END-TO-END TECHNIQUES FOR NETWORK RESOURCE MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Manpreet Singh

August 2006

END-TO-END TECHNIQUES FOR NETWORK RESOURCE MANAGEMENT

Manpreet Singh, Ph.D.

Cornell University 2006

Network-based applications have evolved from simple web browsing to complex commercial transactions. This evolution has made a clear case for network QoS mechanisms, since the delays seen on the network often form the major component of an end-user's perceived delay. The key challenge in providing network QoS is the distributed and decentralized nature of the network resource.

In this thesis, we have attempted to develop techniques for exposing the network as a managed resource to applications without requiring any explicit support from the network elements like routers, gateways, ISPs, etc. As part of this broad initiative, we have developed two key components: MPAT (used to control the network from end-hosts), and Netmapper (used to monitor the network from end-hosts).

MPAT is the first truly scalable algorithm for providing differential services to a group of TCP flows that share the same bottleneck link. The challenge is to hold the cumulative fair share of all flows, while being fair to the background traffic. Our system can give more bandwidth to one TCP flow at the expense of lower bandwidth to another flow, in such a way that the sum total of the bandwidth that the two flows get is same as the fair share of two TCP flows. Using experiments on the real Internet, we show that our system can share congestion state across more than 100 TCP flows with throughput differentials of 95:1.

We also developed a network mapping and annotation service (called Netmapper) for distributed applications sensitive to bandwidth availability, latency, or loss. The system exposes internal network state (e.g. bandwidth available at links, location of bottleneck links) using only end-to-end measurements that are TCP-friendly. Our novel scheme to identify the location of bottleneck links improves upon existing methods for bottleneck identification, which detect incorrect bottleneck link if the bottleneck is due to transient network congestion. We built Netmapper, and show, using experiments on the real Internet, that the system consumes very small amount of bandwidth, and converges even with dynamically varying network state.

BIOGRAPHICAL SKETCH

Manpreet Singh was born on Nov 18, 1978 in Delhi, India where he spent all of his formative years. In his mother tongue Punjabi, *Manpreet* means *one who is loved by everyone from heart*. He attended the Indian Institute of Technology, Delhi from 1996 to 2001 and graduated with an Integrated Masters of Technology in Mathematics and Computing. He came to Cornell University in the Fall of 2001 and began his doctoral studies in Computer Science. He pursued his research in distributed systems and networking under the guidance of Prof. Paul Francis and Dr. Prashant Pradhan. He also pursued a minor in Business and Finance at Johnson Graduate School of Management. He received his Masters degree in Computer Science in May 2004 and a Doctoral degree in Computer Science in August 2006. During his stay at Cornell, he interned at IBM T. J. Watson Research and Microsoft Research and spent very productive and enjoyable summers at Hawthorne, NY and Seattle, WA respectively. Manpreet's research interests include distributed systems, networking, databases. He loves to build challenging systems that can make an impact on the real world.

To Mumma, Papa, Veer, Bhabhi, Taarini and Simran

ACKNOWLEDGEMENTS

"Guru Gobind dou khare, kake lagun pain ?

Balihari Guru apane, jin Govind diyo bataye."

Lord Kabir was once faced in a delusion as to whose feet to touch first when both his Guru (teacher) and God were standing before him at the same time. He decided to choose the Guru, who had shown him the path to the God (meaning Guru is superior to God).

First and foremost, my deep appreciation goes to my advisors Prof. Paul Francis and Dr. Prashant Pradhan, who have guided me through my research work. I started the work on this thesis during my internship at IBM T.J. Watson in summer 2003, where I worked under the guidance of Ms. Donna Dillenberger and Dr. Prashant Pradhan (Networking Software and Services group). The preliminary results were very promising, and I decided to continue this work as part of my PhD thesis.

I have also benefited a lot from the guidance of Ken Birman and Robbert van Renesse during the early phase of my PhD. I extend my gratitude to my other committee members Zygmunt Haas and Alan Demers, who provided helpful input on this thesis. Special thanks to Sambit Sahu from IBM and Jitu Padhye from Microsoft for being great mentors, and always guiding me. I also acknowledge my other colleagues at IBM (Debanjan Saha, Alan Bivens, Anees Shaikh, Eric Nahum, Joshua Knight) and Microsoft (Alec Wolman, Brain Zill, Paramvir Bahl) with whom I have interacted a lot during the summer internships. I also got valuable guidance from my seniors Indranil, Rama, Ranveer, Abhinandan and Rohit.

I feel very fortunate to have great friends like Pallavi, Chandar, Anisa, Pranav, Tanmoy, Mou, Aseem, Saurabh, Shobhit, Reshmi, Ashok, Mahesh, Sudhakar, Kartik, Sagnik, Charanjit, Asita, Giri, Lakshmi, Ganesh, Hitesh, Saikat, Joy, Vivek, Vidya who made my stay at Cornell a memorable period for the whole of life.

I would like to dedicate this thesis to my family, whose support over the years has enabled me to pursue a career path directed from the core of my heart. They have always encouraged and guided me to independence, never trying to limit my aspirations. I am forever indebted to them for their understanding, endless patience and encouragement when it was most required. I shall always consider myself extremely fortunate to have had the opportunity - all due to them - to discover the seductive powers lying in the pursuit of knowledge.

And most of all, thanks to God, in whom *nothing* is impossible.

TABLE OF CONTENTS

1 Introduction **1**
1.1 Need for an end-to-end control 1
1.2 Thesis overview 3
1.2.1 Bandwidth apportionment amongst TCP flows 4
1.2.2 Monitoring the network from end-hosts 5
1.2.3 Locating internet bottlenecks 6
1.3 Relevance of this thesis to IBM 7
1.4 Thesis outline 8

2 MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS **10**
2.1 Introduction 10
2.2 Existing Approaches 13
2.3 QoS through aggregate congestion management 21
2.3.1 An illustration using two flows 21
2.3.2 The general case 26
2.3.3 The MPAT Algorithm 27
2.3.4 Implementation 32
2.4 Evaluation 33
2.4.1 Scalability 34
2.4.2 Performance differentiation 34
2.4.3 Fairness to background flows 35
2.4.4 Interaction between multiple MPAT flows 36
2.4.5 Adaptation to changing performance requirements 38
2.4.6 Reduced variance 42
2.4.7 Adaptation to Transient Congestion 43
2.4.8 Bursty background traffic 45
2.5 Conclusions 46

3 Netmapper: A network mapping and annotation service for distributed applications **52**
3.1 Monitoring the network 52
3.2 Motivation and related work 54
3.3 Our contributions 56
3.4 Network annotation algorithms 58
3.4.1 Basic idea behind annotating a link 59
3.4.2 Naive algorithm 59
3.4.3 An improved algorithm with bottleneck identification 63
3.4.4 The general case 66
3.4.5 Use of Dynamic Programming 75
3.4.6 Choosing the minimum number of flows 75

3.4.7 Identifying bottleneck link with simultaneous flows 79
3.4.8 Estimating bandwidth for multiple TCP flows 81
3.5 Experimental Results 82
3.5.1 Variability in bandwidth 85
3.5.2 Shift in location of bottleneck link 87
3.5.3 Fast converging algorithm and consistent graph 90
3.5.4 Consistency across time 97
3.5.5 Reduction in the amount of probe traffic 101
3.6 Conclusions 105

4 Tneck: A tool to identify bottlenecks 108
4.1 Bottleneck Identification 108
4.1.1 TCP-based bottleneck estimation 109
4.1.2 Need for parallel traceroute 112
4.2 Experimental results 113
4.2.1 Bottleneck due to transient congestion 114
4.2.2 Effect on background traffic 115
4.2.3 The role of TCP and parallel traceroute 115
4.2.4 Limitations of Tneck 116
4.3 Conclusions 117

5 Conclusions and future work 118

Bibliography 122

LIST OF TABLES

2.1 Table showing the bandwidth obtained by two mulTCP flows running simultaneously through the same bottleneck link over Emulab network. 24
2.2 Table showing the bandwidth obtained by TFRC(+mulTCP) connections along with standard TCP connection. 25
2.3 Symbols and their meanings 28
2.4 Multiple MPAT aggregates running simultaneously cooperate with each other (day time). 39
2.5 Multiple MPAT aggregates running simultaneously cooperate with each other (night-time). 40

3.1 Information about various nodes used for the Netmapper experiments. 83
3.2 List of nodes used in various experiments of Netmapper. 103
3.3 Comparison of the probes for various experiments of Netmapper versus Naive algorithm. 104
3.4 Comparison for the breakup of the total number of probes for various experiments using Netmapper and Naive algorithms. 106

LIST OF FIGURES

2.1 This figure shows the scalability of mulTCP scheme. 18
2.2 This figure shows the number of Fast Retransmits incurred by mulTCP scheme over a period of five minutes. 19
2.3 As N increases, due to multiple losses within the same window, fast retransmit does not trigger, causing a large number of timeouts, and hence limiting the scalability of the mulTCP scheme. 20
2.4 As N increases, variance of a mulTCP flow keeps increasing. . . . 21
2.5 Mean bandwidth and standard deviation obtained by a standard TCP flow running in parallel with a mulTCP flow. 22
2.6 As N increases, the total link utilization for mulTCP decreases. . . 23
2.7 MPAT can apportion bandwidth among its flows, irrespective of the total bandwidth available. 36
2.8 MPAT can hold the total fair share while apportioning bandwidth in the desired ratio. 37
2.9 Competing MPAT aggregates cooperate with each other. 38
2.10 mulTCP scheme exhibits very high variance in bandwidth. 41
2.11 MPAT exhibits much lower variance due to sharing of losses. . . . 42
2.12 MPAT is friendly to the background traffic. 43
2.13 This figure shows the how absolute bandwidth of 3 MPAT flows adapts itself very quickly to dynamically changing performance requirements. 44
2.14 This figure shows the how relative bandwidth of 3 MPAT flows adapts itself very quickly to dynamically changing performance requirements. 45
2.15 This figure shows how the total bandwidth achieved by an MPAT aggregate (N=16) adapts itself with additional **UDP** traffic introduced at t=120 and then removed at t=240. 46
2.16 A deeper look into Figure 2.15 at t=120 to see the effect of introducing additonal **UDP** traffic. 47
2.17 A deeper look into Figure 2.15 at t=240 to see the effect of removing additonal **UDP** traffic. 48
2.18 This figure shows how the total bandwidth achieved by an MPAT aggregate (N=16) adapts itself with additional **TCP** traffic introduced at t=120 and then removed at t=240. 49
2.19 A deeper look into Figure 2.18 at t=120 to see the effect of introducing additonal **TCP** traffic. 50
2.20 A deeper look into Figure 2.18 at t=120 to see the effect of introducing additonal **TCP** traffic. 51

