieve 0.9 coverage. For the 30 AS-level source trees built from this data set, there are totally 1687 ASes (a same AS in two different trees are counted twice). Figure 6.3 shows the percentages of ASes (y-axis) that achieve various levels of coverage (x-axis) from top-1 and top-2 source-segments. Of the 1687 ASes, 1101 are mapped onto a single

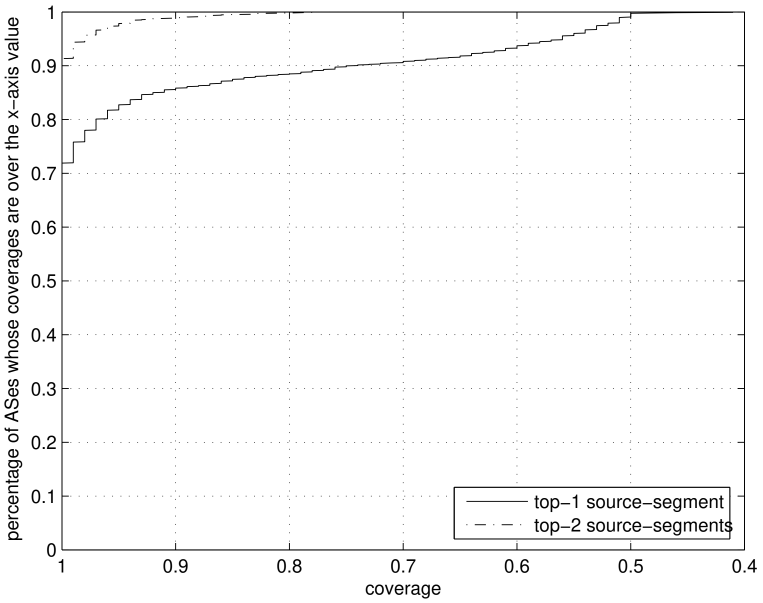


Figure 6.3: Coverage of top-1 and top-2 source-segments

source-segment (i.e. coverage of 1), while 586 (from 17 trees) are mapped onto multiple source-segments. Among these 586 ASes, using the 0.9 coverage threshold, 348 can be covered using the top-1 source-segment, so in total (1101 + 348 = 1449) (85%) ASes can be mapped onto an unique source-segment. If we allow an AS to be covered using the top-2 source-segments, only 2% (17 ASes) can not be covered, i.e., 98% of the ASes can be mapped onto top-2 source-segments. We conclude that an AS can be mapped onto at most two source-segments in most times.

Detailed checking shows that among the 586 ASes that are mapped onto multiple source-segments, 302 (51%) map to source-segments whose 4th hops are in the same AS. We speculate that this use of multiple source-segments is due to load-balance routing. More work is needed to determine why the other 284 ASes use multiple source-segments. So far we have focused on the common case when the common-AS for a path is not one of the two root ASes. If one or both root ASes are returned as a common-AS, the algorithm for selecting the source-segment or sink-segment still applies. The hard cases are when the source and sink node are in the same AS, or when the source or sink node are in a tier-1 AS; more research is need to deal with these two cases.

Using the *Planetlab* data set, for which we can get a large number of downstream routes for each node, we also looked at the sink-segment uniqueness. We found that the above conclusion for source-segments also applies to sink-segment. Among the 99 nodes that have at least 100 complete downstream routes, 69 (70%) nodes have at least 90% of the ASes in their AS-level sink tree mapped onto top-1 sink-segment, while 95 (96%)

nodes have at least 90% of their ASes mapped onto top-2 sink-segments.

With these results of end-segment uniqueness, an AS in an AS-level source/sink tree should be mapped onto top-1 or top-2 end-segments. In the first case, we return the available bandwidth of the top-1 end-segment. In the latter case, we return the average of the available bandwidth of the two top-2 end-segments in the path bandwidth estimation. This method should work well if the reason for having top-2 end-segments is load-balance routing, since the traffic load on both end-segments is likely to be similar.

6.3 End-Segment Bandwidth Measurement

BRoute uses Pathneck to measure end-segment bandwidth. The reason is that Pathneck can both pinpoint the location bottleneck and provide upper or lower bounds for links on the path. For example, Pathneck provides an upper bound for the available bandwidth on the bottleneck link; this upper bound is in general quite tight [56]. It can similarly provide useful bounds for links upstream from the bottleneck. However, an unfortunate feature of Pathneck is that it provides little information about the links past the bottleneck link. Specifically, Pathneck can only provide a lower bound for the available bandwidth on those links and that bound can be very loose.

These Pathneck properties have the following implications for the measurement of end-segment bandwidths in BRoute. If the bottleneck is on the source-segment (or in the core), Pathneck can provide a (tight) upper bound for the source-segment but only a (loose) lower bound for the sink-segment. If the bottleneck is on the sink-segment, Pathneck can provide a (tight) upper bound for both the source and sink segment. In other words, any node can easily measure the available bandwidth on its source-segments. However, to measure the sink-segment bandwidth, nodes need help from another node, ideally a node with high upstream bandwidth.

BRoute can collect end-segment bandwidths in two modes: peer-to-peer or infrastructure. In the peer-to-peer mode, end-segment bandwidth are measured by system nodes themselves in a cooperative fashion. That is, each system node choose a subset of other system nodes as peers to conduct Pathneck measurements with, so as to cover all its end-segments. If we use sampling set to denote the set of paths on which Pathneck measurements are conducted, it should cover all the source-segments and sink-segments in the system to be useful. Theoretically, selecting the sampling set can be transformed to the classical graph problem on edge covering, which is NP-hard [39].

We use a simple greedy heuristic to find this set. For each node s in the system, we count its number of un-measured source-segments (sink-segments) as $srcCnt(s)$ ($sinkCnt(s)$). We then pick the path $Path(s, d)$ that has the largest $(srcCnt(s) * sinkCnt(d))$ value, put it in the sampling set, and label $srcSgmt(s, d)$ and $sinkSgmt(d, s)$ as measured. This process is repeated until all end-segments in the system are measured. The intuition behind the algorithm is as follows. If a node has a large number of end-segments, they are shared by fewer number of paths, and are thus less likely to be covered by many paths, so they should be given a higher priority for being picked for the sampling set. By picking the path that has the largest $(srcCnt(s) * sinkCnt(d))$ value, we achieve exactly that. In the case study presented in Section 6.4, this algorithm finds a sampling set that only includes 7% of all paths, which shows it is indeed effective.

The peer-to-peer operation has the advantage that the bandwidth measurement overhead is shared by all system nodes, so the system scales naturally. However, the cost is that it requires interactions between system nodes. This introduces complexity in a number of ways. First, as new nodes join and leave (including fail), it is necessary to continuously determine what node pairs need to perform measurements. Another issue is that some (or even many) system nodes in the system may not have sufficient downstream bandwidth to accurately measure the available bandwidth on sink-segments of other system nodes, as was explained earlier in the section. This can significantly impact the accuracy of BRoute.

The solution for these problems is to use a measurement infrastructure in the form of bandwidth landmarks that cooperate with system nodes to measure end-segment bandwidth. The bandwidth landmarks can share the same set of physical machines with the traceroute landmarks. In this infrastructure mode, a system node uses its AS-level source tree to pick a subset of bandwidth landmarks to measure its end-segment bandwidth. The system node will use Pathneck to measure source-segment bandwidth, using the bandwidth landmarks as destinations. Similarly, the bandwidth landmarks, at the request of the system, will measure the sink-segment bandwidth using the system node as Pathneck's destination. The infrastructure mode completely removes system-node dependences, which makes the system robust to individual node failures. Note that in order to be effective, bandwidth landmarks should have high upstream bandwidth, so that in general during measurements, the sink-segment of the system node will be the bottleneck.

The problem with using bandwidth landmarks is of course that each bandwidth landmark can only support a limited number of system nodes, so the number of bandwidth landmarks will have to grow with the number of system nodes. For example, assume each system node has on average 10 sink-segments and assume bandwidth is measured once per hour. By default, each Pathneck measurement needs 60 500B packets, or 30K byte, so each system node contributes rate of only about 670 bit/sec in measurement overhead. This means that a bandwidth landmark with a dedicated Internet connection of 100 Mbps should be able to support at least 100K system nodes.

6.4 Overall Inference Accuracy

In the previous sections we describe each step in the BRoute bandwidth estimation process and evaluated it in isolation. While the error introduced in each step is relatively small, these errors can accumulate, so it is necessary to evaluate the performance of the entire system in terms of its end-to-end accuracy. We decided to run a case study on Planetlab,

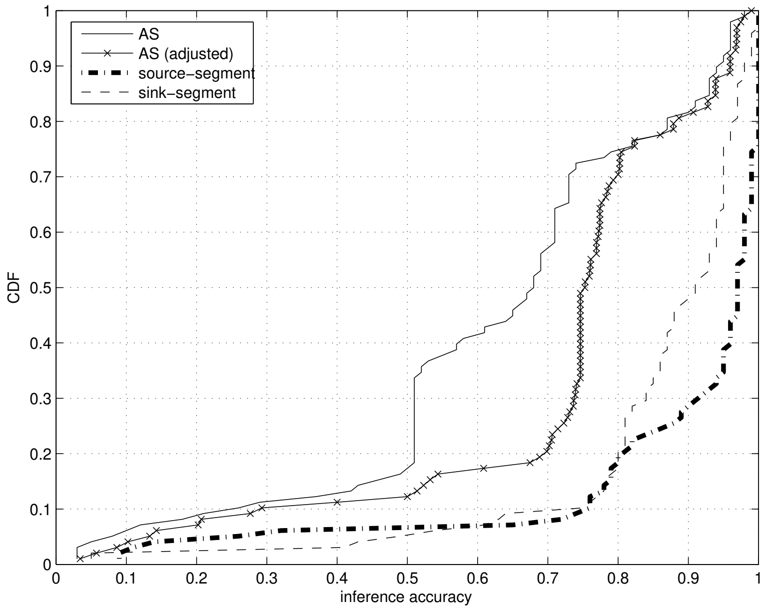


Figure 6.4: Common-AS and end-segment inference accuracy in the Planetlab case study.

using the peer-to-peer mode operation for BRoute. Using Planetlab does have some known disadvantages: (1) Planetlab nodes are generally well connected, and may not be representative for regular network users; (2) Planetlab nodes can have very high load, which can impact bandwidth measurement accuracy. Even so, Planetlab is by far the best evaluation testbed available due to the scale requirement of BRoute. In this section, we first describe our experimental methodology, we then present the results for the end-segment inference and the bandwidth estimation accuracy.

6.4.1 Data Collection and End-Segment Inference

On Planetlab, we select one node from 160 different sites. Each node runs traceroute to all the other 159 nodes and the 237 tier-1 and tier-2 traceroute landmarks selected in Section 5.4, and uses the information to build the AS-level source and sink trees. Finally, each node conducted Pathneck measurements to all the other 159 nodes in the system to measure path bandwidth.

Using the traceroute data, we repeat the analysis described in Sections 6.2.1 and 6.2.2 to study the common-AS and end-segment inference accuracy. The difference is that we use both AS-level source tree and AS-level sink tree in the inference, i.e., we strictly follow Algorithm 1. Here AS-level source trees are constructed using the traceroute results from Planetlab nodes to the traceroute landmarks, while AS-level sink trees are built using the downstream routes from the other system nodes.

Figure 6.4 summarizes the inference results. Each point in this figure corresponds to a Planetlab node. That is, we group the paths that share the same source node, and then compute the percentage of paths for which the common-AS and end-segments are inferred correctly. The solid curve plots the distribution of the common-AS inference accuracy. We can see it is much worse than those presented in Section 6.2.1. The reason turns out to be very simple: because of the limited amount traceroute information (160 nodes is not a lot), the AS-level sink trees are incomplete, so the common-AS algorithm sometimes can not find a shared AS. If we ignore these cases, the inference accuracy improves significantly, as shown by the curve marked with "x": 80% of sources have common-AS inference accuracy of over 0.7. This level of accuracy is still worse than the results presented in Section 6.2.1. The reason is that the $p(.)$ values used in the common-AS algorithm are based on limited data, so they are inaccurate, and again the limited number of nodes used in the study negatively impacts accuracy. These results suggest that, in peer-to-peer mode, the accuracy of BRoute will increase with the size of the system.

The dashed and dash-dot curves plot the distributions of the inference accuracy for source-segments and sink-segments, respectively. We can see that the end-segment inference is much more accurate than the common-AS inference, with around 50% and 70% of paths having inference accuracy of over 0.9. This is not surprising since different ASes could map to the same end-segment, so an incorrect common-AS inference does not necessarily result in a wrong end-segment inference.

6.4.2 Bandwidth Inference

Next we used the peer-to-peer mode to estimate end-segment bandwidth. To estimate bandwidth inference accuracy, we divide the data into a sampling set and an evaluation set. We use the algorithm described in Section 6.3 to select the sampling set. The algorithm identifies a sampling set with 753 paths, which are 7% of the 10,779 paths for which we have enough route data to identify both source- and sink-segments. With the path bandwidth inferred using end-segment bandwidth, we compare them with the real Pathneck measurements that are not in the sampling set. The accuracy is measured in terms of their relative difference: $(BW_{inferred} - BW_{measured})/BW_{measured}$. Figure 6.5 plots the distribution of the relative differences for all paths in the evaluation set. We can see that 30% of the estimates are higher than the measured value, while the other 70% are lower; this is mainly due to the fact that we only have lower bounds for these paths. Overall, around 80% of the paths have less than 50% of difference. Considering that bandwidth measurement generally have a 30% of measurement error [58] and Planetlab nodes' high load can interfere with bandwidth measurement, which we believe have negative impact on our evaluations, we regard 50% of estimation error as a promising result in terms of inference accuracy.

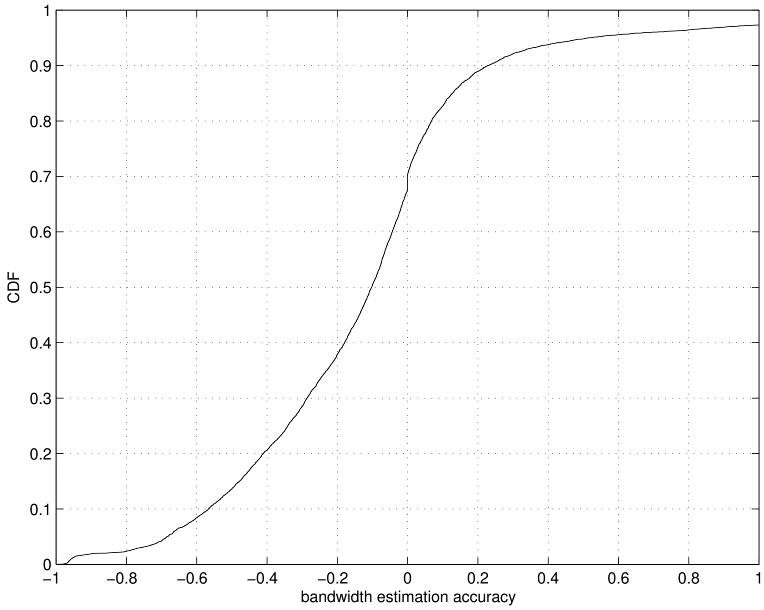


Figure 6.5: End-to-end bandwidth inference accuracy

6.4.3 Discussion

The results in this section highlight several aspects of the BRoute system that need improvements. First, we need to improve BRoute end-segment inference accuracy, for example, by integrating the latest AS path and relationship inference techniques, such as developed by Mao et.al. [82]. Second, we need to explore path available bandwidth measurement techniques based on bandwidth landmarks, which provide access to both end points, for example, by combining IGI/PTR with Pathneck. Third, we need to develop algorithms to select traceroute landmark locations for AS-level tree construction. Finally, it would be useful to understand the interaction and cooperation between multiple BRoute systems, which might help multiple applications to participate and potentially collaborate with each other.