3.1 Example topology used to show how the improved algorithm with bottleneck identification reduces the amount of probe traffic needed to annotate a graph, as compared to a naive scheme. 60

3.2 Annotation steps with naive algorithm for the topology in Figure 3.1. The annotation requires a total of seven steps, three of which drive multiple probe flows simultaneously into the network. 61
3.3 Annotation steps for improved algorithm with bottleneck identification for the topology in Figure 3.1. The annotation requires only four steps (down from seven steps taken by the naive scheme), each requiring only one probe flow. 65
3.4 Example showing that when we identify a link as the bottleneck, we annotate each edge on all paths of which the link is a part, up to the point where another path is incident on this path. 67
3.5 Topology and list of end-to-end paths used to illustrate the improved algorithm using bottleneck identification for the general case. 67
3.6 First six steps for the pre-processing stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5. Continued to Figure 3.7 68
3.7 Steps 7 and 8 for the pre-processing stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5. Continued from Figure 3.6 69
3.8 Examples showing the subgraph spanned by paths through a given edge. This is used to calculate the Maxflow through the edge. . . 72
3.9 Example showing how we add two new nodes in the subgraph, in order to calculate the max-flow edge e_6. 73
3.10 Partial steps for the recursive stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5. . . 76
3.11 Pre-processing stage for the improved annotation algorithm with bottleneck identification (contd. to Figure 3.12) 77
3.12 Recursive stage for the improved annotation algorithm with bottleneck identification (contd. from Figure 3.11) 78
3.13 Example showing how to identify bottleneck link with simultaneous flows. 80
3.14 CDF for *average* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation. 86
3.15 CDF for *median* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation. 87
3.16 CDF for *maximum* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation. 88
3.17 CDF for *average* length of Change-Free Region for shift in location of bottleneck link along a path. 89
3.18 CDF for *median* length of Change-Free Region for shift in location of bottleneck link along a path. 90
3.19 CDF for *maximum* length of Change-Free Region for shift in location of bottleneck link along a path. 91

3.20 Final annotated graph output by Netmapper for four source nodes (around West Coast) and eight destination nodes (around East Coast). 92
3.21 Final annotated graph output by *Netmapper* for single source node and eight destination nodes. 93
3.22 Final annotated graph output by *Naive algorithm* for single source node and eight destination nodes. 94
3.23 Validating the accuracy (*relative change*) of bandwidth annotations output by our improved algorithm (using bottleneck identification), with reference to the naive scheme. 95
3.24 Validating the accuracy (*absolute difference*) of bandwidth annotations output by our improved algorithm (using bottleneck identification), with reference to the naive scheme. 96
3.25 CDF for the *relative change* in bandwidth annotation of a link at two different times. 99
3.26 CDF for the *difference* in bandwidth annotation of a link at two different times. 100
3.27 Probability that an edge is identified bottleneck in one graph, but not another as a function of the time interval between runs. . . . 102

4.1 TCP-based identification of the bottleneck link along a path. . . . 110
4.2 This figure shows how using standard traceroute may not provide accurate information about the queues being built at the bottleneck link. 111
4.3 This figure shows how the modified (parallel) traceroute can capture link latencies very quickly and frequently. 112
4.4 Emulab topology used to show that existing tools detect an incorrect bottleneck link if the bottleneck is due to transient congestion. 114

Chapter 1

Introduction

It is hard to find an example of a commercial application or service today that does not use the Internet or some other form of underlying network. Yet we are equally unlikely to find an example of an application that has explicit mechanisms to manage its network performance and availability. The apparent reason is that network performance and availability is acceptable most of the time. Yet service and content providers constantly seek (and in some cases, are willing to pay for) solutions that essentially provide different ways of bypassing network bottlenecks and failures. Well-studied examples of such solutions are content distribution networks (e.g., Akamai [59]), application-level overlays [5, 8, 38, 47], ISP selection (e.g., multi-homing [60]) and peer-to-peer communication. Several measurement studies have also shown that network bottlenecks are not confined to specific parts of the network, (e.g. access links [53]), and hence are not localized problems that can be fixed easily. There is also no easy or cost efficient way to simply scale or a-priori plan the network capacity between all of an application's endpoints, and businesses already pay significant fixed bandwidth costs to data-centers and network service providers, just for access to the network.

1.1 Need for an end-to-end control

We assert that even though network performance and availability is a real problem, and is a key determinant of an application's end-to-end user experience, most applications today do not have explicit management of network performance. The primary reason being that network performance management has traditionally been thought of as a problem of providing QoS capabilities in the network, which remains an unrealized vision. A number of schemes like RSVP [14], Intserv [15], Diffserv [16], etc. have been proposed in the past. Technically there is little wrong with these schemes; they can be made to work. Nevertheless none of these have been deployed on a large scale in practice. The exact reason for this is not easy to pin down, but ultimately the problem stems from the fact that too many networking players need to come together at once to provide effective network QoS. Both the providers of end equipment and the providers of networking equipment need to agree on a common set of protocols. The ISPs have to deploy the protocols, and application developers have to build applications that use the protocols. In the end, no business model was compelling enough to bring all the players together at the same time.

We envision that network performance management is more of a constrained resource allocation/planning problem, where an application must intelligently "pack" its traffic demand within the capacity available on the network paths it may use. These paths are in turn constrained by IP routing between the application's endpoints, and the knobs the application may use to pack its traffic over these paths are end-to-end techniques like server selection [59], ISP selection [60] and overlay routing [5, 38].

Moreover, it is simply too complex, and in some cases not possible for applications to plan and exercise these controls by themselves. Exercising end-to-end controls for performance management requires an application to understand the underlying network structure, monitor it, and map its performance requirements to the setting of an end-to-end control knob.

Further, on a longer time scale, application traffic demands should be used to scale and evolve the network in an informed manner. This is possible by making application traffic demand patterns available to network providers for long-term capacity planning. This inherent division of responsibility where the application maps its traffic onto the available capacity over a short term, and network providers make informed capacity scaling decisions over a long term, represents a viable approach to the network performance management problem.

The broad goal of this thesis is to find ways to provide network QoS without requiring changes to the network itself. In other words, we wish to find ways to reallocate network resources from one user or application to another without any explicit support from the network. If this can be done, then it should be possible to reduce the number of players that must sync up to provide QoS. For instance, a single organization might be able to build a QoS-capable network overlaid on top of the "best-effort" Internet and offer "QISP" (Quality Internet Service Provider) services. Or, the computing systems that comprise a Web Services infrastructure in an Enterprise might be able to obtain differential services from the network they connect to.

The key challenge in providing QoS without taking the support of network is to do it in a TCP-friendly way (i.e., the performance of other traffic in the network should not get affected). Otherwise, the network becomes unstable, and the proposed system cannot be deployed on a large scale.

1.2 Thesis overview

The broad goal of this thesis is to expose the network as a managed resource to applications without requiring any explicit support from the network elements like routers, gateways, ISPs, etc. As part of this broad initiative, we have developed two key components: MPAT (used to control the network from end-hosts), and Netmapper (used to monitor the network from end-hosts).

1.2.1 Bandwidth apportionment amongst TCP flows

MPAT is the first truly scalable algorithm for providing differential services to a group of TCP flows that share the same bottleneck link. The challenge is to hold the cumulative fair share of all flows, while being fair to the background traffic. Our system can give more bandwidth to one TCP flow at the expense of lower bandwidth to another flow, in such a way that the sum total of the bandwidth that the two flows get is same as the fair share of two TCP flows. The reason this problem is hard is that the fair share of a TCP flow keeps changing dynamically with time, and the end-hosts do not have an explicit indication of the amount of cross-traffic.

It is well-known that the use of TCP by all network flows entitles each flow to a fair share of its bottleneck link bandwidth. Thus, the flows originating from a server that share a bottleneck link are entitled to a total fair share of that link's bandwidth. We let the server (while sitting at an end-point) aggregate congestion information of all these flows, and apportion the total fair share among these flows in accordance with their performance requirements, while also "holding" the fair share of network bandwidth that the server is entitled to. A flow requiring higher bandwidth than each flow's fair share is allowed to send packets through another flow's open congestion window. Correspondingly, when ACKs are received on a flow, we update the congestion window through which that packet was sent.

Using experiments on the real Internet, we have demonstrated that MPAT can be used to share congestion state across more than 100 TCP flows with throughput differentials of 95:1. In other words, MPAT can give 95 times more bandwidth to one TCP flow over another without hurting the background traffic. This is up to five times better than differentials achievable by known techniques. This suggests that MPAT is an appropriate candidate technology for broader overlay QoS services, both because MPAT provides the requisite differentiation and scalability, and because MPAT can co-exist with other TCP flows and with other MPAT aggregates.