6.5 System Issues

As a large scale network system involving many nodes, BRoute needs to address many practical system issues. In this section we briefly discuss two system aspects: system overhead and system security.

6.5.1 System Overhead

The system overhead of BRoute is determined by two factors: the overhead of each measurement instance, and the frequency of information updates. Here one instance includes the measurements for both AS-level source/sink tree and end-segment available bandwidth. In reality, application query introduces another overhead factor, we do not consider it since it is application specific.

The AS-level source/sink trees are constructed based on the traceroute measurements to/from a number of traceroute landmarks. Based on the preliminary results in Section 5.4, we estimate that each system node will need to execute no more than 1000 traceroute probes; the same number of probes are needed from traceroute landmarks to the system node. This overhead is very small. It has been shown that an optimized traceroute tool such as that used by Spring et.al. [108] can finish 120K traceroutes in 6 hours, which translates to 3 minutes for 1000 traceroute probes. The amount of probing packets is around 3M byte (assuming each path has 25 hop, each hop is measured twice).

The AS-level source trees could also be built using local BGP table. The sizes of local BGP tables seem to be generally less than 10MB, which is tolerable since it only involves local communication. Note that even if we use BGP tables for AS-level source tree construction, traceroute is still needed to obtain (IP-level) end-segment information, but fewer traceroute probes are needed in this case since the AS-level source tree provides an upper bound for the number of end-segments.

The overhead for end-segment bandwidth measurements is also fairly small. For the *Rocketfuel* data set, we found the median number of source-segments is around 10. Given that the Pathneck measurement overhead is 30K byte (500byte/pkt * 60pkt), the total available bandwidth measurement overhead for one node is 300K byte for upstream measurements. Similar overhead is expected for downstream measurements.

For long term system monitoring, the overhead is also affected by the system information update frequency. This parameter is ultimately determined by how quickly AS paths and end-segment bandwidths change. The end-segment available bandwidth change frequency is determined by traffic load, which has been shown to change fairly quickly [90]. For this reason, end-segment available bandwidth information needs to be updated frequently, for example, once per hour. The exact frequency should be set by the system node.

For AS paths, recent results from Zhang et.al [99] have shown that the BGP routes associated with most traffic are reasonably stable. Now we look at whether AS-level source trees also have this level of stability. We downloaded BGP tables from Route Views Project and RIPE RIS Project on different dates, and compared them with the BGP tables on 01/10/2005. The change metric is defined as the percentage of prefixes whose maximal uphill path has changed. Figure 6.6 summarizes the results. The top graph plots the long-term change results: we compare the BGP tables on mm/01/2004 (mm=03, ..., 12) with those on 01/10/2005. The x-axis indicates the month, the y-axis is the percentage of

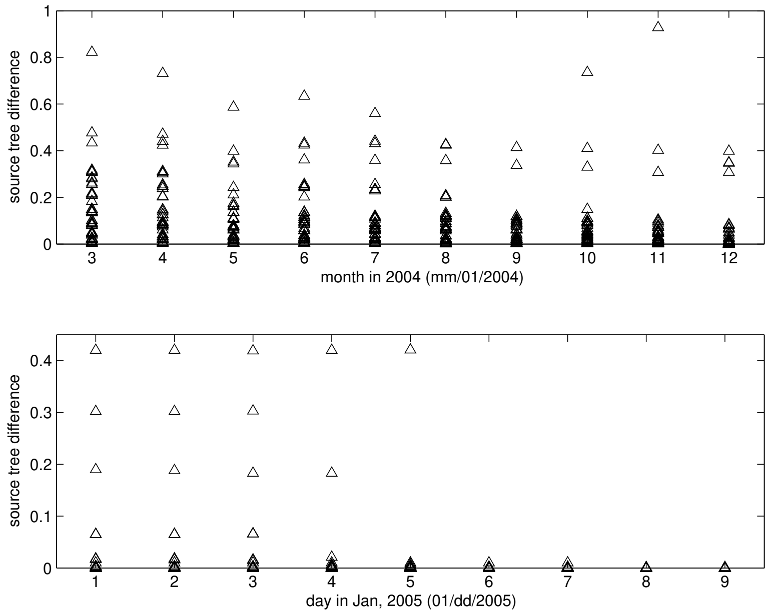


Figure 6.6: AS-level source tree changes with time

prefixes whose maximal uphill path changed, and each triangle corresponds to one peering point. While some peering points experience 40-80% of changes, most are less than 15%, which we think is reasonably small considering the long time interval. The bottom graph focuses on the short-term change: we compare the BGP tables on 01/dd/2005 (dd=01, ..., 09) with those on 01/10/2005. Not surprisingly, the changes are significantly less—on each day, at most 4 points have a change larger than 4%, and all the others are below 4%. We conclude that in general it is safe to update the AS-level source tree once a day. We expect AS-level sink trees to have similar stability.

One case that is not considered in this section is dynamic routing enabled by some multihoming devices. Unlike static multihoming, where routes are pre-configured and do not change frequently, these devices dynamically change data forwarding path based on real-time measurement. A final point is that, since measurements are under the control of the system nodes, not all nodes have to use the same system parameters. For example, system nodes that have historically seen more frequent route or bandwidth changes can run the measurements more frequently. Similarly, system nodes can adjust the length of their end-segments, depending on the bottlenecks they observe. Finally, all the routing and bandwidth information can easily be shared by system nodes in the same AS, thus providing another opportunity to reduce overhead.

6.5.2 System Security

Like all systems running on the Internet, BRoute is also subject to attacks. A detailed discussion on how to deal with attacks is beyond the scope of this dissertation. Here we only discuss attacks related to malicious system nodes. This includes two types of attacks: false traceroute measurements and false end-segment bandwidth measurements. The first attack only has local impact: it only affects the estimates for paths involving the malicious node.

The second attack has a more severe impact when BRoute measures end-segment bandwidth in a peer-to-peer fashion. In the peer-to-peer mode, if a node reports false Pathneck measurement, it not only affects its source-segment bandwidth, but it also affects the sink-segment bandwidth of the destination node. Since sink-segment bandwidth results are shared, this false information can quickly propagate to other paths. One method to deal with this type of abuse is to select a sampling set such that each sink-segment is covered at least twice. That allows BRoute to double check the bandwidth measurement of each sink-segment. If the estimates from one node are consistently different from those of other nodes, it can be labeled as suspicious. Of course, a malicious node can make detection very difficult by only falsifying a small number of bandwidth measurements. The cleanest solution is to use (trusted) bandwidth landmarks, as discussed in Section 6.3. In infrastructure mode, falsified information from a malicious node will only affect the paths involving that node.

6.6 Summary

In this chapter, we presented the design and the system architecture of BRoute—a large scale available-bandwidth inference system. We described the two key operations used by the BRoute system: how to select the common-AS and how to use the common-AS to identify end segments. We showed that in 97% of cases we can identify the common-AS correctly, and in 98% of cases a common-AS can be mapped onto at most two different end segments, with 85% of cases mapping to an unique end segment. The overall end-to-end available bandwidth estimation accuracy is evaluated on Planetlab, where we show that 80% of inferences have inference error within 50%. The relative high error rate is due to the fact that we can not get enough number of traceroute landmarks, and due to the difficulties of obtaining accurate bandwidth information on the high-loaded Planetlab nodes. For this reason, we regard the performance of BRoute as encouraging.

Chapter 7

Topology-Aware Measurement Infrastructure

Compared with popular active measurement techniques like ping or traceroute, available bandwidth measurement techniques (like IGI/PTR) and systems (like BRoute) are harder to use. First, they often require the cooperation of non-local end nodes. For example, the IGI/PTR tool requires access to both the source and the destination of a path, and the BRoute system needs a diverse set of vantage points to serve as traceroute landmarks. However, regular end users often do not have such access. Second, available bandwidth measurement techniques are often more sensitive to configuration details than ping or traceroute, and correctly setting the configuration parameters requires good understanding of the technical details, which in many cases is not trivial. Third, available bandwidth measurement techniques generally use packet trains. If not well coordinated, packet trains used by different end users can interfere with each other and introduce measurement errors. To address these issues, we developed the Topology-Aware Measurement Infrastructure (TAMI).

Compared with existing measurement infrastructures like NWS [118], NIMI [94] and Scriptroute [110], TAMI has two distinguishing characteristics—measurement scheduling functionality and topology-awareness. The measurement scheduling functionality controls the execution times of measurement requests. It can parallelize, synchronize, and serialize measurements from different vantage points, both to improve overall system response times and to avoid measurement interferences. Topology-awareness refers to the capability of TAMI in supporting tree-related operations that are used by the BRoute system, such as measuring the source and sink trees of an end node, identifying the common-AS of two AS-level source or sink trees, etc.

We use the term "topology-awareness" instead of the more specific term "tree-awareness" because TAMI may eventually also support other topology related measurement techniques such as tomography. Topology-awareness is important for advanced network applications that need measurements from a set of network paths. We call such measurements *topology-aware measurements* because to correctly select these paths and avoid measurement interferences we need network topology knowledge. Two typical examples of topology-aware measurements are source and sink tree measurements and tomography measurements. Source and sink trees are the key data structures used in the Doubletree algorithm [44] and the BRoute system (Chapter 6). The DoubleTree algorithm takes advantage of the tree structure to reduce measurement overhead in network topology discovery, while the BRoute system uses source and sink trees to exploit bottleneck sharing for large-scale available bandwidth estimation. As shown later, this tree view of end nodes is also an effective way of diagnosing routing events. Tomography [31] infers link-level performance information by correlating multiple path-level measurements. It is an important active measurement technique for end-users to obtain link-level performance data.

At the system level, TAMI also provides the following benefits: (1) it can reduce the burden on application developers by allowing them to focus on application specific tasks, (2) it can improve applications' performance by providing highly efficient measurement modules, (3) it can better utilize network resources by sharing measurement data, and (4) it can encourage innovation by providing applications a platform to quickly try new methods using complex network performance data.

In this chapter, we first describe the architectural design of TAMI. We then present the implementation and performance evaluation of the TAMI system. We will use three applications to demonstrate TAMI's functionalities.

7.1 TAMI Architecture

To explain the TAMI architecture, we first describe the TAMI functional modules and their interactions. We then present three important design choices that affect the TAMI implementation for different application scenarios. We use three representative deployment architectures to illustrate their impact. For naming convenience, starting from this section, we also refer to vantage points as agents.

7.1.1 TAMI Functional Modules

Figure 7.1 illustrates the TAMI functional modules and their interactions. The three key modules are the topology, scheduling, and agent-management modules, which together enable the topology-aware measurements. It is these three modules that distinguish TAMI from other measurement infrastructures.

The topology module maintains the topology information of the network, and coordinates path-level measurements for topology-aware measurement requests. For example, for a sink tree measurement request, the topology module will select a set of agents, and submit the measurement request to the scheduling module. Once the measurements are done, it will organize the measurement results in a tree data structure. In some cases, this

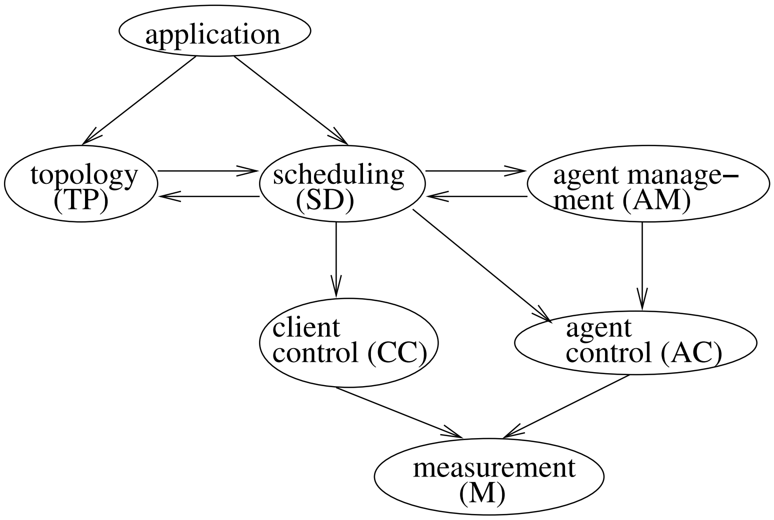


Figure 7.1: TAMI functional modules

module also provides guidance to the scheduling module, for example, to avoid measurement interference or overloading a network link in a multi-path measurement.

The scheduling module decides the execution order of all TAMI measurements. It has the following four capabilities. First, it synchronizes the measurements on different paths that are used to satisfy a same request, so that the measurement data can be correlated. Second, it serializes the measurements from different vantage points to a same destination to avoid measurement interferences. Third, it parallelizes measurements to reduce system response time. Finally, when there are multiple requests in the system, it guarantees fairness. The scheduling module achieves these capabilities by controlling both the clients (through the client-control module) and the agents (through the agent-management and the agent-control modules).

The agent-management module deals with agent specific information, specifically, the membership of active agents. In a deployed system, if there are a large number of agents, keeping track of the status of agents could be cumbersome and may need a separate functional module to take care of it. When the number of agents is small, the agent-management module can be combined with the scheduling module.

The functionality of the other modules is similar to those used in more traditional measurement infrastructures like NIMI [94]. The agent-control and the client-control modules manage low-level measurement operations based on the control messages from the scheduling module. The measurement module simply implements required measurement techniques, and conducts the measurements when needed.

Application requests enter TAMI through either the topology module or the scheduling module, depending on whether topology information is needed. When the request needs the topology information, such as a tree measurement, it should be submitted to the topology module. Otherwise, if the request only needs measurement scheduling support, such as a full-mesh delay measurement among the agents, it should be directly sent to the scheduling module.

7.1.2 TAMI Deployment Architectures

The functional modules in Figure 7.1 illustrate the necessary functionality that TAMI should have, but it does not specify how the functional modules should be mapped to real system software components. This mapping is what we call the *deployment architecture*. The important factors that affect the deployment architecture include:

- **Centralized v.s. distributed**. Some TAMI modules like the agent-control and the measurement modules obviously should be distributed on the agents. Other modules, like the topology and the scheduling modules, can be implemented either in a centralized controller, or in the distributed agents. Both centralized design and distributed design have their advantages and disadvantages. A centralized design simplifies scheduling operations, and provides a platform to aggregate and share data; while a distributed design scales better and has no single-point failure. The optimal design depends on the application scenario.

- **Trust model**. The trust model refers to the relationship of the nodes in the system, for example, whether or not they belong to the same organization and the level of trust among them. It determines the level of security that is needed in the corresponding system design. Obviously, the trust model is also application specific. For example, a TAMI system used by a private network can assume a much stronger trust model than that used on Internet.

- **Relationship between application and TAMI**. TAMI as an infrastructure can be implemented either inside or outside an application. When implemented inside an application, TAMI functional modules are also components of the application and they can be easily customized, thus having much more flexible interactions with the application. On the other hand, if implemented outside the application, the application interface is an important implementation consideration for TAMI.

To illustrate the impact of these three factors, we present three TAMI deployment architectures for three representative application scenarios: P2P, ISP, and Public-Service (see Figure 7.2). The P2P scenario (in Figure 7.2(a)) represents highly distributed environments, where peers have complete control over their behaviors. In the corresponding deployment architecture, all TAMI functional modules are implemented inside the peers. The scheduling modules on different agents use distributed messaging protocols like gossip or epidemic protocols to manage the agents. In this architecture, the agent-management module can be integrated into the scheduling module since it only controls one agent; the

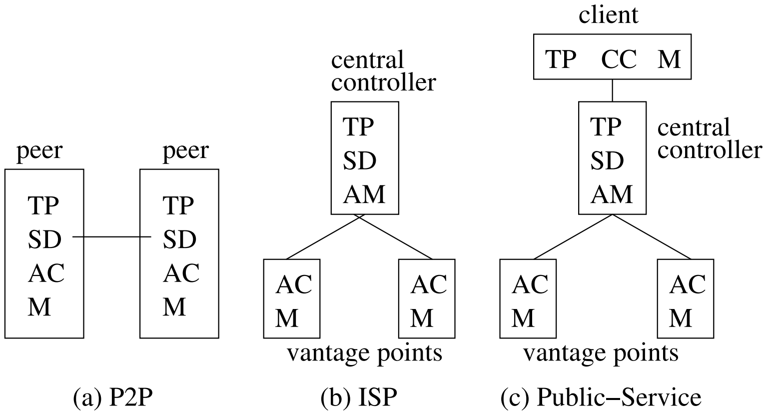


Figure 7.2: Deployment Architecture. The acronyms are taken from Figure 7.1.

client-control module is not needed either, because TAMI is implemented inside the applications. The trust model in this scenario is that of P2P applications.