1.2.2 Monitoring the network from end-hosts

In Chapter 3, we describe a network mapping and annotation service for distributed applications sensitive to bandwidth availability, latency, or loss. Applications are expected to use this service as a guide to plan their network resource usage and fault response. Netmapper takes as input the set of end-points for a distributed application, and maps out the network connectivity between them using layer-3 traceroute. The key distinction between Netmapper and other network distance services is that it annotates the edges of this graph with available bandwidth using only end-to end probes (TCP and traceroute). Available bandwidth is a key performance metric for many commercial bandwidth-sensitive applications like streaming media, video/voice conferencing, and shared whiteboard applications (like Webex) with a large customer base. Network latency measurements do not suffice for such applications since path sharing directly impacts available bandwidth. Hence, a bandwidth annotation service must expose the location of the bottleneck as well as the underlying topology (which determines bottleneck sharing) to the application. Netmapper deploys end-to-end TCP probes between the endpoints of an application to estimate the available bandwidth on various network paths. It then identifies the bottleneck links on these paths using a novel TCP-friendly bottleneck identification technique, and annotates the bottleneck link with the probed available bandwidth. Netmapper eventually annotates all the edges whose annotation is theoretically possible given the network topology and the available probe points. The end-to-end probes are intelligently planned so that the annotation requires a minimum number of probes.

We have built Netmapper, and tested it out on the real Internet. Our experience deploying Netmapper shows that the system consumes a very small amount of bandwidth, and converges fast even with dynamically varying network state. Both variability in available bandwidth and shift in location of bottleneck link are slow enough that they allow our system to converge, and maintain a consistent graph.

1.2.3 Locating internet bottlenecks

We have built a novel scheme to identify the location of bottlenecks in a network for a distributed networked application. The system improves upon existing methods [53, 39] for bottleneck identification. Specific contributions over existing schemes include identifying bottlenecks with little to no impact on the regular traffic, and matching TCP's notion of bottleneck (as opposed to raw bandwidth). We show using experiments (under controlled settings of the Emulab environment [56]) that existing bottleneck identification schemes detect an incorrect bottleneck link if the bottleneck is due to transient congestion. We used the bottleneck identification tool as a key primitive in the Netmapper network annotation algorithm described above. The applicability of the tool is broad enough that end-users can use it to estimate the performance of the network path to a given destination, while ISPs may use it to quickly locate network problems, or to guide traffic engineering both at the interdomain and intradomain level.

1.3 Relevance of this thesis to IBM

A killer application of this thesis work for the enterprise network is in workload management for e-business. An enterprise network typically consists of front-end appliance servers, middle-tier web servers, and the backend database servers. These servers are geographically distributed across different parts of the world. Each request from a customer needs to be processed by multiple servers. So, there are large numbers of TCP connections running simultaneously between these servers at any given instant of time. Some of these TCP flows serve requests for "gold" customers, while others are for "best-effort" customers.

We use eWLM group at IBM as an example. eWLM is developing a workload manager that controls various resources in order to provide end-to-end response time guarantees to each customer. eWLM currently manages various resources like CPU, memory, inbound network accept queues, etc. As long as eWLM's current resources are local to the server, mechanisms to control them can also be locally created in servers. However, the network is a resource in which contention occurs outside the server, and at some point that the server cannot control. Network delays are often a significant component of the overall delay seen by a request [1, 2, 25, 38, 53], and hence we need to develop mechanisms to manage them.

The MPAT system developed as part of this thesis exposes the network as a managed resource to workload manager, hence, it can be used to provide a knob to the workload manager for controlling the delays that occur inside the network (without requiring any cooperation from the network elements like routers, ISPs, etc). This will provide faster service to gold customers over first-time visitors. The approach can, in general, be applied to any system that needs to provide some kind of service differentiation inside the network without involving the routers or other network elements. This work will become increasingly significant as IBM deploys more service-level agreements for B2B web services.

Because IBM employees frequently download software in the process of developing, engineering, testing and marketing products, IBM must ensure that its employees utilize the most efficient methods for downloading large files. Many employees use IBM Standard Software Installer to download and install software using file transfer protocol and dedicated servers throughout the IBM worldwide intranet. However, download times for software installation, demos, videos and other types of large files take up hundreds of thousands of employee hours each year. To reduce this time, improve productivity and use existing resources more efficiently, IBM IntraGrid, a testbed for Grid services and solutions, is developing solutions to download large digital files faster than before. The Netmapper system developed as part of this thesis can be used to monitor the network, and hence manage network delays for IBM Intra-grid. Netmapper builds a map of the IP network from the perspective of a set of hosts using the network, and annotates each link with available bandwidth using end-to-end techniques. The idea here is that a smart scheduler could use this annotated network map to maximize its performance and minimize network load. We are planning to integrate Netmapper in server selection for the download Grid.

1.4 Thesis outline

The remainder of this thesis is organized as follows. In Chapter 2, we describe MPAT, an end-to-end approach for bandwidth apportionment. In Chapter 3, we provide a network mapping and annotation service for distributed applications. Chapter 4 describes a novel technique to identify the location of bottlenecks in networks. We conclude in Chapter 5 by describing a set of applications we are interfacing our system with.

Chapter 2

MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS

2.1 Introduction

After the heady optimism of the vision of a universal QoS in the form of RSVP [14, 15], we have come to understand that realistically QoS can only be deployed piecemeal. Rather than a single ubiquitous QoS architecture, what is evolving is an arsenal of tools that can be opportunistically deployed where benefits can clearly be demonstrated. These benefits are most clear where a bottleneck resource (e.g. a single bottleneck link) can be identified, there is an identifiable set of user or traffic classes that would benefit from arbitration of that bottleneck, and the organization with a vested interest in providing QoS has control over the bottleneck resource.

One example is an enterprise network using IP telephony over the Internet[33], where the bottleneck is the access link between the enterprise locations and the Internet. Differentiation between voice and non-voice traffic provides clear benefits, and this traffic can be identified by a router, say by looking for RTP packets. The enterprise has control over the routers on either end of the access link, either directly or by coordinating with its ISP. Another common example is in wireless networking, either 802.11 or cellular, where there may be different types of traffic or different classes of users (gold, silver, bronze customers, or emergency services) vying for scarce radio spectrum. Here, QoS mechanisms may be implemented at several layers in the protocol stack (for instance, [11]), and the wireless network provider controls these protocols.

In these diverse examples, a single organization (enterprise, wireless network provider) was able to apply a QoS tool (router queuing, radio access control) to an identifiable bottleneck resource (enterprise access link, radio spectrum). These examples represent the "low hanging fruit", if you will, of QoS tools—those cases where the opportunity and motivation are clear. What about the more difficult scenarios where there is an underlying network of some complexity, and no single organization has the motivation or wherewithal to implement QoS services in that network?

An interesting approach to create a network with controllable properties, on top of an unstructured Internet, is the overlay network. Overlay networks have been produced for reducing network delays [22], improving network resilience [5], and providing QoS to network applications like games [8]. The basic building block for providing QoS over an overlay network is the ability to provide QoS over each logical overlay link. Flows between the two ends of an overlay link share the same underlying network path, thus sharing the bottleneck link. Hence mechanisms to arbitrate the bottleneck link bandwidth to provide QoS can be deployed at the ends of the overlay link.

Note that the question of whether overlay networks is a valid approach to providing QoS is an open question, and this chapter does not answer that question. In particular, if the overlay changes the characteristics of the individual TCP flows that pass through it, and if multiple overlays try to "compete" under these circumstances, then the overlays may lose their effectiveness, or worse may destabilize the Internet. What is clear, however, is that if we are to make progress in understanding what can legitimately be achieved with overlays, we need to understand the characteristics of a single hop in the overlay. More specifically, in order to co-exist with Internet traffic (and with other similar overlays), such a QoS mechanism should satisfy certain properties. We state these properties below, assuming there are N flows belonging to an overlay using a logical overlay link.

- *Fairness:* The overlay flows should not steal bandwidth away from other flows (overlay or non-overlay) sharing the same bottleneck. In particular, other TCP or TCP-friendly flows using that bottleneck link should get the same bandwidth as they would with N standard TCP flows using the overlay. This requirement ensures that the overlay's QoS mechanism is "transparent" to the background flows.

- *Utilization:* The overlay flows should be able to hold the fair share of the bottleneck link bandwidth entitled to them. In particular, the total bandwidth available to the overlay's flows should be equal to the total fair share of N TCP flows. This ensures that the overlay flows do not lose bandwidth to the background traffic.

- *Scalability:* The above properties should hold for large values of N.

The key technical contribution of this chapter is an aggregate TCP congestion management scheme that scales well by the number of flows in the aggregate (we have demonstrated up to N=100), that provides large differentiation ratios (we have demonstrated up to 95:1), and that does so fairly. We call the scheme MPAT, for Multi-Probe Aggregate TCP, because it maintains an active AIMD loop (i.e. a "probe") for every TCP flow in the aggregate. This can be contrasted with mulTCP[3], which tries to obtain N fair shares by running a single, more aggresive AIMD loop. A secondary contribution of this chapter is to provide more experimental data and insights on the existing schemes (primarily mulTCP). Even though the scalability and stability of mulTCP is known to be limited, we feel it is important to provide this data for two reasons. First, contrasting MPAT with mulTCP allows us to clarify and verify our intuitions as to why MPAT performs well. Second, mulTCP, in spite of its stated limitations, is nevertheless still being proposed for overlay QoS [8] and so can be considered the incumbant. Thus it is important to make more direct comparisons between MPAT and mulTCP.

The rest of this chapter is organized as follows. Section 2.2 discusses the existing techniques to provide QoS to TCP or TFRC flows over a single path or bottleneck link, and demonstrates that they fall short of meeting the desirable criteria for such a QoS mechanism. Section 2.3 presents the proposed aggregate TCP approach in detail. Section 2.4 presents a detailed performance evaluation of MPAT using experiments in controlled settings as well as over the Internet, and compares MPAT with proposed techniques. Section 2.5 concludes the chapter, detailing our key findings.

2.2 Existing Approaches

One class of approaches (e.g. pTCP [31]) that provide network QoS in a best-effort network, try to open extra simultaneous TCP connections, instead of a single connection, to give more bandwidth to an application. By contrast, MPAT does not create extra TCP connections. It tries to provide relative QoS among only the TCP flows that the applications normally create. The resulting behavior of schemes like pTCP is clearly not desirable, and in fact CM[2] proposes to explicitly forbid this kind of behavior by using one active AIMD loop (i.e. one congestion "probe") for multiple flows between two given end-points. Further, this approach does not scale to higher performance ratios between flows, since a large number of flows active at a bottleneck lead to significant unfairness in TCP [37]. Of course, the scalability of both MPAT and pTCP is limited by the delay-bandwidth product of the aggregated flows. But pTCP would reach this limit much before MPAT. As an example, we later show in Section 2.4.5 (using experiments on the real Internet) that MPAT can give 95 times more bandwidth to one application over another using only three TCP connections. On the other hand, schemes like pTCP would need to open 95 parallel TCP connections for this.