The ISP scenario (in Figure 7.2(b)) represents a more centralized environment. An ISP generally needs a complete picture of the network status, thus a centralized deployment architecture is the natural choice. In such a scenario, the vantage points are purely used for measurement, therefore they are kept as simple as possible, with only the measurement and the agent-control modules installed on them. The other modules are maintained in the central controller. In this scenario, applications can be tightly integrated into the central controller, so the client-control module may not be needed. Also since both the agents and the central controller are deployed and used by the same organization, we can use a trust module appropriate for that organization.

The Public-Service scenario (in Figure 7.2(c)) represents the classic client-server scenarios. This scenario supports external applications, and the client-control module is needed to provide an application interface. Functionalities in the topology module can be split into both the central controller and the client. Details of the splitting depend on the application. One example is to let the central controller implement topology constructing and storing functions, while the client implements the operations that *use* the topology information. Finally, since external applications can control the measurement operations, security is a major challenge.

The TAMI prototype described in the next section is based on the Public-Service deployment architecture, mainly because it matches best with the networks we have access to.

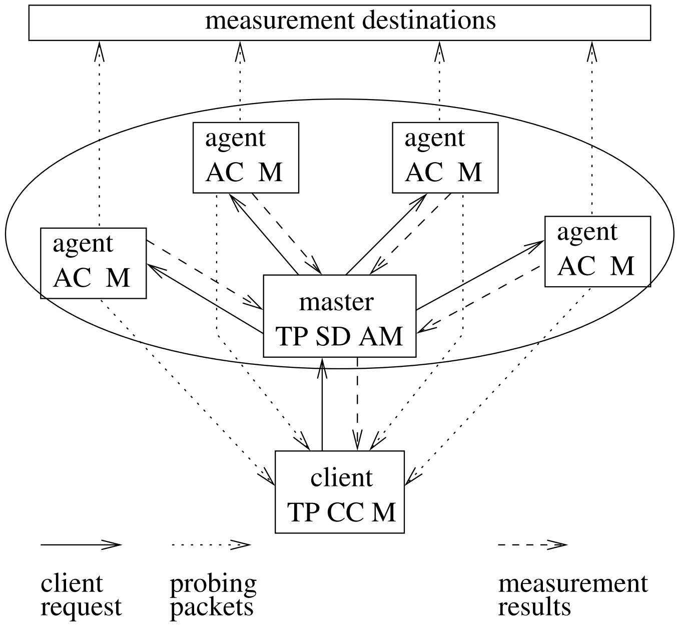


Figure 7.3: TAMI prototype system structure

7.2 TAMI System Implementation

Figure 7.3 shows the system structure of our TAMI prototype implementation. To distinguish this implementation with the general architecture discussed in the previous section, we use the term "*TAMI system*" to refer to our TAMI prototype implementation, while using "*TAMI*" to refer to the general architecture. As explained above for the Public-Service scenario, the TAMI system needs to be deployed on three different types of network nodes, which we will refer to as *client, master*, and *agent*. We now discuss the details of these system components, focusing on the features related to topology-aware measurement.

7.2.1 Client Interfaces

Client interfaces of the TAMI system allows clients to control their measurements through a set of configuration options. The four most important options control measurement techniques, measurement destinations, measurement agents, and measurement scheduling. Their values are listed in Table 7.1. In this table, all the parameters are self-explanatory except for the scheduling parameters, which will be discussed in Section 7.2.4.

The TAMI system provides three client interfaces: web interface, command-line interface, and programming interface. The web interface is provided mainly to help new users learn how to use the TAMI system without having to install any TAMI software. It is a CGI wrapper for the command-line interface, and it only accepts a subset of system

Table 7.1: Client Request Parameters

Measurement techniques	
PING	measures path delay using ping
TRACEROUTE	measures path route using traceroute
IGI/PTR	measures path available bandwidth using IGI/PTR
PATHNECK	locates path bottleneck using Pathneck
Measurement destinations	
(default)	when no destination is specified, client itself is the measurement destination
(explicit)	client can specify a destination list
Measurement agents	
(explicit)	explicitly specifies measurement agents
AUTO_AGENT	the master automatically selects a diverse set of agents to cover the source/sink tree of specified destinations. This is the default mode.
ALL_AGENT	uses all available agents
ANY_AGENT	uses an arbitrary agent to measure the specified destinations
Measurement scheduling	
PM_SYN	measurements from different agents to a common destination need to be serialized
PM_SPLIT	split all destinations "evenly" among agents to achieve the smallest measurement time
PM_RAN	different agents measure the same set of destinations in a random order

parameters to limit the load on the web server. The command-line interface consists of a set of executables that can be invoked from a command line, and provides access to all features provided by the TAMI system. The programming interface is a library that allows applications to use the TAMI system directly.

7.2.2 Measurement Module Implementation

There are two ways to integrate the measurement techniques into the TAMI system. One is to embed the implementation code in the TAMI system; this typically requires rewriting some of the codes. The other method is to directly invoke existing implementations through system-call interfaces such as `system()`. Using the second method, new tools can be easily added, but the existing implementations may not be easily controllable by the TAMI system. For example, the standard traceroute can not send probe packets faster than one packet per second, and it can be very slow when a router does not respond. Also, existing implementations may not export fault signals, making it hard for the TAMI system to handle exceptions. As a result, we choose to use the first method in our prototype implementation.

The TAMI system currently implements four measurement techniques: ping, traceroute, Pathneck, and IGI/PTR. Among them, ping and traceroute are well understood, and their implementations are fairly simple. Pathneck [56] locates Internet path bandwidth bottlenecks; IGI/PTR [58] measures available bandwidth using two techniques, estimating background traffic (IGI) and packet transmission rate (PTR), respectively. Since Pathneck and IGI/PTR were designed and implemented by ourselves, it was easy to adopt their implementations into the TAMI system. An important point is that our implementation supports both user-level and kernel-level (using libpcap) timestamps. Kernel-level timestamps are needed to improve measurement accuracy on heavily loaded hosts such as some Planetlab nodes.

7.2.3 Topology-Aware Functionality

As described at the beginning of this chapter, TAMI topology-aware functionality is used to collect network topology information that is useful for certain applications. For example, for network connectivity monitoring, this can simply be the link-level topology of the whole network; for tomography techniques, this can be a set of end-to-end paths that share a common link. In our TAMI prototype system, since we focus on supporting the BRoute system, topology information refers to the source and the sink trees as defined in Chapter 5.

In the TAMI system, the tree operations are implemented in both the master and the client. The master focuses on tree-construction operations, including (1) selecting a set of agents that are diversely distributed on the Internet to satisfy a tree measurement request; (2) IP-level source and sink tree measurements, which are obtained by combining traceroute results measured either by client or by the agents selected by the master; (3) AS-level source and sink tree data, which can be inferred using IP-level tree data. The client focuses on the operations that *use* the trees, for example, measuring tree-branch available bandwidth using Pathneck.

7.2.4 Measurement Scheduling Algorithms

The TAMI system implements three different scheduling algorithms: PM-SYN, PM-SPLIT, and PM-RAN.

- PM-SYN serializes measurements towards a common destination. This, for example, supports source and sink tree measurements.

- PM-SPLIT is used when a large number of destinations need to be measured exactly once, but which agent measures which destination is not important. This scheduling algorithm makes full use of all the available agents to finish the measurements as quickly as possible. This can, for example, be useful when measuring the last-hop bandwidth for a set of nodes.

- PM-RAN is a “best-effort” scheduling algorithm where neither measurement order nor measurement completeness is important.

As will demonstrated in Section 7.4, PM-SYN can help applications to effectively avoid measurement interference. It can also parallelize measurements from multiple clients. Therefore, PM-SYN is the most important scheduling algorithm and the focus of the following analysis. While we do not use PM-SPLIT and PM-RAN in the applications discussed in Section 7.4, they are supported for two reasons. First, they are needed for the TAMI system administration and debugging. For example, PM-RAN is very useful for quickly checking the status of each agent. Second, they provide two candidates to compare with PM-SYN as an effort to better understand the scheduling overhead, and also to study the extensibility of the scheduling data structures.

Despite targeting different application scenarios, these scheduling algorithms all share the following three features: (1) efficient, i.e., finishing client requests in the shortest possible time; (2) fair, i.e., no performance bias among multiple measurement clients; and (3) fault-tolerant, i.e., they accommodate agent failures, including both software failures (e.g., agent software crashes) and hardware failures (e.g., physical agent node goes down). Below we describe these three algorithms in more detail, focusing on efficiency and fairness; fault-tolerance is discussed in Section 7.2.5.

PM-SYN

PM-SYN uses two data structures to implement its functionalities: an AVL tree and a 2D linked list (see Figure 7.4). The AVL tree is used to store all destination IP addresses that are currently being measured by some agent, so as to avoid measurement interference. That is, if a destination is already in the AVL tree, the corresponding measurement will not be scheduled. The 2D linked list maintains the working relationships between client requests and active agents. The crossing points ($c_i a_j$ in Figure 7.4) identify whether an agent should work on a client request. For example, crossing point $c_2 a_2$ means that client request c_2 needs agent a_2 to conduct measurement for it—we say agent a_2 is “linked” with client c_2. Each client request has a list (pointed by $client.ca$) of crossing points to identify all the agents that need to work for it; and each agent also has a list of crossing points (pointed by $agent.ca$) to identify its client requests. For each client request, we use as many linked agents as possible to conduct measurements in parallel so as to reduce overall measurement time.

To achieve client fairness, every agent iterates through all the clients it is linked with, using the structure $agent.w\_ca$. For example, after agent a_2 finishes one measurement for client c_2, it will work for client c_3, even if there are other destinations need to be measured for c_2. To guarantee measurement completeness, we keep track of the measurement

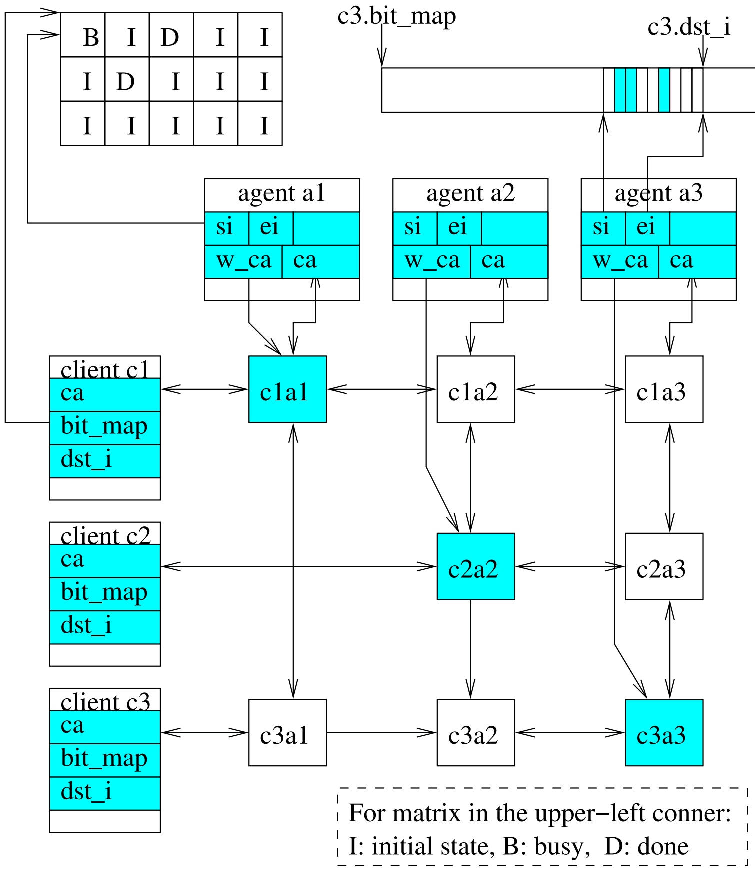


Figure 7.4: Data structure used for scheduling

progress of each client using a bit-map. For the PM-SYN algorithm, since one destination needs to be measured by multiple agents, its implementation uses a multi-line bit-map, where the rows correspond to agents and columns correspond to destinations. For example, the 3x5 matrix in the top-left corner of Figure 7.4 is the bit-map for client c_1. Using a bit-map allows us to measure any destination at any possible time, so as to achieve better parallelism.

PM-SPLIT

The PM-SPLIT algorithm has more relaxed scheduling constraints, and its goal is to finish a client request in the shortest time possible. Since each destination needs to be measured only once, a simple method is to split all destinations evenly among all active agents, send a sublist to each agent in a single message, and then wait for measurement results. This way, the master, unlike the PM-SYN case, does not have to submit a request message for each destination, thus avoiding the corresponding overhead. However, in this method, slow agents can become a performance bottleneck. For this reason, we limit the scheduling granularity as $max(10, client.dst\_num/agent\_num)$, where $client.dst\_num$ is the total number of destinations need to measure in the request, while "10" is an empirically selected upperbound on the number of destinations the master can forward to an agent in a single message. This approach reduces the overhead while still achieving a high level of parallelism and also being resilient to slow agents.

The PM-SPLIT implementation uses the same 2D linked list data structure as PM-SYN, except that its bit-map is a single line since each destination only needs to be measured once. We use $client.dst\_i$ to point to the first destination that has not been measured. By carefully manipulating the bit order, we can ensure all destinations before $client.dst\_i$ are in non-idle status, so that an idle agent only needs to start from $client.dst\_i$ to search for the next set of destinations to measure, and each agent only needs to remember the starting point ($a.si$) and ending point ($a.ei$) in the bit-map.

PM-RAN

PM-RAN scheduling is managed by the agents. That is, measurement requests in PM-RAN mode are directly forwarded to agents. Each agent maintains a queue of the measurement requests received from the master, and the tasks are ordered by their submission times. With these queues, agents can conduct measurement independently without waiting for instructions from the master. In Section 7.3.4, we will see that PM-RAN incurs the smallest management overhead.

7.2.5 Agent Management

In distributed systems, software and hardware failures are a common phenomenon. For example, on PlanetLab, the MTTF of links has been shown to be 9.48 hours, and the MTTR from these failures is 2.69 hours [51]. Here, we are mostly concerned about agent failures. TAMI agent management module detects agent failures using socket-level disconnection signals and by using a keep-alive protocol.

Socket Signals

All communication in the TAMI system is based on TCP. As a result, software crashes on one end will generate socket-level disconnect signals such as `SIGPIPE` at the other end of the connection. When receiving such a signal, the TAMI system software cleans up the corresponding data structures, updates system status, and then either exits (for the client) or remains active (for the master and the agents). For agents, this means that they enter an idle state and periodically try to connect to the master (the IP address of the master is fixed). When the master comes back, all agents will automatically join the system, thus avoiding having to restart each agent, which could be time consuming. When an agent working on a PM-SPLIT request crashes, we need to move the un-measured destination elements into the later part of the bit-map, i.e., behind $client.dst\_i$ (see Figure 7.4). This is necessary to ensure measurement completeness.

Since there could be a large number of agents in the system, we need a mechanism to automatically restart them. In the TAMI system, we use a cron job to periodically check whether the agent software is still running, restart it if it is not. Since the cron job is started by the OS at boot time, this automatically deals with the case of agent crash due to OS rebooting.

Keep-Alive Protocol

When disconnection is due to network problems, as explained in Section 5.12-5.16 of [112], it is possible that no error signal is generated on either endpoint of the connection. We address this issue using an application-level keep-alive protocol between the master and agents. The master maintains a timer for each active agent, and every agent also keeps a timer for the master. The master and agents periodically exchange keep-alive messages to refresh these timers. An expired timer is treated as a disconnection. That is, we assume the corresponding peer has crashed when a timer expired, and will carry out the same operations used to deal with socket signals.

7.2.6 Other System Features

We briefly discuss several other important features of the TAMI system—security, periodic measurement support, and caching.

Security

Given that the TAMI system controls a large number of agents, a key security concern is that the agents can be exploited to launch DDoS attacks. The TAMI system tries to address this problem in two ways. First, we expose the agents as little as possible. Clients are only allowed to connect to the master, thus there is only one service point that needs to be protected. When communication between agents and clients is inevitable, for example during IGI/PTR measurement, we always let agents initiate the connection, so that agents do not need to open any listening port for outsiders. Second, the scheduling algorithm implemented in the TAMI system ensures that there is at most one agent measuring a destination at any given time. This method, however, does not address the case where the attack target does not correspond to the destination IP addresses. As Spring et.al. [110] pointed out, with a sophisticated understanding of network topology and routing, distributed systems like the TAMI system can still be used to generate heavy traffic load on some network node or link. To deal with this issue, we turn to mechanisms like limiting the measurement frequency of clients and user-behavior logging.

Periodic Measurement Support

An important function of the TAMI system is to support network monitoring and diagnosis, which often require repeatedly measuring the same metric. The TAMI system supports this directly: a client does not need to repeatedly submit measurement request. It can just submit a request once and include a measurement frequency. The TAMI system automatically launches the measurement at the corresponding time, and sends back measurement results. This feature not only makes measurement management easier, it also makes it possible to continue the measurement if end users lose network connectivity. Obtaining network monitoring results is especially important during those periods for diagnostic purposes.

Caching

In some application scenarios like Planetlab projects, many end users may want to measure the same metric at the same or similar times. To avoid redundant measurements and also to reduce response time, the TAMI system uses MYSQL to store all the measurement results. If a client request specifies an acceptable cache time, the TAMI system will use cached measurement results for that time period, if there are any, to satisfy the request.

7.3 Performance Evaluation

The TAMI system achieves good user-perceived performance through fast measurement setup and measurement scheduling. To set up measurements using the TAMI system, a client only needs to submit its measurement request using a TCP connection, instead of using the time-consuming ssh command line. For example, we randomly selected 46 Planetlab nodes to issue an "`ssh -l <user> <host> ls`" command, the median response time is 3.3 seconds, which is much larger than the time needed by the TAMI system. In this section, we study the TAMI system scheduling performance.

The performance metric that we use in this section is *response time (t_{resp})*, which is defined as the average time used by the TAMI system on one measurement. For example, if a request needs to measure N_d destinations from all N_a agents, i.e., a total of $(N_d * N_a)$ measurements, and the TAMI system uses T time to finish the request, then $t_{resp} = T/(N_d * N_a)$. Note that unlike the traditional definition, our definition is based on individual measurement, not a measurement request. This is mainly for the convenience of comparing with the real *measurement time*, which is the time used by an agent for an individual measurement. The measurement times for the tools are fairly well understood since they are typically studied when the tool is proposed, e.g. [58] and [56] for IGI/PTR and Pathneck respectively. In this section, we will evaluate various aspects of TAMI's response time. Note that for tree-related applications, we mainly care about TAMI's performance on large-scale and measurements of bandwidth and bottleneck properties.

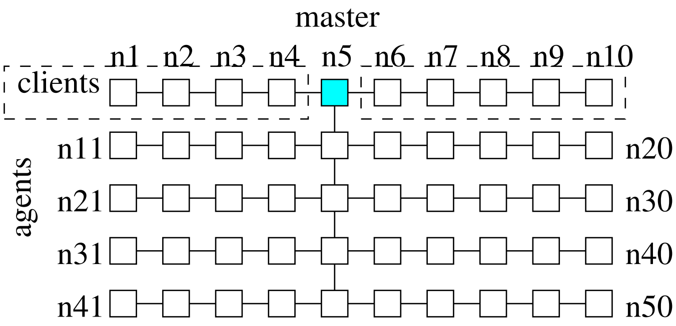


Figure 7.5: Emulab testbed topology

For repeatability, we use Emulab [5] for the experiments in this section. Performance results for the TAMI system running on the Internet are presented in Section 7.4.2. Using Emulab, we first conduct a typical run using the PM-SYN algorithm to obtain some basic measurement times and response times for the TAMI system (Section 7.3.2). We then look at the relationship between the number of agents and the response time (Section 7.3.3). This is followed by a performance comparison of the scheduling algorithms (Section 7.3.4). Finally, we look at the fairness of the TAMI system when serving multiple clients (Section 7.3.5). For all the experiments in this section, we use user-level timestamps unless stated otherwise. We first describe the testbed configuration.

7.3.1 Emulab Testbed Setup

Figure 7.5 shows the topology of the Emulab testbed used in this section. All links have 50Mbps capacity and 5ms delay. The 50 nodes are used as follows: node n_5 runs as the master, $n_1 - n_4$ and $n_6 - n_{10}$ are clients, and the other 40 nodes are agents. The overall configuration should be viewed as an example of a reasonable large set of nodes with Internet-like delays. We do not claim this topology is representative of the Internet as a whole, but we do believe it is sufficient to provide useful insights into how the TAMI system properties affect response times.

7.3.2 Performance of PM-SYN

In this section, we use the PM-SYN algorithm to obtain some baseline performance results: the actual measurement time of each measurement technique, the agent utilization, and the response time of the TAMI system when all agents are utilized. For single-end probing techniques, i.e., ping, traceroute, and Pathneck, we submit one PM-SYN measurement request that uses all 40 agents to measure a 50-IP destination list, which corresponds to all the nodes in the testbed. For the two-end control technique (IGI/PTR), we use n_1 to submit a PM-SYN measurement request that uses all 40 agents to measure

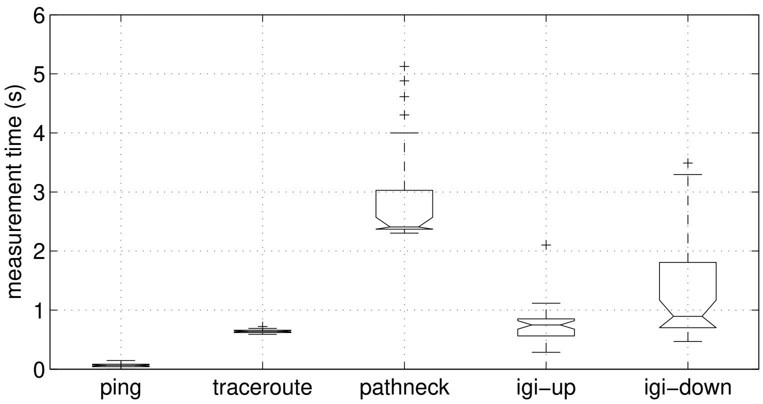


Figure 7.6: Measurement time of each measurement technique in the TAMI system, using user-level timestamps

the available bandwidth for paths between the agents and the client. The upstream and downstream paths are measured separately; these are marked as "igi-up" and "igi-down", respectively. For every technique, we repeat the experiment using both user-level timestamps and kernel-level timestamps.

Figure 7.6 plots[1] the medians and variances for the actual measurement times of each measurement technique; the results presented use user-level timestamps but the results for kernel-level timestamps are similar. The median values are also listed in the "Real Time" column of Table 7.2. This data provides the performance benchmarks for each measurement technique. Note that the actual measurement times are closely related to the RTTs of the paths being measured, e.g. changes in the RTT directly affect the measurement times. Also, these results are for the case when all destinations respond to probing packets. If a destination does not reply to probing packets, TAMI agents will wait until a timeout occurs, which will also affect the results.

Table 7.2 presents three different views of the PM-SYN performance results. The "Real Time" column lists the median of actual measurement times, which are copied from Figure 7.6. The "Resp. Time" column lists the response times, while the "Speed-Up" column is the ratio of the previous two columns. Finally, the "Agent Idle" column is the median idle time intervals for all agents. We did not compute Agent-Idle time for igi-up and igi-down because each agent is only involved once in each measurement. From this table, we can draw two conclusions. First, Agent-Idle times for ping are both around 0.33–0.35 seconds. We believe this reflects TAMI-system overhead because the real measurement times of ping are much smaller the Agent-Idle times. Agent-Idle times for traceroute are

[1]The graphs were generated using the `boxplot` function of Matlab, where one bar corresponds to one measurement type. The middle boxes have three lines corresponding to the lower quartile, median, and upper quartile values, and the whiskers are lines extending from each end of the box to show the extent of the rest of the data.

Table 7.2: Performance of PM-SYN (unit: second)

	Real Time	Resp. Time	Speed-Up	Agent Idle
ping	0.063	0.014	4.5	0.354
ping.b	0.063	0.015	4.2	0.348
traceroute	0.640	0.034	18.8	0.338
traceroute.b	0.640	0.034	18.8	0.338
pathneck	2.411	0.130	18.5	0.985
pathneck.b	2.402	0.131	18.3	0.990
igi-up	0.750	1.015	0.7	—
igi-up.b	0.753	1.074	0.7	—
igi-down	0.901	1.610	0.6	—
igi-down.b	0.934	1.365	0.7	—

[.b means using kernel-level timestamps.]

also around 0.33 seconds, although their real measurements times are larger than those of ping. However, Pathneck has much larger Agent-Idle times. This is because its larger measurement time results in longer waiting times for destinations to free up. Comparing the Agent-Idle times for ping, traceroute and Pathneck, we can see that when real measurement time is small enough, Agent-Idle time is determined by the system overhead and is not sensitive to the real measurement time. Second, by parallelizing the measurements, the TAMI system significantly improves the response time. The speed-up is at least 4 (for ping), and can go as high as 19 (for traceroute). Note that none of the techniques achieve a 40 times of speed-up, as one might expect given that there are 40 agents running in parallel. This is mainly due to TAMI-system overhead as identified by the "Agent Idle" column. Not surprisingly, the smaller the real measurement time, the larger impact the system overhead has.

7.3.3 Impact of Multiple Agents

Next, we study how the number of agents affects the response time. In this experiment, we submit requests from only one client node, but the number of agents used to conduct measurements changes from 1 to 40. We use ping and Pathneck to do the experiment, due to their wildly different measurement times (see Table 7.2). The same experiment is repeated using both PM-SYN and PM-SPLIT. For each configuration, we repeat the experiment five times.

The experimental results are shown in Figure 7.7, where we show both response times and the corresponding benchmark values. The points labeled with "x" the median values of the five experiments, and the bars show the maximum and minimum values. The benchmark values (the unlabeled curve) are computed as $(t_{resp}\_with\_1\_agent/N_a)$—we expect to see the response time reducing proportionally as we increase the number of agents.

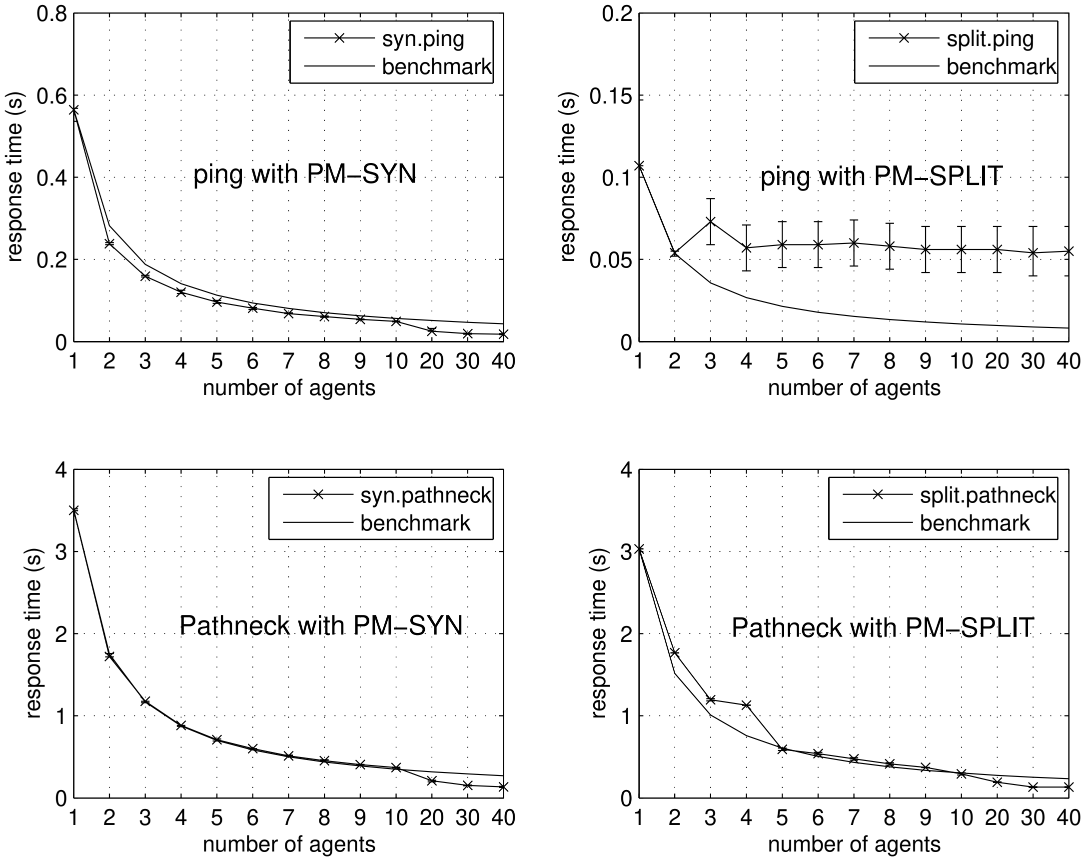


Figure 7.7: Comparing the TAMI system performance with different number of agents, under both PM-SYN and PM-SPLIT

The left two figures show the results using the PM-SYN scheduling algorithm. We can make several interesting observations. First, with only one agent, the response time of ping measurement under PM-SYN is 0.58 seconds, which is much larger than the real measurement time listed in Table 7.2. This is a result of the system overhead. The fact that it is larger than the 0.35 seconds listed in Table 7.2 ”Agent Idle” column tells us that the system overhead has a higher impact when there are fewer agents. Second, as the number of agents increases, the response times of both ping and Pathneck when using the PM-SYN algorithm decrease proportionally. This shows that both the system overhead (for ping) and the measurement time (for Pathneck) experienced by end users can be reduced proportionally as the number of agents increases. Third, when the number of available agents is over 10, the response times, for both ping and Pathneck, are significantly less than the benchmark values. This is because system concurrency can only be fully exploited when there is enough parallelism. For example, with a large number of agents working simultaneously, downstream and upstream communications can happen simultaneously; the local

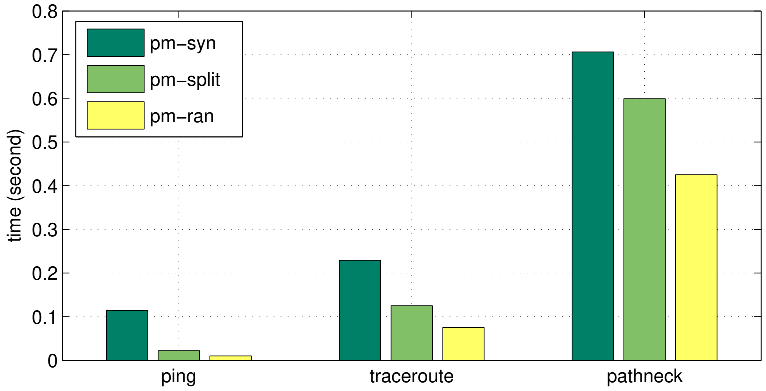


Figure 7.8: Comparing the TAMI system performance under PM-SYN, PM-SPLIT and PM-RAN

processing of master and agents and their communications can also execute in parallel. These results show that the TAMI system is very efficient for large-scale measurements, which is the target application domain for the TAMI system.