The idea of providing performance differentiation between TCP flows sharing a path has been discussed in the context of aggregate congestion management approaches [9, 6] like TCP Session [1, 4], Congestion Manager (CM) [2, 10], TCP Trunking [21], A TCP trunk is a TCP connection between two routers, which is shared by multiple TCP flows active between the two routers. The end-to-end flows terminate at the end points of the trunk, where data from all the flows is buffered. The bandwidth available on the trunk is the bandwidth available on the TCP trunk, which evolves using the standard TCP AIMD[29, 30] algorithm. The data from various TCP flows can then be sent over the trunk according to any chosen scheduling policy. CM[2] is a congestion management scheme that provides unified congestion management to TCP as well as non-TCP flows, decouples congestion control from reliability, and ensures that end-points of a connection cannot hog more than their fair share of network bandwidth by opening multiple connections between them. CM defines an aggregate (termed "macroflow") as a set of TCP flows between a source and a destination host (termed "microflows"). Both CM and TCP Session keep one AIMD bandwidth estimation loop active per aggregate, and hence the bandwidth available to the aggregate is that entitled to one TCP flow. Microflows can share this bandwidth according to any chosen scheduling policy. Thus, in case of a logical overlay link, flows between the endpoints of the link could be aggregated into a CM macroflow or a TCP trunk, and performance differentiation could be provided between them.

However, by virtue of using one AIMD loop per trunk or per CM macroflow, both TCP Session and CM, in their current form, do not satisfy the *utilization* requirement. If N flows constitute an aggregate (trunk or macroflow), and the aggregate shares a bottleneck link with M background flows, then with equal sharing within the aggregate, the share of the bottleneck link bandwidth received by the aggregate's flows is $1/(N(M+1))$. If the N flows compete for bandwidth like standard TCP flows, each of the flows would be entitled to a share of $1/(M+N)$. This has also been mentioned in Chapter 7 of [1].

The above problem could be addressed by using a variant of the AIMD loop that acts like N TCP flows, as proposed in [1]. When a TCP flow has a bottleneck link where the loss probability p characterizes the congestion state at the link, TCP's AIMD algorithm allocates the flow a bandwidth given by $B = K/(RTT * sqrt(p))$ [19], where K is proportional to $\sqrt{\frac{\alpha}{\beta} * (1 - \frac{\beta}{2})}$. To achieve performance differentiation among these flows, one possible approach is to play with the parameters α and β in the AIMD algorithm. This is the approach used by mulTCP [3]. The idea behind mulTCP is that one flow should be able to get N times more bandwidth than a standard TCP flow by choosing $\alpha = N/2$ and $\beta = 1/2N$. Thus, the congestion window increases by N (as opposed to 1) when a congestion window worth of packets is successfully acknowledged. Losses leading to fast retransmit cause the window to be cut by (1 - β), and losses leading to a timeout cut down the window to 1. Analytically, a mulTCP flow achieves a bandwidth N times that of a standard TCP flow experiencing the same loss rate. It is thus possible for CM or TCP trunking to use one mulTCP AIMD loop per aggregate to address the utilization problem. An equivalent TFRC [23] variant of mulTCP exists, where the throughput equation of mulTCP can be used in conjunction with a TFRC rate adjustment loop. A TFRC variant of mulTCP is used by OverQoS[8] to estimate the bandwidth entitled to N TCP flows on a logical overlay link.

Note however, that the loss process induced by a single mulTCP flow is quite different from that generated by N independent TCP flows. Hence, it is not clear if a mulTCP flow, especially for large N, would continue to behave like N TCP flows in the network. Also note that the 'amplitude' of mulTCP's AIMD control loop increases with N, leading to an increasingly unstable controller as N grows. This is in contrast to N control loops of N independent TCP flows, where each has a small amplitude and thus tends to be more stable. This also has implications for a TFRC variant of mulTCP, since for large N, the analytically derived bandwidth equation of mulTCP may not represent the actual bandwidth consumed by a mulTCP flow.

Further, it has been noted in [3] that a mulTCP flow cannot act exactly like N independent TCP flows because timeout losses force the entire mulTCP window to be cut down to 1, whereas with N independent TCP flows, such a loss would cut down only one TCP connection's window to 1. In [3], it is shown that this limits the value of N for which a mulTCP flow can achieve as much bandwidth as N independent TCP flows (the recommended value for N is 4 [7]).

The above discussion indicates that QoS approaches based on mulTCP may not meet the fairness and utilization requirements for large N, in turn violating the scalability requirement. In the remaining part of this section, we experimentally investigate this hypothesis.

We conducted experiments on the Emulab network testbed at University of Utah [56] over an emulated link of bandwidth 90Mbps and RTT 50msec. We ran a mulTCP flow along with 10 standard TCP flows through the same bottleneck link. To create a mulTCP connection equivalent to N TCP flows, we set the additive increase factor (α) and multiplicative decrease factor (β) such that $\frac{\alpha}{\beta} = N^2$ [36, 35, 31]. We performed all our experiments with two sets of parameters: $\alpha = N$ (i.e. $\beta = 1/N$), and $\alpha = \sqrt{N}$ (i.e. $\beta = N^{-1.5}$). We used a TCP-SACK based implementation of mulTCP.

Figure 2.1 shows how the achieved bandwidth ratio between mulTCP and a standard background TCP flow varies with N. Figures 2.2 and 2.3 show the corresponding number of fast retransmits and timeouts seen on the mulTCP flow. Note that for large N, due to multiple losses within the same window, fast retransmit does not trigger, causing a large number of timeout-induced window resets. The result is that mulTCP's achieved bandwidth ratio flattens out for $N > 20$ for $\alpha = \sqrt{N}$, and actually falls for $\alpha = N$.

The 45$-$degree line in figure 2.1 represents the ideal achieved bandwidth ratio for a given choice of N. Note that for small values of N, mulTCP's achieved bandwidth ratio lies above this graph, indicating that mulTCP is more aggressive than N TCP flows to the background TCP flow.

Figure 2.4 shows how the mean and variance in the bandwidth achieved by mulTCP varies with N. Note that the variance increases with N, till it eventually flattens out together with the bandwidth. This increased variance is a manifestation of the increasing instability of the modified AIMD controller. Note that even though mulTCP's achieved bandwidth flattens out for large N, figure 2.5 shows

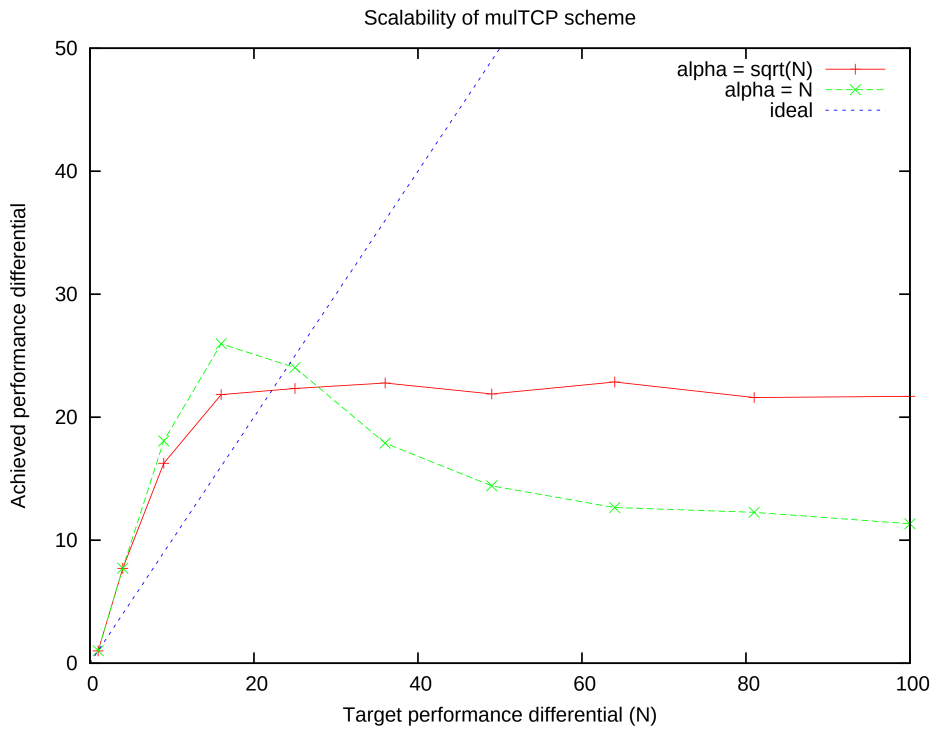


Figure 2.1: This figure shows the scalability of mulTCP scheme.

that the background TCP flow is also not able to get the remaining bandwidth, as its throughput also flattens out. The result, as shown in figure 2.6, is a loss of utilization of the bottleneck link. The reason for this behavior is also an increased number of timeouts both for mulTCP and for background TCP traffic, induced by mulTCP's aggressive control loop for large N.

To analyse the behavior of multiple virtual overlay links using mulTCP, we measured the bandwidth distribution achieved between two mulTCP flows with $N = N_1$ and $N = N_2$. As shown in table 2.1, the achieved bandwidths are not in the ratio N_1/N_2. This suggests that the interaction between two control loops, both with large amplitudes, does not lead to the ideal bandwidth distribution. The fairness of the bandwidth distribution tends to be much better with identical control loops, as used by standard TCP flows. Moreover, the utilization of bottleneck

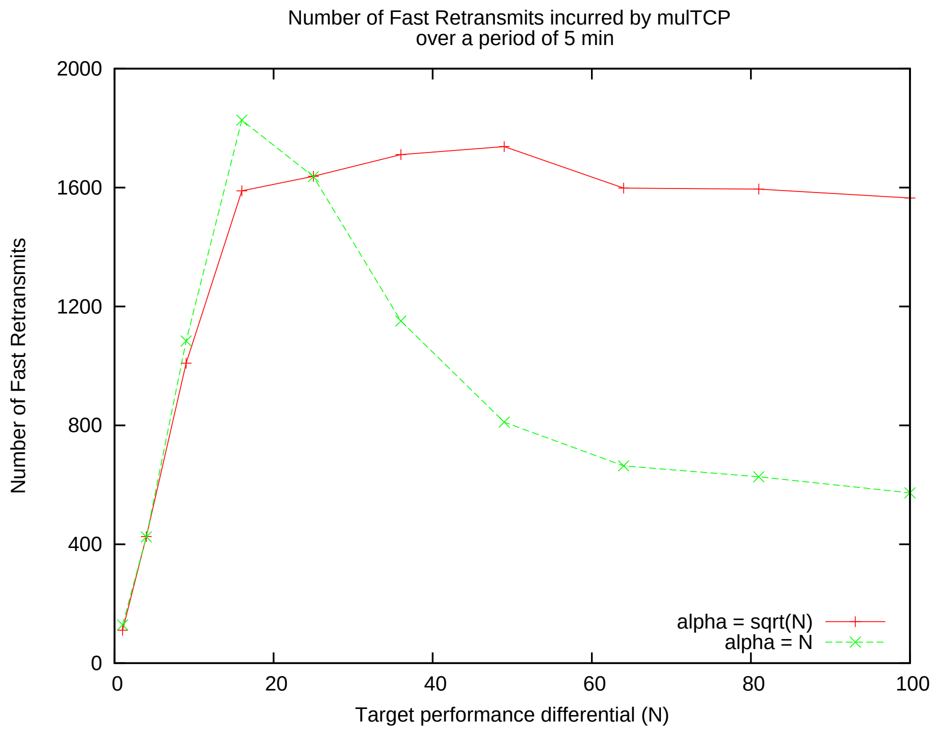


Figure 2.2: This figure shows the number of Fast Retransmits incurred by mulTCP scheme over a period of five minutes.

link (given by $B_1 + B_2$) keeps decreasing with larger values of N_1 and N_2.

Finally, we study the behavior of a TFRC flow using mulTCP, to verify if the slower responsiveness and averaging effects of TFRC mitigate some of mulTCP's problems for large N. In this experiment, 5 mulTCP TFRC flows run simultaneously with 5 background TCP flows. Each mulTCP TFRC flow uses the mulTCP bandwidth equation, and a measurement of the loss probability, to adjust the sending rate in response to loss events. Table 2.2 shows the average bandwidths achieved by a mulTCP flow compared to a TCP flow and their ratio, for varying N. Note that the ratio grows much faster than N, which indicates that the mulTCP TFRC flows aggressively take bandwidth away from background TCP flows as N increases.

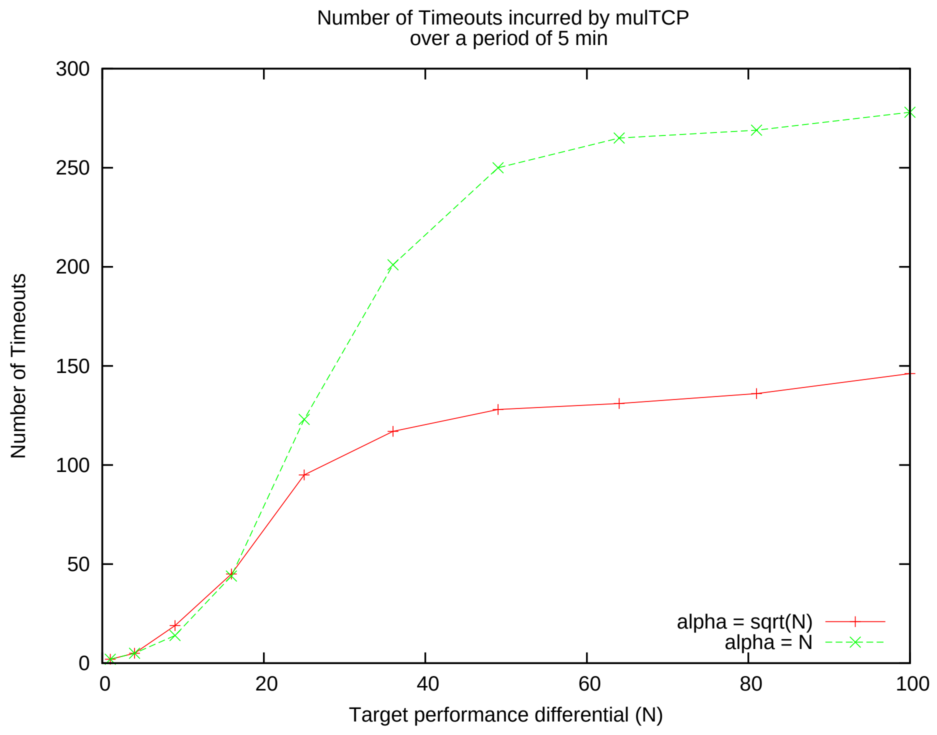


Figure 2.3: As N increases, due to multiple losses within the same window, fast retransmit does not trigger, causing a large number of timeouts, and hence limiting the scalability of the mulTCP scheme.

The above analysis shows that beyond small values of N, mulTCP does not provide an adequate solution to the problem of creating an N-TCP abstraction. In particular, QoS schemes based on mulTCP do not satisfy the scalability requirement. In the next section, we present an aggregate congestion management scheme that satisfies the fairness, utilization as well as scalability requirements mentioned in section 2.1.

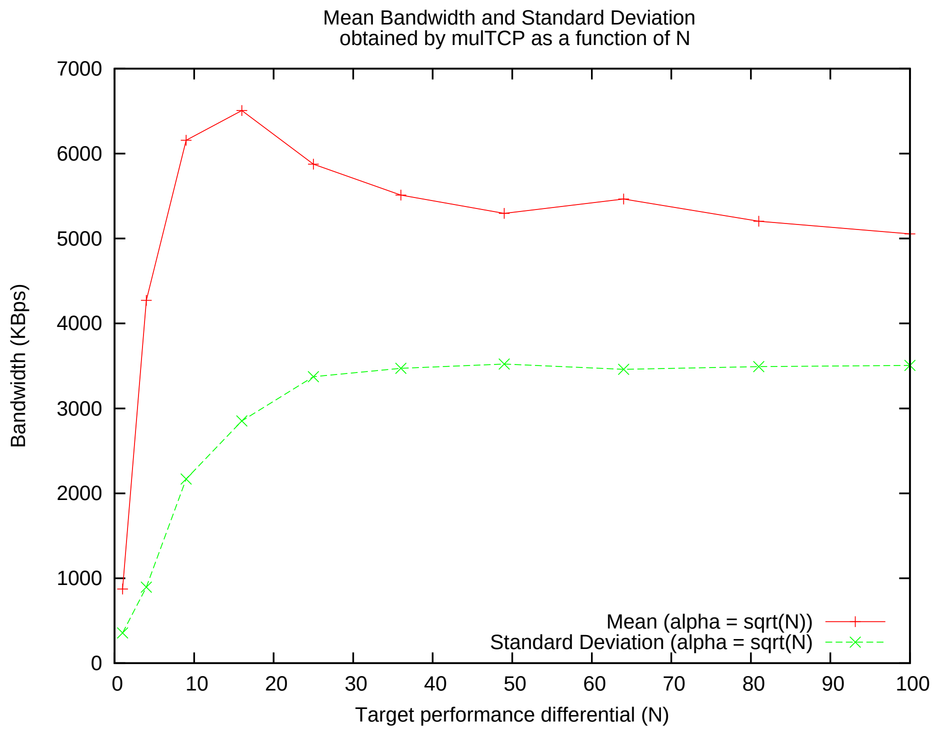


Figure 2.4: As N increases, variance of a mulTCP flow keeps increasing.

2.3 QoS through aggregate congestion management

2.3.1 An illustration using two flows

Before we discuss how our MPAT scheme exactly works, let us start with a simple example. Consider the following scenario: we have two TCP flows running simultaneously between the same end-hosts, thus sharing the same bottleneck link and experiencing similar delay. For ease of illustration, we refer to the first flow as the *red* flow and the second flow as the *blue* flow. [1]

Our goal is to provide performance differentation among these two flows under the following constraints: Firstly, we should not affect other background flows

[1]Please note that in contrast to schemes like pTCP, we are not intentionally opening these flows in order to hog more network bandwidth. These are the flows that applications normally create.