When using the PM-SPLIT algorithm (the right two graphs), as expected, the curve for Pathneck is very similar with that for the PM-SYN algorithm. The curve for ping, however, has a different shape—its response time stops improving after there are more than two agents. This is the combined effect of the small real measurement time of ping and the implementation of PM-SPLIT. In PM-SPLIT, we assign at most ten destinations to an agent at a time, and the agent sends back measurement results for each *single* destination. Due to the small measurement time for each destination, the master is kept busy by the measurement results sent back by the first two agents. The result is that, before the third agent is assigned any destination, the first two agents have already finished their work. Hence most of the time only two agents are working in parallel. Of course, this is not a problem if the measurement time is as large as in Pathneck. This problem is of course easy to fix, e.g. by batching more results.

7.3.4 Comparison of Different Scheduling Algorithms

This experiment is designed to compare the TAMI system performance with different scheduling algorithms. The experimental configuration is similar to the one used in Section 7.3.2. The difference is that we use only five agents so we can fully utilize the agents with the PM-SPLIT algorithm. Figure 7.8 plots the response times using different configurations. We see that PM-SYN always has the largest response time, while PM-RAN has the smallest. The reason is fairly intuitive: PM-SYN needs to avoid destination conflicts, which can force agents to wait for each other. PM-SPLIT does not have this problem since each destination in PM-SPLIT only needs to be measured once. Compared with

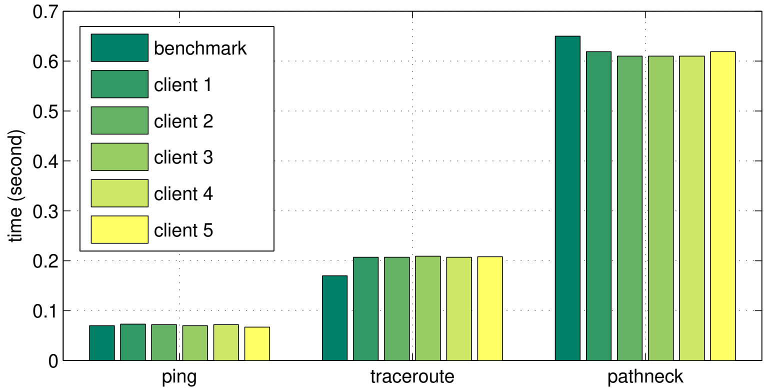


Figure 7.9: Fairness of the PM-SYN scheduling algorithm

PM-SPLIT, PM-RAN incurs minimal management overhead in the master, which further reduces the response time.

7.3.5 Fairness Among Clients

To study the fairness of the TAMI system, we repeat the same experiment that was conducted in Section 7.3.2, except that we have five different client nodes submitting the same measurement request.

Figure 7.9 plots the experimental results. The first bar in each group is the benchmark value, which corresponds to five times the response time measured in Section 7.3.2. It is presented for comparison purposes, since with five clients requesting service simultaneously, the response times are expected to increase by a factor of five. The other five bars are the response times measured from each individual client. For Pathneck, the performances of individual clients are apparently better than the benchmark values. This is because with more clients, the PM-SYN algorithm has a higher probability of finding a non-conflicting task for an idle agent, thus reducing agent idle time. For ping and traceroute, however, individual client's performance is very similar or slightly worse than the benchmark value. This is because the performance of these two types of techniques is limited by the TAMI system overhead instead of the measurement overhead, thus there is no free processing power that can be exploited by multiple clients. Overall, for all three measurement techniques, the performance of the five clients are very similar, which confirms that the TAMI system is indeed fair when serving multiple clients.

7.3.6 Summary

In this section, we have quantitatively analyzed TAMI system response time under various conditions, the actual measurement times of all implemented measurement techniques, and

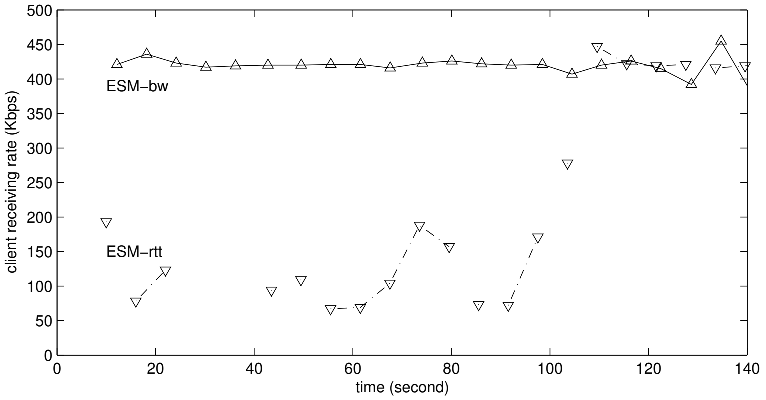


Figure 7.10: Join-performance difference from ESM-bw and ESM-rtt on a single client

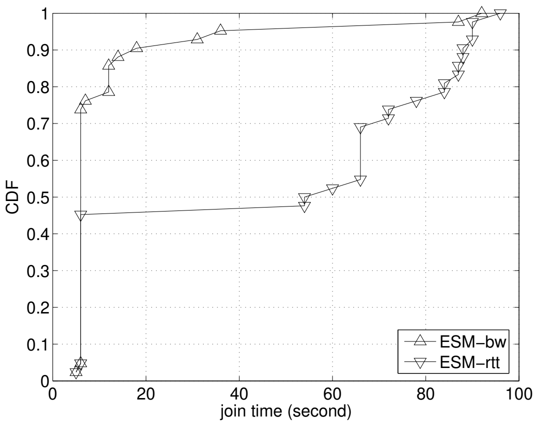
the system overhead of different scheduling algorithms. We showed that for the scenarios that we focused on, i.e., for large-scale measurements and bandwidth measurements, the TAMI system is effective in achieving small response time by fully utilizing measurement resources, and remains fair when serving multiple clients. Besides, we also learned that (1) system scheduling overhead is an important factor that affects system response time, especially when the real measurement time is small; and (2) system parallelism can be limited by the complicated interactions among measurement overhead, scheduling overhead, and scheduling communication patterns.

7.4 Applications

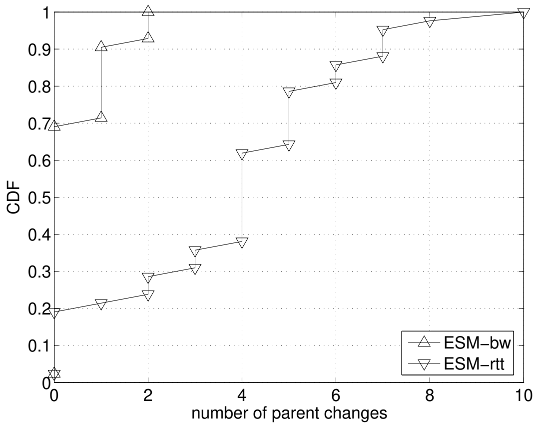
As a measurement infrastructure, the ultimate goal of the TAMI system is to support network applications by providing easy and real-time access to topology-aware measurement data. In this section, we describe three applications that can benefit from the TAMI system: bandwidth resource discovery for P2P applications, testbed performance monitoring, and route-event diagnosis. Since the results presented in this section were collected on Planetlab, we use kernel-level timestamps for the measurements.

7.4.1 Bandwidth Resource Discovery for P2P Applications

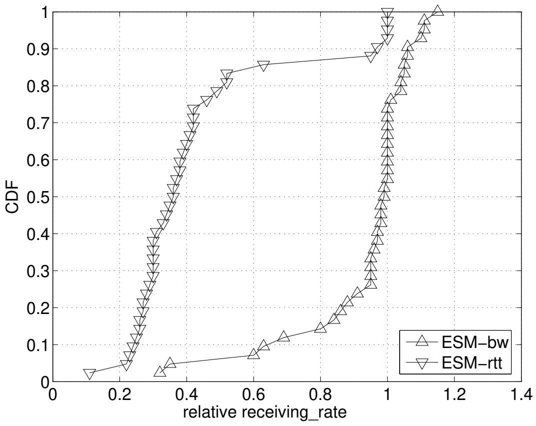
The BRoute system makes it possible for us to quickly infer end-to-end available bandwidth between two end nodes once we have the topology and the tree-branch bandwidth information of their source and sink trees, as provided by the TAMI system. That can help peer-to-peer applications to discover and adapt to bandwidth changes by optimizing overlay routing. In this section, we use ESM (End System Multicasting) [35] as an example to demonstrate this capability.



(a) join time



(b) number of parent changes



(c) relative receiving-rate

Figure 7.11: Performance difference from ESM-bw and ESM-rtt for all clients

ESM is a peer-to-peer multicast system that uses a self-managed data forwarding protocol to transmit data among peers. In ESM, when a new node joins, the system must choose a peer in the system that will function as its parent. The goal is to pick the peer with the highest available upstream bandwidth. This requires information about the available bandwidth between the new node and all existing peers. Unfortunately, obtaining this information is expensive due to the high overhead of available bandwidth measurement techniques. Therefore, ESM instead uses delay measurements to select the initial parent for a new node during the join procedure. Once a new node joins, it will keep probing other peers for both bandwidth and delay information and switch to a better parent if there is one. We will call this the "ESM-rtt" version of ESM.

The TAMI system makes it practical to use available bandwidth to select the first parent. With the programming interface provided by the TAMI system, we modified ESM such that, before starting the join procedure, a new node first obtains its source and sink tree structures, using ten TAMI-selected agents. It also obtains bandwidth information for the branches in the source and sink trees. In this way, when a new node joins, all current peers should already have information on source and sink trees and tree-branch available bandwidth, which can be sent to the new node. With the peers' source trees and its own sink tree, the new node can calculate the available bandwidth from other peers to itself, using the algorithms proposed in BRoute (see Chapter 6). The peer that has the largest available bandwidth is selected as the parent. This modified ESM is denoted as the "ESM-bw" version of ESM.

Using 45 Planetlab nodes, we did experiments using ESM-bw and ESM-rtt, respectively. In these experiments, we only focus on the join performance since only the join procedure of ESM uses the TAMI system. To emulate real world network connection speeds, we intentionally selected ten nodes that had less than 1Mbps available bandwidth. The multicast source was on one well provisioned node, sending out a data stream in a pre-configured 420Kbps rate. The other 44 nodes were clients and joined the system sequentially.

We use Figure 7.10 to illustrate how ESM-bw improves the performance for an individual client. The x-axis is the running time (ignoring the time used to measure source and sink trees), and the y-axis is the data stream receiving-rate at the client. The data points marked with "$\triangle$" are from ESM-bw, and those marked with "$\bigtriangledown$" are from ESM-rtt. Disconnected points correspond to parent changes, i.e., the new node switches to a new parent, hoping to improve its performance. We see that ESM-bw can immediately find a good parent, while ESM-rtt experienced several parent changes over the first 100 seconds before identifying a good parent.

Figure 7.11 quantitatively compares the performance of ESM-bw and ESM-rtt for all clients, using the cumulative distributions of three metrics: the join time (graph (a)), the number of parent changes during the join time (graph (b)), and the relative receiving-rate during the join time (graph (c)). Here the join time is defined as the time to reach 90% of source sending rate (420Kbps); and the relative receiving-rate is defined as the ratio between the average data stream receiving-rate of a client and the source sending-rate. We can see that 90% of ESM-bw clients have join times less than 18 seconds, while over 50% ESM-rtt clients have join times over 60 seconds. The smallest join time is 6 seconds, as we sampled the performance every 6 seconds. The longer join time for ESM-rtt clients is because they could not find a good parent quickly—over 60% of them experienced at least four parent changes (see graph (b)). Parent changes directly affect the relative receiving-rate during the join time, as shown in Figure 7.11(c), 70% of ESM-rtt clients have relative receiving-rates less than 0.5, while over 80% of ESM-bw clients can achive relative receiving-rates that are over 0.8. Overall, we conclude that the TAMI system can indeed significantly improve ESM clients' performance by providing bandwidth information.

Detailed analysis shows that the main reason why ESM-bw performs better than ESM-rtt in this experiment is the existence of a significant number of slow clients. The slow clients generally do not perform well as parents, and their delay is poorly correlated with bandwidth. Without these slow clients, ESM-rtt performs very well, as confirmed by our experiment on another set of Planetlab nodes. Also note that we exclude ESM-bw's tree measurement time (around 60 seconds) from the join time in the analysis. For this reason, the comparison may not be completely fair, although this measurement time can be hidden if tree measurement is run as a background service on client nodes.

From the architecture perspective, we have treated ESM peers as clients of our TAMI system. At first, since ESM is a peer-to-peer application, it may appear that the P2P architecture illustrated in Figure 7.2(a) is a better fit. This is actually not the case, because sink tree measurement needs a *diverse* set of agents that have significantly different views of Internet routing, and these agents should have enough upstream bandwidth. ESM peers may not have sufficient diversity and bandwidth to function as effective agents, especially during the startup phase of the system.

7.4.2 Testbed Performance Monitoring

The TAMI system is deployed on Planetlab, making it easy to monitor Planetlab's performance. Compared with other monitoring efforts, such as CoMon [3], Iperf bandwidth monitoring [14], and pairwise delay monitoring [13], a distinguishing characteristic of the TAMI system is that it significantly improves the available bandwidth monitoring capability on Planetlab by supporting the tree operations used by the BRoute system. As demonstrated in Chapter 6, for a case study using 160 Planetlab nodes, 80% of end-to-end available bandwidth estimates using BRoute had an error of less than 50%. The TAMI support also makes the BRoute system very efficient. To demonstrate this capability, we write a simple client-side code (around 30 lines) using the tree library interface provided by the TAMI system, and run it on each Planetlab node. That is, the Planetlab node is both an agent and a client for the TAMI system. This client-side code sends requests to the system to measure both its source tree and its sink tree. When it has the tree data, it sends them to a central storage node, where full mesh bandwidth data can be calculated. We ran this code on 190 Planetlab nodes, and it only took 652 seconds to finish.

Besides the above available bandwidth monitoring, we also conducted other types of monitoring tasks using the TAMI system and obtained some preliminary yet interesting results. For example, using TAMI's PM-SYN algorithm, full-mesh ping measurements become very easy—only one request message is enough. Figure 7.12 plots the performance of TAMI in measuring full-mesh end-to-end delay for 210 Planetlab nodes. In this figure, the x-axis shows the time used by TAMI, in log scale; the left y-axis, corresponding to the dot points, is the number of measurements finished in each one-second interval; and the right y-axis, corresponding to the curve, is the percentage of measurements that have been finished for the request. Although the total time used is 1855 seconds (around 31 minutes), corresponding to a response time of 42ms, 98% of measurements are finished within 424 seconds, the rest 1431 seconds are due to a few slow nodes. In a similar way, we also conducted a full-mesh traceroute measurements using TAMI, where the total measurement time is 7794 seconds, and 98% finished within 723 seconds.

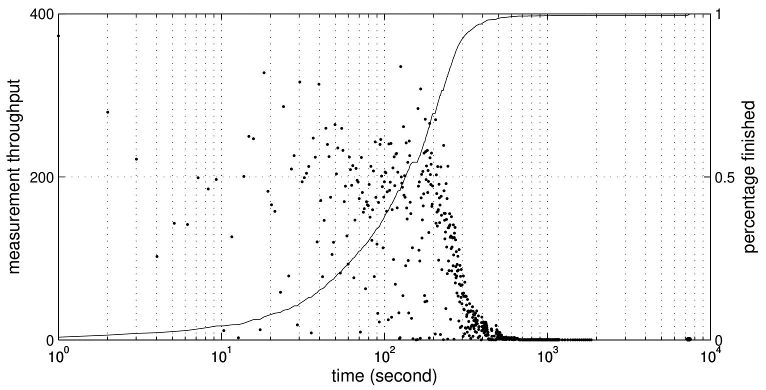


Figure 7.12: Performance of N^2 ping measurement on Planetlab

7.4.3 Route-Event Monitoring and Diagnosis

Route-event diagnosis is another important application that can be supported by the TAMI system. Route event refers to network anomalies that cause network connectivity problems. Typical examples include broken links, route loops, etc. As mentioned earlier, there have been some route-event diagnostic systems [48, 119]. A common property of these systems is that they use passive monitoring methods to identify route events. For example, the AS-level route-event diagnosis system presented in [48] uses BGP update messages, and the Planetseer system [119] monitors the TCP performance of a web-proxy's clients to identify possible route problems on the paths between the web proxy and the clients. Based on the TAMI system, we propose to use active methods to monitor IP-level route performance. This method uses IP-level source and sink trees to provide a complete view of end-node routes, and then focuses on monitoring the trees to detect route events. This way, we monitor all routes that may be used by the end nodes, not just those currently used for data transmission.