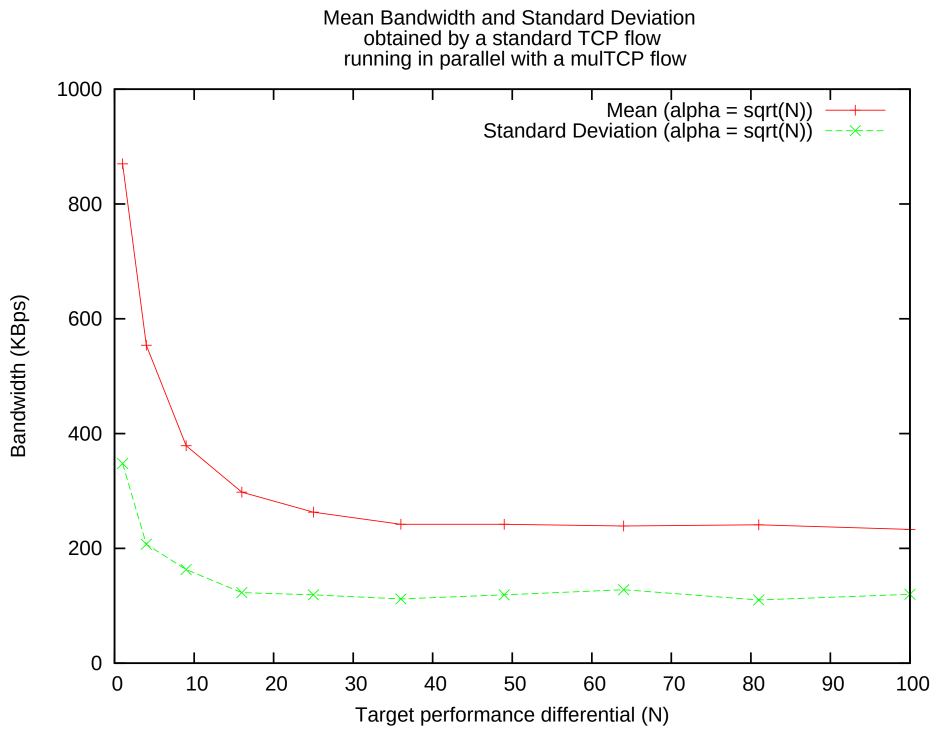


Figure 2.5: Mean bandwidth and standard deviation obtained by a standard TCP flow running in parallel with a mulTCP flow.

running in the network at the same time (in terms of bandwidth, delay, loss rate, etc). This is referred to as the *fairness* property in Section 2.1. Secondly, the sum total of bandwidth acquired by the red flow and the blue flow should be equal to the fair share of two standard TCP flows going through the same bottleneck link at the same time. This is referred to as the *utilization* property in Section 2.1. The reason this problem is hard is that the fair share of a TCP flow keeps changing dynamically with time, depending upon the number of background flows going through the bottleneck link, and the end-hosts do not have an explicit indication of the amount of such cross-traffic.

Suppose we want to apportion the available bandwidth [2] among these two flows

[2]In this example, by 'available bandwidth', we refer to the fair share of two

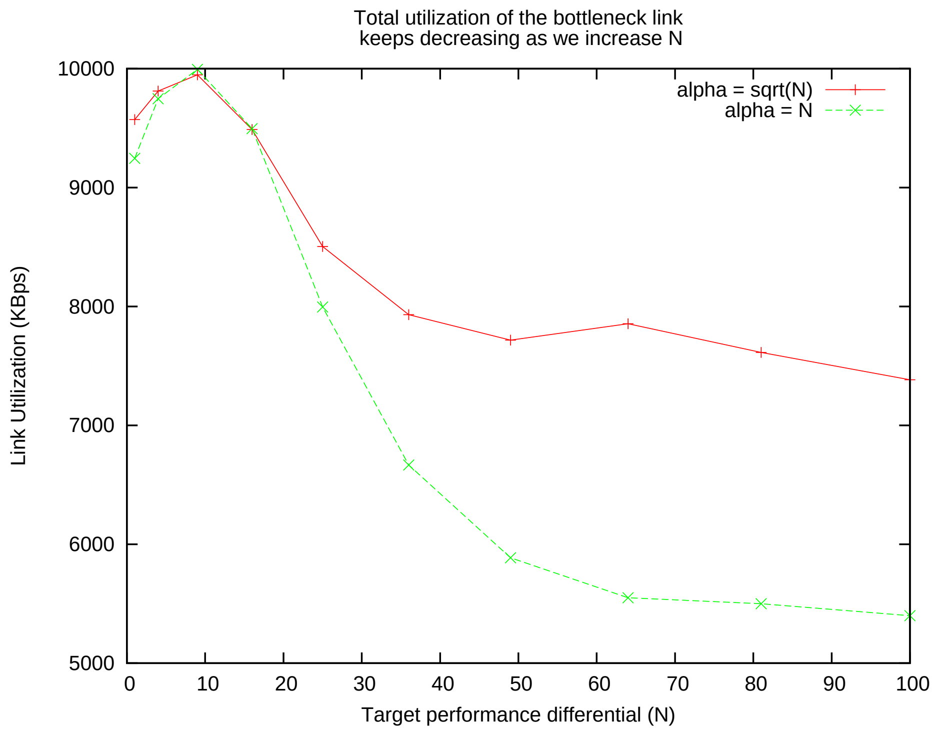


Figure 2.6: As N increases, the total link utilization for mulTCP decreases.

in the ratio 4:1. Assume that the congestion windows for each of the two flows is 5. Thus, the network allows us to send 10 packets every Round Trip Time (RTT). In the case of standard TCP, we would have sent 5 red packets and 5 blue packets every RTT. But the network does not really care how many red and blue packets we actually send, as long as the total number of red and blue packets is 10. If we *aggregate* the congestion information of both these flows at the end-host (the sender side), we can transmit 8 red packets and 2 blue packets every RTT, thus giving four times more bandwidth to the red flow over the blue flow. This would also make sure that we do not hurt the background traffic at all.

The above step allows us to split the instantaneous bandwidth among the two flows in the ratio 4:1. But will we be able to achieve that over a long period of

TCP flows.

Table 2.1: Table showing the bandwidth obtained by two mulTCP flows running simultaneously through the same bottleneck link over Emulab network.

N_1	N_2	B_1 **(KBps)**	B_2 **(KBps)**	B_1/B_2
4	1	3990	498	8.01
9	1	5519	392	14.07
16	1	6091	307	19.84
25	1	5907	280	21.09
9	4	4876	1795	2.72
16	4	5156	1320	3.91
16	9	4340	2833	1.53
25	4	5596	1171	4.78
25	9	4082	2618	1.56
25	16	2925	2264	1.29

time? The answer is NO, and the reason is as follows. Since the probability of each packet getting dropped in the network is the same, the red flow would experience a higher loss rate than the blue flow. This would force the red flow to cut down its congestion window more often than the blue flow. In the long run, the total bandwidth acquired by the two flows would be much less than the fair share of two TCP flows, thus violating the *utilization* property.

To overcome this problem, the fundamental *invariant* that we try to maintain at all times is that *the loss rate experienced by each congestion window should be the same as in standard TCP*. Standard TCP sends data packets, and receives

Table 2.2: Table showing the bandwidth obtained by TFRC(+mulTCP) connections along with standard TCP connection.

N	$TFRC$ **(KBps)**	TCP **(KBps)**	$TFRC/TCP$
5	1857	350	5.3
7	2078	166	12.5
8	2153	101	21.3
9	2196	63	34.8
10	2223	37	60
20	2230	22	101
30	2247	13	172
40	2254	7	322

congestion signals (either explicitly in terms of ECN, or implicitly in terms of duplicate acks, fast retransmissions, timeouts, etc) back from the receiver. If we had used standard TCP for each of the two flows, both the red window and the blue window would (on an average) experience equal number of congestion signals. In order to maintain this property, we first separate reliability from congestion control, as proposed in CM[2]. Next, we decouple the actual growth of congestion window from the identity of the flow whose packets advance the congestion window.

In the above example, when we get 8 red acks, we send three of these to the blue congestion window. In other words, we assign 5 acks to the red window, and 5 to the blue window (3 red + 2 blue). This ensures that each of the two congestion windows experiences similar loss rate, even though we can split the available bandwidth in the ratio 4:1. Please note that since we separate reliability from congestion control, it is the red flow that is responsible to maintain the reliability of each of the 8 red packets (in terms of buffering, retransmission, etc). When we get 8 red acks, we separate each of them into 'reliability ack' and 'congestion signal'. For the purpose of reliability, we send each of the 8 red acks to the red flow. But for the purpose of congestion control, we apply only 5 of these red acks to the red window, and apply rest 3 to the blue window. This ensures that the sum total of bandwidth acquired by both the flows is equal to the fair share of two TCP flows, thus satisfying the *utilization* property.

In practice, the actual algorithm is much more complex than this, and we formally describe it in Section 2.3.3.

2.3.2 The general case

A source of N TCP flows sharing a bottleneck link is entitled to a total share of the link bandwidth given by the sum of the fair shares of each of the N TCP flows. The source should thus be able to arbitrate this total share among the N flows according to their performance requirements. Each TCP connection opened by an application has a corresponding control loop which uses the AIMD algorithm to adjust its congestion window in response to feedback from the network. The key idea behind our aggregate congestion management scheme is to *keep as many AIMD control loops active in the network as the number of TCP flows in the aggregate*, but to decouple application flows from their congestion windows. We call this scheme MPAT, for Multi-Probe Aggregate TCP, to emphasize the fact that we keep N AIMD control loops (i.e. "probes") active.

Thus, in an aggregate of N flows, the N application data streams are decoupled from the N congestion windows that are each evolving using the AIMD algorithm. The N AIMD control loops allow us to hold the fair share of N TCP flows, while an additional step of mapping application packets to congestion windows allows us to arbitrate the aggregate bandwidth to provide the desired QoS. Since the identity of packets driving a given AIMD loop is irrelevant, this remapping is transparent to the network. Thus, the aggregate appears as N standard TCP flows to the network, providing appropriate fairness to any background TCP or TFRC traffic. Please note that this is different from all the existing schemes[2, 3, 21], which keep only one bandwidth estimation probe active in the network, and hence suffer from problems of scalability and fairness.

The following section describes MPAT in detail.

2.3.3 The MPAT Algorithm

Consider N TCP connections running simultaneously as part of an aggregate, sharing the same bottleneck link in the network. Let these flows be labeled as f_i, $1 \leq i \leq N$. Let C_i represent the congestion window of flow f_i. We introduce another variable A_i which denotes the MPAT window for flow f_i. Let C denote the aggregate congestion window, given by the sum of the congestion windows of all flows. Let x_i denote the fraction of the total bandwidth allocated to flow f_i, such that $\sum_i x_i = 1$. The shares x_i are derived from the performance requirement of the flows f_i, and could change dynamically with time.

Note that C represents the product of the bandwidth available to the aggregate and the round-trip delay (RTT) on the logical overlay link. While TCP would allocate C among the N flows roughly equally in every RTT, MPAT would allocate C in accordance with the performance requirements of the flows. In other words,

$$A_i = x_i * C \tag{2.1}$$

The actual number of packets that flow f_i is allowed to send every RTT is $min(A_i, W_i)$, where W_i is the minimum of the sender and receiver window sizes for flow f_i. With standard TCP, flow f_i would be allowed to send $min(C_i, W_i)$ packets every RTT.