The implementation of our system has three characteristics. First, we use IP-level source and sink trees to monitor end-users' routes. For overhead consideration, we only focus on the first few layers of the IP-level trees. This method is not only efficient in reducing monitoring load, it also tends to be effective because route events often occur in small ISPs which are close to end users (i.e., we assume small ISPs have more limited management resource or capability to maintain their networks). Second, we focus on two types of route events—*deterministic* and *partial* route events. Deterministic events refer to those route events that are persistent during a monitoring interval. The rest are partial events. For example, a broken link is a typical example of deterministic route events, while route loop can be either deterministic or partial. Packet loss can also cause connectivity problem which may appear as partial route event. Therefore, we also consider packet loss as a route event in our diagnostic system. Third, we use two measurement tools for route-event diagnosis—ping and traceroute. Ping uses much fewer probing packets, while traceroute can obtain information for each hop.

Below we first present an empirical study on the number of route events that can be measured in networks. We then describe the diagnosis details for a route event. We finally explain the design and implementation details of the route-event diagnostic system.

Observations of The Two Types of Route Events

To understand how frequently deterministic and partial route events occur in networks, we collect IP-level sink-tree measurements for 135 Planetlab nodes using the TAMI system. We found that only 23 nodes did not experience any connectivity problem. Among the other nodes, 29 nodes experienced at least one packet-loss events, 8 nodes experienced only deterministic events, 38 more observed both deterministic and partially events, and 75 nodes experienced only partial events. These results show that route events are not rare on today's network.

The root causes of these route events are generally hard to identify. Our diagnostic system can only provide possible explanations or auxiliary information to simplify the diagnostic operations of human operators. Sometimes, a route event detected this way is not a real event, but a measurement artifact. An example is the incomplete traceroute due to routers that are pre-configured not to respond to ICMP probing packets. Since these routers do forward real data packets, they do not cause route events for real user traffic. Another example is routing policies that can generate "fake" route events. In our measurements, we found that the IP-level sink tree of node `pli2-pa-2.hpl.hp.com` has a router where 16 other nodes' traceroutes toward that node stop at that router, while other 117 nodes can pass it successfully and reach that node. We were later told that that this node has a routing policy that only allows nodes with Internet2 access to reach it. Obviously, to figure out such a policy, one would need domain knowledge to develop highly customized algorithms. For example, in this example, we would need ways to differentiate Internet2 nodes with the others. Next, we describe a detailed case study on how we used IP-level sink-tree information provided by the TAMI system to diagnose a route event.

Detailed Case Study of A Route Event

Around 00:30am EDT on 08/10/2005, we found that www.emulab.net was inaccessible from one of our home machines but it was accessible from university office machines. The home machine uses a commercial DSL service for Internet access. To figure out the problem, we used the TAMI system to launch a sink-tree measurement for www.emulab.net (with IP address 155.98.32.70) from all the 200 Planetlab nodes where the TAMI system was deployed at that moment. This request took 4.5 minutes (273 seconds) to finish. Based on the route data, we plotted the AS-level sink tree for the destination in Figure 7.13. In this figure, the labels on the links have the form "a/b", where b is the total number of

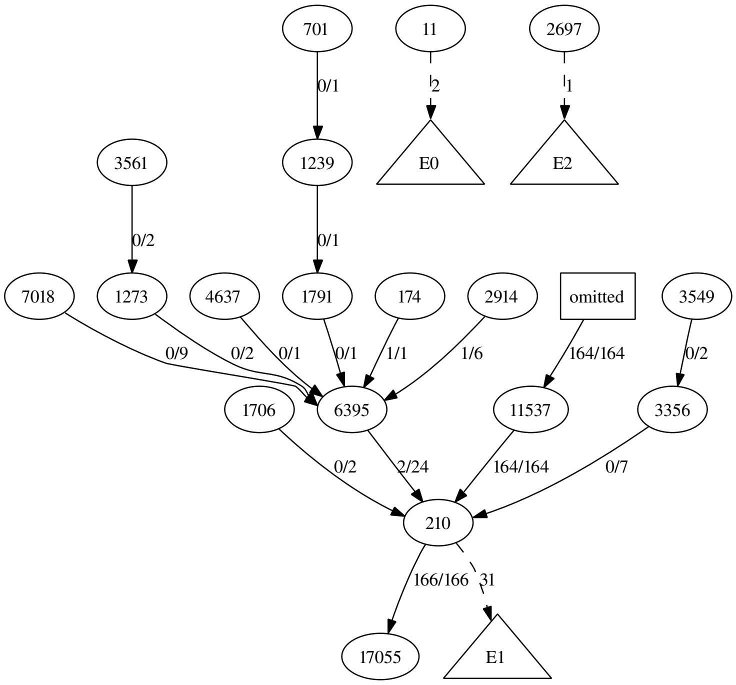


Figure 7.13: AS-level sink tree of www.emulab.net at 00:24am, 08/10/2005

traceroutes that pass that link, and a is the number of traceroutes that can reach the destination. Note that, due to the definition of AS-level trees (its technical definition can be found in Section 4.1 of [60]), the total number of ingress traceroutes into an AS can be less than the egress traceroutes from that AS. An example is AS6395 in Figure 7.13, which has four more egress traceroutes than ingress traceroutes, because the AS tree branch for these four traceroutes start from AS6395.

From Figure 7.13, we see that only AS11537 (Abilene Network) had good reachability to the destination, while measurement packets from other ASes mostly disappeared after entering AS210. That implies something was wrong with AS210's routing, specifically with paths from non-Internet2 networks. We also looked at the IP-level sink tree and found that there was a routing-loop between two routers within AS210. At that moment, we thought AS210 was doing network maintenance and the problem was going to be temporary.

At 11:59am EDT the same day, we found that the problem still persisted. We conducted another sink-tree measurement using the TAMI system (with 153 Planetlab agents), and found that although the sink tree is slightly different, the problem remained the same (see Figure 7.14). For the 22 traceroutes entering AS210 from non-Internet2 ASes, 21 of them did not reach the destination. After this measurement, we realized something was wrong in AS210, and confirmed with Emulab staff that AS210 had routing problems with

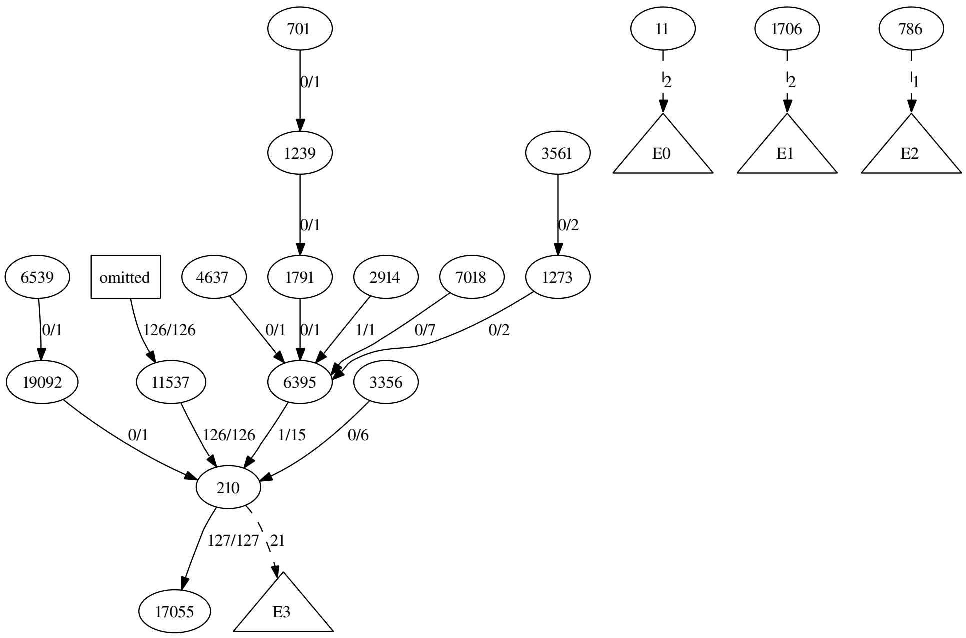


Figure 7.14: AS-level sink tree of www.emulab.net at 11:59am, 08/10/2005

commercial ISPs.

This is an excellent example of detecting a partial route event. It clearly demonstrates the benefit of IP-level sink-tree measurements in route-event diagnosis—with measurements from only one or a few agents, it is often not possible to pinpoint the problem.

An Example Route-Event Diagnosis System

We now describe a route monitoring and diagnostic system. This system is designed as a client software of the TAMI system. That is, it uses the TAMI system to measure IP-level sink trees but maintains all information on the client side. Figure 7.15 shows how the system works. Roughly speaking, this system includes three components: bootstrap, monitoring, and diagnosis. The bootstrap component initializes the system by obtaining benchmark route information for each end node in the system; the monitoring component periodically updates the benchmark information; while the diagnosis component launches measurements for specific route changes to find out the possible causes of the events. Below we describe each component in more details.

The bootstrap component has three responsibilities: (1) initializing the system by providing a complete set of route information from each end node, (2) periodically updating these route information, and (3) measuring the sink trees for end nodes. The complete set of route information is used for constructing IP-level sink trees, based on which monitoring can be done more efficiently by only measuring each tree branch just once. Since route

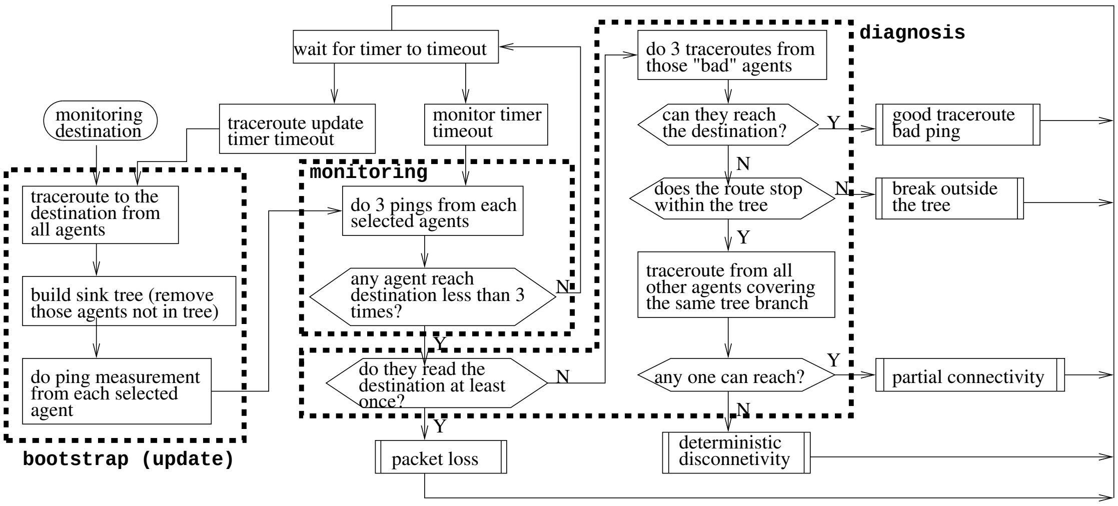


Figure 7.15: Node connectivity diagnostic framework

information can become stale, these trees need to be updated periodically by re-measuring all the routes. The update period depends on application requirements. Our system uses a five-hour update period, which is sufficient for experimental purpose.

When measuring IP-level sink trees, we also need to specify the depth of the tree. This is an important parameter that not only affects tree sizes, but also determines the scope of route problems the system can monitor and diagnose. Our current system builds four-level trees, since they are the most critical part of the network for applications like the BRoute system. This parameter can of course be customized for different applications. The traceroute data used for IP-level sink-tree constructions are classified into three categories according to the type of information they can provide for tree branches. The first category includes those complete routes that can reach the destination. Route data in this category are directly used for sink-tree constructions. The second category includes a particular type of incomplete routes whose last hop is within the sink-tree that is already constructed using the first category of route data. Route data in this category can still be used if they introduce a new sink-tree branch. The third category includes the incomplete routes whose last hop is not in the sink tree. The third category is not useful for our monitoring purpose since they can't observe any events in the tree. Note that the classification can change when routes change, and they need to be updated during each update period. For all routes in the first and second categories, the system further conducts ping measurements towards the destination, in preparation of the monitoring operations.

The second component of the system is the monitoring component. To reduce overhead, monitoring measurement for each destination are done using ping—each selected agent does three consecutive ping measurements towards the destination. If all ping measurements reach the destination, we claim there is no route events. Otherwise, the diagnostic component—the third component in this system—is invoked. The diagnostic component works as follows.

1. It first checks if there is at least one ping packet reaches the destination. If yes, we claim it is only a *loss event*, and finish the diagnostic procedure.

2. Otherwise, if none of the three ping packets reach the destination, it could be a potential disconnectivity problem. We then launch three traceroute measurements toward the destination to obtain more detailed route information about the path. If at least one traceroute successfully reaches the destination, we report a *good-traceroute-bad-ping event*. In our experiments, we saw such events but we are unable to explain the real root cause.

3. If none of the three traceroutes can reach the destination, we have a high confidence in claiming a diconnectivity event. We then check the connectivity from other agents to see the scope of the disconnectivity event. This is where the tree information is used. The simpler case is when traceroute measurements stop outside the tree, we can not further diagnose the event since it is outside the scope of our diagnostic capability. In this case, we report a *break-outside-tree event*.

4. If the traceroute measurement stops within the tree, we will know which branch the event is on. We can then use other agents that also traverse that branch to double-check the event. If all other agents can not reach the destination, either, we then report a *deterministic dicconnectivity event*; otherwise, we report a *partial disconnectivity event*.

Note using the traceroute results in this diagnostic procedure, we can also diagnose the *route-loop event* as demonstrated in the previous subsection. Although not illustrated in Figure 7.15, our system will also report this type of event.

We ran the system for 42 hours using 136 Planetlab nodes as destinations. During this period, 88 nodes experienced at least one route event. Among the 88 nodes, 31 nodes experienced only one type of events, 35 experienced two types of event, 16, 5, and 1 nodes experienced three, four, and five types events respectively. Packet loss seems to be the most popular type of events—78 nodes experienced such event; the next type is break-outside-tree type of events, with 58 nodes experiencing it; the third is good-traceroute-bad-ping, with 22 nodes experiencing it; while we saw no route-loop event. Also, when a destination experienced multiple same-type events, these events are often due to a same problematic router. For example, 41 nodes saw at least four packet-loss events, among them, 21 nodes' packet-loss events were due to problems on a same router. We also looked at the events from the source trees of these 136 nodes. Except there were only 46 nodes experiencing route events, the other results are similar with those from the sink trees. For example, packet loss is still the dominant popular event—41 of the 46 nodes experienced such events.

7.5 Related Work

Based on their application scenarios, existing network measurement infrastructures can be separated into two groups. One group includes those developed by ISPs, like IPMON [49] and NetScope[47]; they rely on passive measurements and focus on network internals. The other group includes those designed by the networking research community, they mostly rely on end-system based active measurements and focus on end-to-end performance. Representative systems in the second group include NWS [118], Scriptroute [110], NIMI [94], and ANEMOS [25]. These systems are general infrastructures that offer functions such as automated deployment, privacy protection, and security. TAMI also belongs to the second category, and shares many of the features of the above systems. However, TAMI's focus is on supporting topology-aware measurements by collecting and maintaining topology information and scheduling capabilities.

We mentioned that TAMI can be used as a data sharing platform. This feature is also discussed in ATMEN [73], which proposed a communication and coordination architecture across multiple administrative entities to achieve various goals including reuse of measurement data. SIMR [27] also proposed an infrastructure to share measurement data collected by different research groups. Since it relies on external data sources, it needs to address issues like data format compatibility, privacy, and security, many of which TAMI does not have to consider.

An important design goal of TAMI is to facilitate network event diagnosis by simplifying complicated measurements. It potentially can be used to automate the diagnosis operations in systems like PlanetSeer [119], which correlates web proxy logs with Internet traceroute data to detect changes in end-to-end performance.

While most end-system based monitoring systems, including the TAMI system, rely heavily on active measurements, several groups are also exploring the use of passive measurements. One example is NETI [69], whose monitoring activities include collecting data from regular end users by monitoring traffic on their systems.

The TAMI system currently implements four probing techniques: ping, traceroute, IGI/PTR, and Pathneck. While ping is implemented mainly for evaluation purposes, the other three tools are useful because they help collect topology and bandwidth information of source and sink trees, which is our focus. Of course, the TAMI system can also integrate other measurement techniques such as those estimating path capacity [32, 45, 77], available bandwidth [32, 65, 101, 113], and path loss [102, 80, 107].

7.6 Summary

In this chapter, we described the design and implementation of a TAMI prototype system. We evaluated the performance the TAMI system on Emulab, and used three representative applications to demonstrate TAMI's functionality. We showed that TAMI's scheduling functionality significantly reduces measurement response time, and its topology-awareness greatly simplifies network monitoring and diagnosis. Using three applications, we demonstrated the benefit that the TAMI system can provide. Specifically, they exemplify applications that require a global view of network performance information (the diagnosis application), fast measurement speed (the diagnosis and the monitoring applications), a convenient application interface (ESM and the monitoring application), and the ability to obtain available bandwidth for large-scale systems (ESM and the monitoring application).

Chapter 8

Conclusion and Future Work

Available bandwidth is an important metric for network monitoring and diagnosis due to its role in quantifying data transmission performance. This dissertation studied how to estimate available bandwidth and locate bandwidth bottleneck links, and how to provide systematic support for these measurement operations. Specifically, we developed techniques to estimate end-to-end available bandwidth and locate bandwidth bottlenecks. We also studied the general properties of Internet bottlenecks. Insight into bottleneck properties helped us solve the problem of estimating available-bandwidth for large-scale network systems. We proposed a monitoring and diagnostic platform that supports all the above bandwidth measurement operations. In this chapter, we first provide a detailed description of each contribution of this dissertation, and then discuss future work that can build on the results of this dissertation.

8.1 Contributions

8.1.1 Adaptive Packet-Train Probing for Path Available Bandwidth

As a first step towards estimating end-to-end available bandwidth, we have designed a single-hop model to study the detailed interaction between probing packets and background traffic packets. We found that the arriving rate of the packet-train at the destination is a function of its sending rate at the source and background traffic load, and that the path available bandwidth corresponds to the smallest sending rate where the arriving rate equals the sending rate. That smallest sending rate is what we called the turning point. Based on this insight, we developed the IGI/PTR measurement tool, which adaptively adjusts a packet-train's sending rate to identify the turning point. We have shown that this tool has a similar measurement accuracy (over 70%) as other tools like Pathload, but has a much smaller measurement overhead (50-300KB) and measurement time (4-6 seconds). That makes IGI/PTR very attractive to network applications.

We also studied the various factors that affect the measurement accuracy of IGI/PTR.

We found that (1) probing packet size can not be too small, because small packets are subject to measurement noise; however, once they are reasonably large, say 500 bytes, measurement accuracy is no longer sensitive to packet size. (2) Multi-hop effects are an important factor that can introduce measurement error. Multi-hop effects can be classified into pre-tight-link effect and post-tight-link effect. We found that it is the post-tight-link effect that introduces most of the multi-hop measurement errors, because pre-tight-link effect can be mostly smoothed by the background traffic on the tight link, thus having very small impact on measurement accuracy. For the same reason, unevenly generated probing packet-trains do not significantly affect measurement accuracy, either.

Finally, to demonstrate the applicability of the idea of adaptive packet-train probing, we integrated the PTR algorithm into TCP, and designed a new TCP startup algorithm—TCP Paced Start (PaSt). We showed that this algorithm can significantly reduce TCP startup time and avoid most of packet losses during startup.

8.1.2 Locating Bottleneck Link Using Packet-Train Probing

Our technique for locating bottlenecks—Pathneck—is a direct extension of the packet-train probing technique used for PTR. By appending small but carefully-configured measurement packets at the head and the tail of the PTR packet train, we create the ability to associate bandwidth information with link location information. The two most intriguing properties of Pathneck is that it only needs single-end control and that it is extremely light-weight: it uses upto ten packet trains to finish the measurement, which is several orders of magnitude lower than other bottleneck locating tools. Using both Internet experiments and testbed emulations, we showed that the location accuracy from Pathneck is over 70%. The main reason that Pathneck makes mistakes is the existence of a link that has very similar available bandwidth with that of the real bottleneck link. We also evaluated the tightness of the path available-bandwidth bound measured from Pathneck. Experiments on RON showed that the bound is actually fairly tight. Therfore, for applications that do not require precise values of available bandwidth but have only single-end access, Pathneck is a good option.

8.1.3 Internet Bottleneck Properties

The single-end control and small overhead properties of the Pathneck tool make it possible for the first time to conduct an Internet-scale measurement study of bottleneck properties. Our measurement study used around 100 measurement sources from RON, Planetlab, and an Tier-1 ISP, and selected destinations based on global BGP tables to cover the whole Internet. We found that (1) over 86% of Internet bottlenecks are within four hops from end nodes, which validates the popular assumption that Internet bottlenecks are mostly on Internet edges. (2) Internet end-user access links are still dominated by slow links, with 40% of which slower than 2.2Mbps. (3) Internet bottleneck locations are not very persistent because of route changes and path load dynamics. Depending on whether we focus on link-level or AS-level bottlenecks, only 20% to 30% of bottlenecks are perfectly presistent. (4) Bottleneck links, link loss and link delay are all considered as signals of path congestion, but only bottleneck links and packet loss can be correlated (for 60% of cases), while the correlation probability between bottleneck links and high-queueing delay links is very low (only upto 14%).

8.1.4 Source and Sink Trees and The RSIM Metric

To address the problem of large-scale available bandwidth estimation, we proposed the source-tree and sink-tree structures, at both the IP-level and the AS-level, to capture the topology information of end nodes. Source and sink trees are composed by all the upstream and downstream routes that an end-node uses. We used extensive traceroute and BGP data to show that the tree structures are indeed very close to real trees, and if we only focus on the first four layers—where most bandwidth bottlenecks locate—the tree sizes are also limited: over 90% have only 10–20 different branches. This tree concept not only helps us develop a large-scale available bandwidth inference system, it is also very important in route-event diagnosis and multi-path routing.

Similar to the idea of source and sink trees, we also proposed the RSIM metric to quantify route similarity between an arbitrary pair of end nodes. RSIM can be measured using only a small number of random traceroutes, and it captures the similarity of both upstream and downstream routes. We show that RSIM can be easily used for path-edge inference.

8.1.5 Large-Scale Available Bandwidth Inference

Based on the insight that most Internet bottlenecks are on Internet edges, and equipped with the source and sink trees, we developed the BRoute system, which can infer the available bandwidth of all N^2 paths in a N-node system with only ($O(N)$) overhead. The key operations of the BRoute system are to label the tree branches with bandwidth information, and then infer which two branches (one from the source node's source tree, the other from the destination node's sink tree) are used for the real end-to-end path. The first operation is currently done using Pathneck, whose bottleneck information can help us decide if the bandwidth information should be used for the source tree or the sink tree. For the second operation, we developed an inference algorithm based on AS-level source and sink trees, where we first identify the common-AS where the end-to-end path has the highest probability to pass, we then map the common-AS to the IP-level tree branches. Planetlab experiments showed that this algorithm has over 85% of inference accuracy. Overall, available bandwidth inference accuracy is over 50% for 80% of the paths in our case study.

8.1.6 Topology-Aware and Measurement Scheduling

To lower the bar of using the techniques that we have developed, including IGI/PTR, Pathneck, and BRoute, we developed the TAMI system to systematically support these available bandwidth measurement operations. We identified two important features that are needed by such an infrastructure but are not supported by existing measurement systems—the topology-awareness and the measurement scheduling functionality. For BRoute, the topology-awareness is responsible for obtaining the tree information of end nodes, while the measurement scheduling functionality is important for avoiding packet-train probing interference. These two features can both be used for other applications. For example, the topology-awareness is critical for tomography, and the measurement scheduling functionality is important for systems to support multiple applications whose measurements may interfere with each other. We showed that the TAMI system can be used to improve the joining performance of End System Multicasting, large-scale available-bandwidth monitoring, and route-event diagnosis.

8.2 Future Work

At least four areas of research can leverage on the results in this dissertation: improving the current systems, available bandwidth measurement in different environments, supporting new applications, and next-generation network architecture design.

8.2.1 Improving The Current Systems

Quite a few features of the monitoring and diagnosis work presented in this dissertation can be improved. First, we can improve the BRoute available-bandwidth inference accuracy by using both IGI/PTR and Pathneck for tree-branch bandwidth estimation. That is, we can use IGI/PTR to obtain a more accurate estimation for path available bandwidth, while using Pathneck to tell if the path available bandwidth should be used for source-segment or sink-segment. Second, the evaluations of the BRoute system and the RSIM metric are still limited by the diversity of the end nodes that we have access to. When more vantage points are available, it is necessary to do larger-scale evaluations than those presented in this dissertation. Third, this dissertation only presents a diagnosis system for route events, which are easier than diagnosing available-bandwidth related events. Part of the reason is that we have very limited experience in network operations and very limited network internal information to study available-bandwidth related events. With more experience and information, we expect to be able to identify more events, for example, network path congestion. Although Pathneck is developed towards this goal and we have shown that Internet bottlenecks have a close relationship with packet loss, we were not able to make claims in terms of diagnosis because we did not have network internal information to confirm the congestion events that we identified. When such information is available, it will be very interesting and important to study how well congestion events can be identified. Fourth, some bottleneck properties need to be explored deeper. For example, the analysis on the relationship between bottleneck link and link delay is still premature, partly due to the difficulty of obtaining a good estimation of link delay. It will be interesting to explore other data sets or experimental methods to get a better understanding of this problem.

8.2.2 Available Bandwidth Measurement in Different Environments

The discussion of this dissertation has three important assumptions about the measurement environment: (1) we only consider wired network, where link capacity is constant, and the dominant factor affecting available bandwidth is background traffic; (2) network routers use FIFO packet scheduling algorithm, where all packets goes through a same queue in the order of their entering time; and (3) we have dedicated measurement end hosts, which have low CPU load, and the packet timestamp measurement granularity is only determined by OS and is predictable. In practice, it is not rare that some of these assumptions can not be satisfied, and we need more research work to understand how to accurately measure available bandwidth in those environments.

For example, wireless networks directly break assumption (1) and often do not follow assumption (2). On wireless networks, link capacity can be dynamically adjusted, and both environment noise and access-point distance can degrade data transmission rate. Furthermore, wireless networks often use either DCF (Distributed Coordination Function) or PCF (Point Coordination Function) protocols for packet scheduling, neither of which is FIFO. That means we probably need a completely new set of techniques to address available bandwidth estimation problem on wireless networks, starting from the very definition of “available bandwidth”.

Assumption (3) can also be violated if a measurement application is not the only code running on an end host. A typical example is the Planetlab nodes, which are shared by a large number of projects. Under this condition, the packet timestamps measured by an application can have a big difference with the real sending or receiving times on the network interface cards. That is one of the major barriers for using existing available bandwidth measurement techniques on Planetlab. A possible solution is to use some filtering method to identify and discard those measurements using incorrect timestamps.

8.2.3 New Applications of Available Bandwidth

In this dissertation, we have explored several types of applications that require available bandwidth information. These include TCP startup, End System Multicasting, overlay routing, and multihoming. There are certainly more applications worth studying. Two important emerging applications are VoIP and IPTV. VoIP is often designed in a way that voice traffic share the same link with other IP traffic (e.g., web traffic). It is preferable to maximize the utilization of access-link capacity so that voice traffic only uses the bandwidth it is supposed to, which is often fairly small (e.g., 64Kbps). Achieving that requires an accurate information of the access link capacity so that we can know exactly the amount of bandwidth that can be used for best-effort IP traffic. That is, bandwidth information is critical for the packet scheduling in VoIP devices. For IPTV, each channel generally occupies several Mbps of bandwidth, to support multi-channel IPTV, available bandwidth is again very important for various scheduling and monitoring operations.

8.2.4 Implications for Next-Generation Network Architecture

Our research experience has shown that it is hard to monitor the Internet, and a fundamental reason is that the current Internet architecture does not have a good set of performance monitoring capabilities. That is, today's networks do not keep track of some types of key information, or do not provide information in the best way. For example, router queueing delay is a fundamental network performance metric. However, even with a decade of research and engineering effort, there is no good way to obtain router queueing delay information on today's Internet. Technically, this is not a hard problem—a router simply needs to time special packets when they enter and leave a router. With this capability, many heavy-weight active measurement techniques can be implemented using light-weight ping-like measurements. Another example is the timestamps returned by routers when they are queried by `ICMP_TIMESTAMP` packets. Although the ICMP protocol defines three different timestamps, router vendors only use one of those timestamps. That forces people to spend time and energy to design techniques like cing [28] to measure metrics like ICMP generation delays. Also these timestamps are in millisecond, which is not precise enough for applications that need micro-second information. The third example is the router IP addresses that are obtained from traceroute measurements. Such IP address is often an interface IP address instead of the loopback address of the router. A consequence of this implementation is that traceroutes from different end users result in different IP addresses, referred as IP aliases, for the same router. That makes it very hard to correlate route data measured from different vantage points. Although techniques like Ally [109] have been developed to solve this issue, they are subject to practical limitations and do not always work.

Note that implementing these features is not a hard problem. The key challenge is providing such functionalities while respecting other architectural requirements such as scalability and security. The reason is that these features not only give end users more methods to obtain network internal information, but they also provide malicious users more power to abuse network systems. Therefore, we must consider these features not just from a performance perspective, but from a higher architectural level.

In this context, interesting future work is to (1) identify the fundamental performance metrics that are needed for next-generation networks and applications, and (2) design and develop techniques to measure these metrics, and explore the possible mechanisms and protocols that are needed to maintain, update, and distribute this information in a secure and efficient way. During this process, one will want to look at other new network architecture research proposals, such as the 100x100 project [19], the FIND (Future Internet Network Design) initiative [20], and the GENI (Global Environment for Networking Investigations) initiative [21], to understand how the design fits in the larger picture. This effort hopefully can identify the general principles that can be used to guide the design and development of performance monitoring and diagnostic components for the next-generation network architectures.

8.3 Closing Remarks

Available bandwidth measurement has been shown to be a solvable problem. However related techniques have not been widely used or integrated into popular applications. One reason is that Internet bottlenecks are still on the edge, and link capacity seems to be already a good metric to get a rough idea about the performance of data transmissions. At the same time, link capacity information is often pre-known and does not require measurement. The second reason is that, as our measurements show, current Internet end-user access links are still dominated by slow links, which have very limited dynamics in terms of available bandwidth change. Both factors contribute to the impression that available bandwidth is not important. However, I believe this situation will soon be different, with the deployment of high-bandwidth applications like IPTV. I am looking forward to see more and more network applications to integrate available bandwidth measurement techniques to obtain network performance information and to improve their adaptability. For the same reason, I believe the techniques presented in this dissertation on available-bandwidth monitoring and diagnosis have a bright future.