Table 2.3: Symbols and their meanings

C_i	`Congestion Window for flow` i
A_i	`MPAT Window for flow` i
C	$\sum_i C_i$
x_i	`Bandwidth share of flow` i
W_i	`Minimum of sender and receiver window size for flow` i

Each connection i maintains a mapping of the sequence numbers of its packets to the identifier of the congestion window through which the packets were sent. We refer to this mapping as $seqno2id_i()$. This mapping is maintained as a list ordered by increasing sequence number. For each congestion window i, we also maintain the inverse of this mapping, i.e. the list of packet sequence numbers sent through this window. We refer to this mapping as $id2seqno_i()$. This mapping is also maintained as a list ordered by increasing timestamps (i.e., the time at which the packet was sent). We use a variable per congestion window that keeps track of the number of outstanding packets that have been sent through that window.

Transmit Processing

Whenever connection f_i sends out a packet with sequence number s, it tries to find a connection (say j) with open congestion window. Congestion window j is said to be open if the number of outstanding packets sent through it is less than C_j. Note that the packets sent using congestion window j could belong to f_j, or to any other flow in the aggregate. In practice, j could be the same as i. Connection i then stores the mapping of seqno s to congestion window j in $seqno2id_i()$. We also store the inverse mapping in $id2seqno_j()$

Receive Processing

Our implementation is based on TCP-SACK. When we receive an acknowledgement for a packet with sequence number s, belonging to flow f_i, we use the mapping $seqno2id_i()$ to find the congestion window j through which the packet was sent. We then look at the inverse mapping $id2seqno_j()$ to find if the ack received is in sequence or not. We then apply the standard AIMD algorithm to congestion window j. Thus, if the ack is in sequence, we linearly increase C_j to $C_j + 1/C_j$. If it is a duplicate ack (in case of SACK, this would be the ack for a later packet in sequence), we enter the fast retransmit phase, halving C_j if we get three duplicate acks (in case of SACK, three out-of-sequence acks).

Mapping sequence numbers to congestion windows

The choice of the mapping of a sequence number to a congestion window is critical, since it affects the number of fast retransmit-induced window halvings that an MPAT aggregate receives over a given interval of time, and hence the total bandwidth gained by it. Recall that standard TCP-SACK considers multiple losses within a single congestion window as one loss event for fast retransmit, and hence cuts down its window by half only once. To understand how this affects MPAT, consider the following example: There are two TCP flows running simultaneously as part of an MPAT aggregate. Connection 1 has a window size of 20, and has packets 1 through 20 outstanding in the network. Suppose packets 8 and 9 get dropped. Since both these packets belong to the same window, standard TCP-SACK would treat them as one signal for fast retransmit, and hence cut down its window only once.

MPAT could treat this situation as either one or two fast retransmit signals, depending upon the mapping $seqno2id_1()$. Consider the case when sequence numbers 8, 10-12 are mapped to C_1, and sequence numbers 9, 13-15 are mapped to C_2. The acks for packets 10-12 will be charged to C_1, which will treat all three of them as duplicate acks for packet 8, and hence cut down its window by half. But the acks for packets 13-15 will be charged to C_2, which will again treat them as duplicate acks for packet 9, and hence cut down its window by half. This example shows how two packet losses within the same window could lead to two fast retransmit signals. On the other hand, if both packets 8 and 9 were mapped to C_1, this would lead to only one fast retransmit signal. In this case, flow 2 would treat acks for packets 13-15 as if they were received in sequence, and hence would *not* reduce its window.

We tried out three different algorithms to map the packet with sequence number s, belonging to flow f_i, to a congestion window :

- If sequence number $s - 1$ was sent through congestion window k, send the current packet s through window k if it is open. Else, randomly pick an open congestion window to send the packet. This algorithm turns out to be aggressive to the background flows since consecutive packet losses are charged to the same congestion window, and get absorbed into one loss event for fast retransmit, reducing the number of window reductions.

- If sequence number $s - 1$ was sent through congestion window k, randomly choose an open congestion window different from k. This scheme turns out to be more conservative than standard TCP, since it forcibly maps multiple losses on a flow to different congestion windows, causing each one of them to be cut down.

- Map s to an open congestion window picked uniformly at random. This option turns out to be a reasonable compromise between the two extremes. In fact, it turns out to be conservative when one of the flows has a very large share of the aggregate congestion window, since it still spreads out the losses of packets on that flow to multiple congestion windows. We use this option in our implementation.

Whenever any of the congestion windows in the aggregate changes, the aggregate congestion window also changes, and hence we must also update the MPAT windows A_i of all connections using equation 2.1. Note that this means that whenever the congestion window for one of the flows is cut down, the loss in available bandwidth gets distributed proportionally among the MPAT windows of all the flows. This has the effect of reducing the inter-connection variance in throughput, as we shall see later in section 2.4.6.

TFRC variant

There exists a TFRC[23] variant of MPAT, in which instead of N AIMD control loops, we have N rate adjustment loops, each driven by the standard TCP throughput equation for one TCP flow. Note that loss probabilities will be measured by congestion window, i.e. over data sent using a given rate adjustment loop, irrespective of which application flow the packets came from. Loss events will be mapped to congestion windows in the same way as described above. The TFRC variant enjoys the same transparency properties as the AIMD version, in that each rate adjustment loop is oblivious to the identity of the packets driving it. As a result, the TFRC variant of MPAT appears as a set of N independent TFRC flows to the network. All of our experiments in this chapter are with the AIMD version of MPAT.

2.3.4 Implementation

We have prototyped our proposed aggregate congestion management scheme in Daytona[28], a user-level TCP stack running on Linux. Daytona can be linked as a library into an application, and offers the same functions as that provided by the UNIX socket library. Daytona talks to the network using a raw IP socket, thus avoiding any processing in the operating system's TCP stack. All TCP processing is done at user-level, and pre-formed TCP packets are handed to the IP layer to be sent over a raw socket.

Our user-level implementation allowed us to run wide-area experiments on diverse locations in the Internet, as well as in controlled public network testbeds like Emulab [56], without requiring control over the operating system.

A key goal in MPAT is to ensure that we hold the fair share of N TCP flows in the process. Thus, each AIMD control loop should be kept running as long as there are packets available to drive it. Note that if a flow with a high share of the aggregate congestion window gets send or receive window limited, or has a period of inactivity, then it may leave some of its share of the aggregate congestion window unused. Left alone, the aggregate would lose this unused share to the background traffic. By using a work-conserving scheduler on the aggregate, however, we make sure that other flows in the aggregate take up such unused bandwidth, if they have data to send.

An implementation detail concerns the amount of memory allocated per socket corresponding to a connection. Due to an unequal apportionment of the total bandwidth among different connections, each connection needs a different amount of buffer memory at any given instant of time. Currently our implementation statically partitions a large chunk of memory between connections in accordance with their performance requirements. In the future, our implementation would dynamically allocate socket memory, as proposed in [20].

2.4 Evaluation

In this section, we conduct extensive experimentation to evaluate the fairness, utilization and scalability properties of MPAT. We have conducted experiments both over the *real Internet* and in controlled settings under stable conditions, reverse traffic, and transient congestion. We built our own wide-area network testbed consisting of a network of nodes in diverse locations (including US, Europe and Asia). Our main goal in choosing these nodes is to test our system across wide-area links which we believe have losses. For this reason, we made sure that the whole path is not connected by Internet2 links (known to have very few losses). We changed the routing tables of our campus network to make sure that the path to our destination did not take any of the Internet2 links. We also did experiments on the Emulab network [56], which provides a more controlled setting for comparing our scheme with other approaches. All of our experiments used unconstrained send and receive windows on both ends of the TCP connections, so that flow control does not kick-in before congestion control. As background traffic we use long lived TCP connections and bursty web traffic. We had a Maximum Segment Size (MSS) of 1460 bytes in all our experiments.

2.4.1 Scalability

Using experiments on both the Emulab network and the real Internet, we found that the MPAT scheme can hold the fair share of 100 TCP flows running simultaneously as part of an aggregate, irrespective of how the aggregate bandwidth was apportioned among the aggregate's flows. The limits on scalability beyond 100 flows arise due to other factors like minimum amount of bandwidth needed per connection, bottleneck shifting from network to memory, etc.

As noted in section 2.2, mulTCP scales only up to 20-25 flows in the Emulab setting. In a real Internet experiment, we found mulTCP to scale upto 10 to 15 flows.

2.4.2 Performance differentiation

The MPAT scheme can be used to apportion the total available bandwidth among different flows in any desired ratio. Even though the total fair share allocated to the aggregate keeps changing with time, MPAT can maintain the target performance differential at all times. We could give a maximum of 95 times more bandwidth to one TCP flow over another.

The real limits to scalabiliy for large performance differential arise from the fact that every connection needs to send a minimum of 1-2 packets every RTT [37]. In the future, we plan to change this to 1 packet every k RTTs, as proposed in [26].

As an example, Figure 2.7 shows the bandwidth allocated to five TCP flows running as part of an MPAT aggregate between Utah and our campus network on the east coast (RTT approximately 70 msec). There were five more long-running standard TCP flows in the background (in both directions). We tried to split bandwidth in the ratio 1:2:3:4:5. The graph is for a period of 5 min, with data samples taken every 500msec (approx 7 RTTs). The average amount of bandwidth that each of the five flows got was 135 KBps, 265 KBps, 395 KBps, 530 KBps and 640 KBps respectively. We can see that the MPAT scheme is very effective at maintaining the desired target differential among different flows at all times, irrespective of the total share of bandwidth that the aggregate is entitled to. The average bandwidth of a standard TCP flow running in background during this time period was 400 KBps. During the process of splitting bandwidth, the MPAT aggregate holds its total fair share of approximately 2000 KBps, as seen in Figure 2.8, thus satisfying the *utilization* criterion.