Bibliography

[1] Abilene network monitoring. http://www.abilene.iu.edu/noc.html.

[2] AS hierarchy data set. http://www.cs.berkeley.edu/~sagarwal/research/BGP-hierarchy/.

[3] CoMon. http://comon.cs.princeton.edu/.

[4] Dummynet. http://info.iet.unipi.it/~luigi/ip_dummynet/.

[5] Emulab. http://www.emulab.net.

[6] IGI/PTR Homepage. http://www.cs.cmu.edu/~hnn/igi.

[7] IP to AS number mapping data set. http://www.research.att.com/~jiawang/as_traceroute.

[8] Iperf. http://dast.nlanr.net/Projects/Iperf/.

[9] libpcap. ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[10] ns2. http://www.isi.edu/nsnam/ns.

[11] Pathneck Homepage. http://www.cs.cmu.edu/~hnn/pathneck.

[12] Planetlab. http://www.planet-lab.org.

[13] Planetlab All Pairs Pings. http://pdos.csail.mit.edu/~strib/pl_app/.

[14] Planetlab Iperf. http://jabber.services.planet-lab.org/php/iperf.

[15] Resilient Overlay Networks. http://nms.lcs.mit.edu/ron/.

[16] RIPE RIS (Routing Information Service) Raw Data. http://www.ripe.net/projects/ris/rawdata.html.

[17] Rocketfuel Data Sets. http://www.cs.washington.edu/research/networking/rocketfuel.

[18] Route Server Wiki. http://www.bgp4.net/cgi-bin/bgp4wiki.cgi?Route_Server_Wiki.

[19] The 100x100 Project. http://100x100network.org/.

[20] The FIND Initiative. http://find.isi.edu.

[21] The GENI Initiative. http://www.nsf.gov/cise/geni/.

[22] Tulip Manual. www.cs.washington.edu/research/networking/tulip/bits/tulip-man.html.

[23] University of Oregon Route Views Project. http://www.routeviews.org.

[24] RFC 792. Internet control message protocol, September 1981.

[25] A.Danalis and Constantinos Dovrolis. Anemos: An autonomous network monitoring system. In *In Proceedings of the 4th Passive and Active Measurements (PAM) Workshop*, April 2003.

[26] Aditya Akella, Srinivasan Seshan, and Anees Shaikh. An empirical evaluation of wide-area Internet bottlenecks. In *Proc. ACM IMC*, October 2003.

[27] Mark Allman, Ethan Blanton, and Wesley M. Eddy. A scalable system for sharing internet measurements. In *Proc. PAM*, March 2002.

[28] Kostas G. Anagnostakis, Michael B. Greenwald, and Raphael S. Ryger. cing: Measuring network-internal delays using only existing infrastructure. In *Proc. IEEE INFOCOM*, April 2003.

[29] Giuseppe Di Battista, Maurizio Patrignani, and Maurizio Pizzonia. Computing the types of the relationships between autonomous systems. In *Proc. IEEE INFOCOM*, April 2003.

[30] Jean-Chrysostome Bolot. End-to-end pakcet depay and loss behavior in the Internet. In *Proc. ACM SIGCOMM'93*, San Francisco, CA, September 1993.

[31] Tian Bu, Nick Duffield, Francesco Lo Presti, and Don Towsley. Network tomography on general topologies. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 21–30. ACM Press, 2002.

[32] R. Carter and M. Crovella. Measuring bottleneck link speed in packet-switched networks. Technical report, Boston University Computer Science Department, March 1996.

[33] Meeyoung Cha, Sue Moon, Chong-Dae Park, and Aman Shaikh. Positioning Relay Nodes in ISP Networks. In *Proc. IEEE INFOCOM*, March 2005.

[34] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *Proc. ACM SIGCOMM*, August 2004.

[35] Yanghua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early experience with an Internet broadcast system based on overlay multicast. In *Prof. of USENIX Annual Technical Conference*, June 2004.

[36] Leonard Ciavattone, Alfred Morton, and Gomathi Ramachandran. Standardized active measurements on a tier 1 ip backbone. In *IEEE Communications Magazine*, pages 90–97, San Francisco, CA, June 2003.

[37] Mark Claypool, Robert Kinicki, Mingzhe Li, James Nichols, and Huahui Wu. Inferring queue sizes in access networks by active measurement. In *Proc. PAM*, April 2004.

[38] Coralreef. `http://www.caida.org/tools/measurement/coralreef/`.

[39] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[40] M. Costa, M. Castro, A. Rowstron, , and P. Key. PIC: Practical Internet coordinates for distance estimation. In *International Conference on Distributed Systems*, March 2004.

[41] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *Proc. ACM SIGCOMM*, August 2004.

[42] Dag. `http://dag.cs.waikato.ac.nz/`.

[43] Peter A. Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Dean Sutherland. The architecture of the remos system. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing*, California, USA, August 2001.

[44] Benoit Donnet, Philippe Raoult, Timur Friedman, and Mark Crovella. Efficient algorithms for large-scale topology discovery. In *Proc. ACM SIGMETRICS*, June 2005.

[45] Constantinos Dovrolis, Parmesh Ramanathan, and David Moore. What do packet dispersion techniques measure? In *Proc. of ACM INFOCOM*, April 2001.

[46] Allen B. Downey. Clink: a tool for estimating Internet link characteristics. `http://rocky.wellesley.edu/downey/clink/`.

[47] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. Netscope: Traffic engineering for ip networks. *IEEE Network Magazine, special issue on Internet traffic engineering, 2000*, 2000.

[48] Anja Feldmann, Olaf Maennel, Z. Morley Mao, Arthur Berger, and Bruce Maggs. Locating internet routing instabilities. In *Proc. ACM SIGCOMM*, August 2004.

[49] Chuck Fraleigh, Christophe Diot, Bryan Lyles, Sue B. Moon, Philippe Owezarski, Dina Papagiannaki, and Fouad A. Tobagi. Design and deployment of a passive monitoring infrastructure. In *Proceedings of the Thyrrhenian International Workshop on Digital Communications*, pages 556–575. Springer-Verlag, 2001.

[50] L. Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM Trans. Networking*, December 2001.

[51] S. Gerding and J. Stribling. Examining the tradeoffs of structured overlays in a dynamic non-transitive network. In *MIT 6.829 Fall 2003 class project*, December 2003.

[52] Ramesh Govindan and Vern Paxson. Estimating router ICMP generation delays. In *Proc. PAM*, March 2002.

[53] Junghee Han, David Watson, and Farnam Jahanian. Topology aware overlay networks. March 2005.

[54] Khaled Harfoush, Azer Bestavros, and John Byers. Measuring bottleneck bandwidth of targeted path segments. In *Proc. IEEE INFOCOM*, April 2003.

[55] Ningning Hu, Li Erran Li, Zhuoqing Mao, Peter Steenkiste, and Jia Wang. A measurement study of internet bottlenecks. In *Proc. IEEE INFOCOM*, April 2005.

[56] Ningning Hu, Li Erran Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *Proc. ACM SIGCOMM*, August 2004.

[57] Ningning Hu, Oliver Spatscheck, Jia Wang, and Peter Steenkiste. Optimizing network performance in replicated hosting. In *the Tenth International Workshop on Web Caching and Content Distribution (WCW 2005)*, September 2005.

[58] Ningning Hu and Peter Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE JSAC Special Issue in Internet and WWW Measurement, Mapping, and Modeling*, 21(6), August 2003.

[59] Ningning Hu and Peter Steenkiste. Improving TCP startup performance using active measurements: Algorithm and evaluation. In *IEEE International Conference on Network Protocols (ICNP) 2003*, Atlanta, Georgia, November 2003.

[60] Ningning Hu and Peter Steenkiste. Exploiting internet route sharing for large scale available bandwidth estimation. In *Proc. of ACM/USENIX IMC*, October 2005.

[61] Internet end-to-end performance monitoring. `http://www-iepm.slac.stanford.edu/`.

[62] Internet performance measurement and analysis project. `http://www.merit.edu/~ipma/`.

[63] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, August 1988.

[64] Van Jacobson. pathchar - a tool to infer characteristics of internet paths, 1997. Presented as April 97 MSRI talk.

[65] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proc. ACM SIGCOMM*, August 2002.

[66] Manish Jain and Constantinos Dovrolis. End-to-end estimation of the available bandwidth variation range. In *Proc. ACM SIGMETRICS*, June 2005.

[67] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc, New York, NY, 1991.

[68] Guojun Jin and Brian Tierney. System capability effect on algorithms for network bandwidth measurement. In *Internet Measurement Conference (IMC) 2003*, Miami, Florida, USA, October 2003.

[69] Charles Robert Simpson Jr. and George F. Riley. Neti@home: A distributed approach to collecting end-to-end network performance measurements. In *Proc. PAM*, April 2004.

[70] k claffy, G. Miller, and K. Thompson. the nature of the beast: recent traffic measurements from an internet backbone. In *Proceedings of ISOC INET '98*, July 1998.

[71] Rohit Kapoor, Ling-Jyh Chen, Li Lao, Mario Gerla, , and M. Y. Sanadidi. Capprobe: A simple and accurate capacity estimation technique. In *Proc. ACM SIGCOMM*, August 2004.

[72] Srinivasan Keshav. Packet pair flow control. *IEEE/ACM Transactions on Networking*, February 1995.

[73] Balachander Krishnamurthy, Harsha V. Madhyastha, and Oliver Spatscheck. Atmen: a triggered network measurement infrastructure. In *WWW'05: Proceedings of the 14th International Conference on World Wide Web*, pages 499–509, New York, NY, USA, 2005. ACM Press.

[74] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental Study of Internet Stability and Wide-Area Network Failures. In *Proc. International Symposium on Fault-Tolerant Computing*, June 1999.

[75] C. Labovitz, R. Malan, and F. Jahanian. Internet routing stability. *IEEE/ACM Trans. Networking*, 6(5):515–528, October 1998.

[76] C. Labovitz, R. Malan, and F. Jahanian. Origins of pathological Internet routing instability. In *Proc. IEEE INFOCOM*, March 1999.

[77] Kevin Lai and Mary Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *Proc. of the USENIX Symposium on Internet Technologies and Systems*, March 2001.

[78] X. Liu, K. Ravindran, and D. Loguinov. Multi-hop probing asymptotics in available bandwidth estimation: Stochastic analysis. In *Proc. ACM IMC*, October 2005.

[79] Bruce A. Mah. pchar: A tool for measuring internet path characteristics, 2001. `http://www.employees.org/~bmah/Software/pchar/`.

[80] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level internet path diagnosis. In *Proc. SOSP*, October 2003.

[81] Z. Morley Mao, Jennifer Rexford, Jia Wang, and Randy Katz. Towards an Accurate AS-level Traceroute Tool. In *Proc. ACM SIGCOMM*, September 2003.

[82] Zhuoqing Morley Mao, Lili Qiu, Jia Wang, and Yin Zhang. On AS-level path inference. In *to appear in SIGMETRICS'05*, June 2005.

[83] Matthew Mathis and Jamshid Mahdavi. Diagnosing internet congestion with a transport layer performance tool. In *Proc. INET'96*, Montreal, Canada, June 1996.

[84] A. McGregor, H-W. Braun, and J. Brown. The NLANR network analysis infrastructure. *IEEE Communications Magazine*, 38(5):122–128, May 2000.

[85] Bob Melander, Mats Bjorkman, and Per Gunningberg. A new end-to-end probing and analysis method for estimating bandwidth bottlenecks. In *Proc. IEEE GLOBECOM*, November 2000.

[86] Minc: Multicast-based inference of network-internal characteristics. `http://www.research.att.com/projects/minc/`.

[87] Federico Montesino-Pouzols. Comparative analysis of active bandwidth estimation tools. In *Proc. PAM*, April 2004.

[88] T. S. Eugene Ng and Hui Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. IEEE INFOCOM*, June 2002.

[89] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. ACM SIGCOMM*, September 1998.

[90] Konstantina Papagiannaki, Nina Taft, and Christophe Diot. Impact of flow dynamics on traffic engineering design principles. In *Proc. IEEE INFOCOM*, March 2004.

[91] A. Pasztor and D. Veitch. Active probing using packet quartets. In *ACM SIGCOMM Internet Measurement Workshop 2002*, 2002.

[92] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U.C. Berkeley, May 1996.

[93] Vern Paxson, Andrew Adams, and Matt Mathis. Experiences with nimi. In *In Proceedings of the Passive and Active Measurement Workshop*, 2000.

[94] Vern Paxson, Andrew Adams, and Matt Mathis. Experiences with nimi. In *In Proc. of the Passive and Active Measurement Workshop (PAM)*, 2000.

[95] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti. Lighthouses for scalable distributed location. In *International Workshop on Peer-to-Peer Systems*, February 2003.

[96] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth estimation: Metrics, measurement techniques, and tools. November 2003.

[97] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. In *Passive and Active Measurements (PAM) workshop*, April 2004.

[98] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proc. IEEE INFOCOM*, April 2001.

[99] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. Bgp routing stability of popular destinations. In *Proc. ACM IMW*, November 2002.

[100] Vinay Ribeiro. Spatio-temporal available bandwidth estimation for high-speed networks. In *Proc. of the First Bandwidth Estimation Workshop (BEst)*, December 2003.

[101] Vinay Ribeiro, Rudolf Riedi, Richard Baraniuk, Jiri Navratil, and Les Cottrell. pathchirp: Efficient available bandwidth estimation for network paths. In *Proc. PAM*, April 2003.

[102] S. Savage. Sting: a TCP-based network measurement tool. In *Proc. of the 1999 USENIX Symposium on Internet Technologies and Systems*, October 1999.

[103] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP buffer tuning. In *Computer Communications Review*, volume 28, October 1998.

[104] Srinivasan Seshan, M. Stemm, and Randy H. Katz. Spand: shared passive network performance discovery. In *In Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97)*, Monterey, CA, December 1997.

[105] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *Proc. IEEE INFOCOM*, April 2004.

[106] Alok Shriram, Margaret Murray, Young Hyun, Nevil Brownlee, Andre Broido, Marina Fomenkov, and k claffy. Comparison of public end-to-end bandwidth estimation tools on high-speed links. In *Passive and Active Measurement Workshop*, March 2005.

[107] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. Improving accuracy in end-to-end packet loss measurement. In *ACM SIGCOMM*, August 2005.

[108] Neil Spring, Ratul Mahajan, and Tom Anderson. Quantifying the Causes of Path Inflation. In *Proc. ACM SIGCOMM*, August 2003.

[109] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with rocketfuel. In *Proc. ACM SIGCOMM*, August 2002.

[110] Neil Spring, David Wetherall, and Tom Anderson. Scriptroute: A public internet measurement facility. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[111] Mark Stemm, Srinivasan Seshan, and Randy H. Katz. A network measurement architecture for adaptive applications. In *Proc. IEEE INFOCOM*, Monterey, CA, March 2000.

[112] W. Richard Stevens. *Unix Network Programming, Third Edition , Volume 1*. Prentice Hall.

[113] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. ACM IMC*, October 2003.

[114] Lakshminarayanan Subramanian, Sharad Agarwal, Jennifer Rexford, and Randy H. Katz. Characterizing the Internet hierarchy from multiple vantage points. In *Proc. IEEE INFOCOM*, June 2002.

[115] Surveyor. `http://www.advanced.org/surveyor/`.

[116] Ttcp. `http://ftp.arl.mil/~mike/ttcp.html`.

[117] Limin Wang, Vivek Pai, and Larry Peterson. The effectiveness of request redirection on cdn robustness. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[118] Rich Wolski, Neil T. Spring, , and Jim Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1999. also UCSD Technical Report Number TR-CS98-599, September, 1998.

[119] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *Proc. of OSDI*, December 2004.

[120] Yin Zhang, Nick Duffield, Vern Paxson, and Scott Shenker. On the constancy of internet path properties. In *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[121] Yin Zhang, Vern Paxson, and Scott Shenker. The stationarity of internet path properties: Routing, loss, and throughput. Technical report, May 2000.