2.4.3 Fairness to background flows

As noted earlier, an MPAT aggregate is naturally fair to background TCP traffic since the remapping of application packets to congestion windows is transparent to the N AIMD control loops in the aggregate, which means that the network effectively sees N standard TCP flows. Under the same experimental conditions

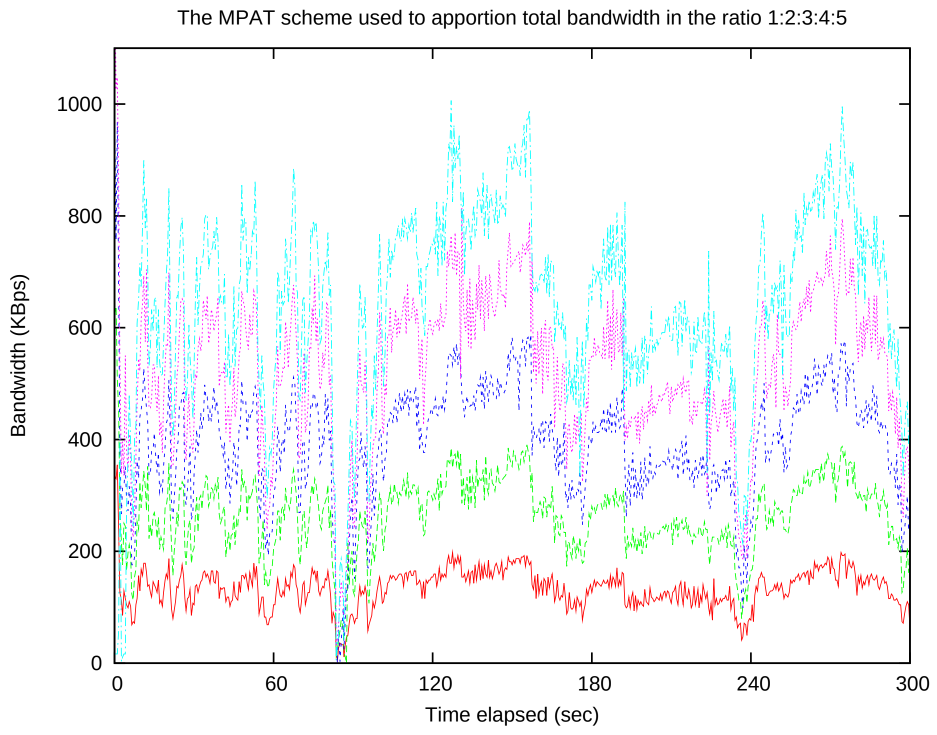


Figure 2.7: MPAT can apportion bandwidth among its flows, irrespective of the total bandwidth available.

as above, Figure 2.12 shows the ratio of total bandwidth occupied by the MPAT aggregate ($N = 16$) as compared to the total bandwidth of 10 standard TCP flows running in background.

2.4.4 Interaction between multiple MPAT flows

To analyse the behavior of multiple competing virtual overlay links using MPAT, we study the bandwidth achieved by multiple MPAT aggregates running simultaneously. Again, as noted earlier, the behavior of an MPAT aggregate with N flows is similar to N independent TCP flows running without any aggregation. This suggests that each MPAT aggregate should be able to get a share of the bottleneck bandwidth depending upon the number of flows in each aggregate, and the number

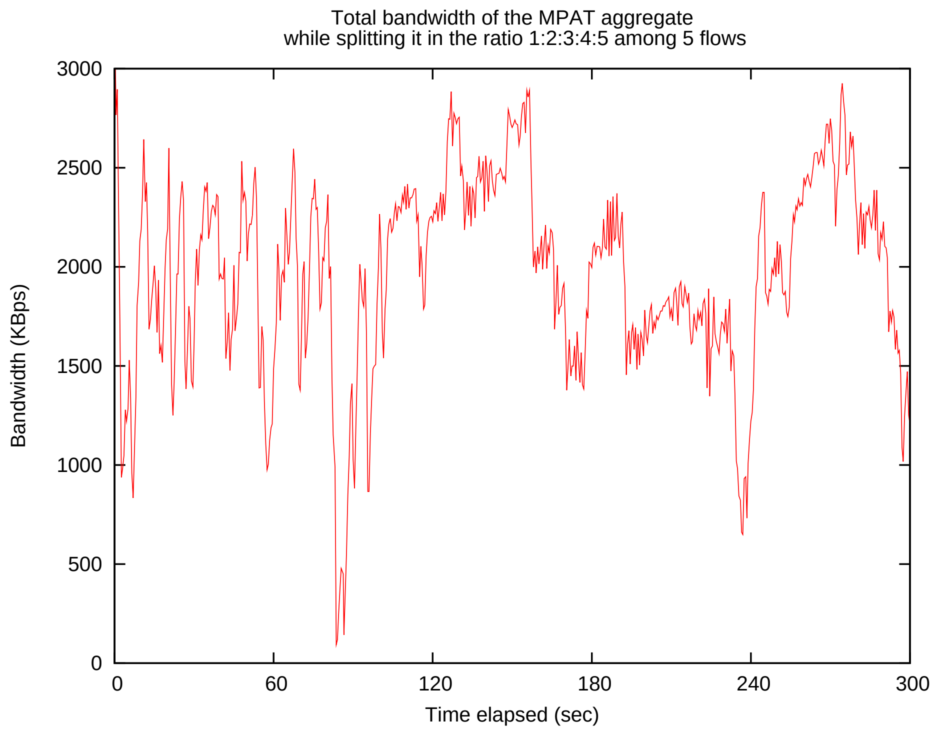


Figure 2.8: MPAT can hold the total fair share while apportioning bandwidth in the desired ratio.

of background TCP flows.

To verify this hypothesis, we ran 5 MPAT aggregates, each with a different value of N, together with 5 background TCP flows between Utah and our campus network on the east coast. The five MPAT aggregates use N = 2, 4, 6, 8 and 10. We then replaced each aggregate with the same number of independent TCP flows. The experiment was repeated during different times of day. Figure 2.9 shows the bandwidth for each of the five MPAT aggregates, with data points sampled every 1sec. Tables 2.4(a) and 2.5(a) show the number of fast retransmits, (labelled # *Fast*), number of timeouts and bandwidth for each flow within the MPAT aggregate over a period of 390 seconds. Tables 2.4(b) and 2.5(b) show the corresponding data for the non-aggregated case. Note that in both cases, the achieved bandwidth and loss rates seen by the background TCP flows are similar. The bandwidth obtained by each aggregate is always proportional to the number of flows it had, and when the flows within an aggregate were run independently, the sum of bandwidths achieved by these flows was similar to the bandwidth achieved by the aggregate. This shows that MPAT is not a selfish scheme. Competing MPAT flows cooperate with each other, and with the background traffic. The experiment also illustrates that MPAT adequately satisifies the utilization property.

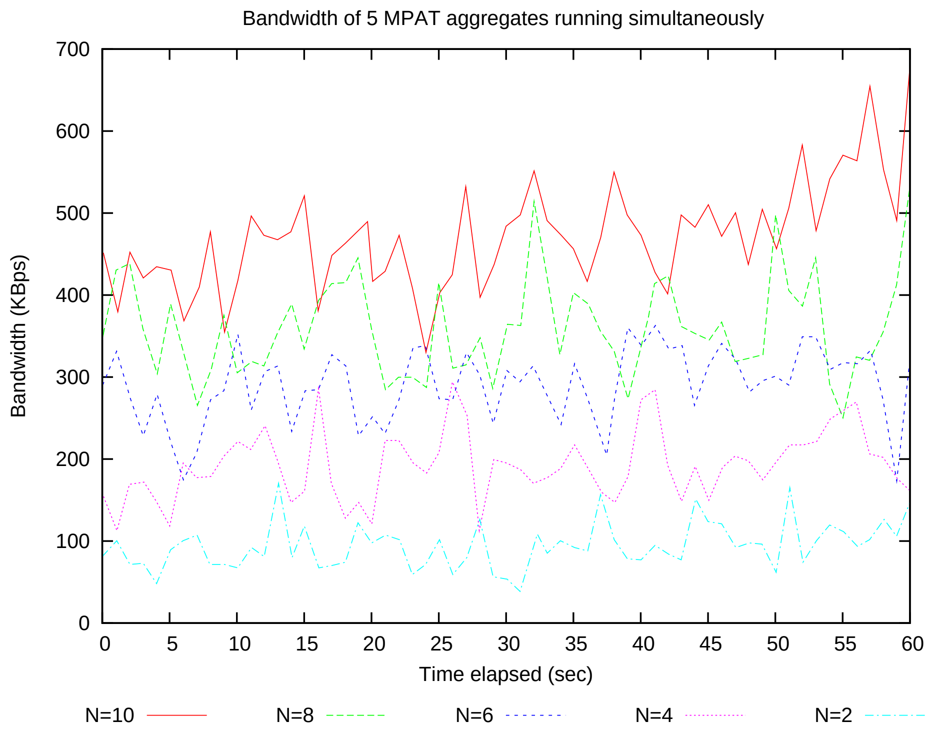


Figure 2.9: Competing MPAT aggregates cooperate with each other.

2.4.5 Adaptation to changing performance requirements

When the performance requirements of various flows within an aggregate change, our scheme simply needs to update the MPAT windows A_i for all flows using Equation 2.1, and thus should be able to respond very quickly (typically 2-3 RTTs)

Table 2.4: Multiple MPAT aggregates running simultaneously cooperate with each other (day time).

(a) With Aggregation (day time)

N	# Fast	# Timeouts	Bandwidth (KBps)
2 (MPAT)	511	250	51.4
4 (MPAT)	474	231	49.5
6 (MPAT)	499	235	50.3
8 (MPAT)	495	235	50.1
10 (MPAT)	502	236	49.8
5 (TCP)	401	258	48.6

(b) Without Aggregation (day time)

N	# Fast	# Timeouts	Bandwidth (KBps)
2(TCP)	498	218	56.1
4(TCP)	479	234	58.5
6(TCP)	481	215	58.9
8(TCP)	481	216	57.1
10(TCP)	490	218	57.2
5(TCP)	497	213	56.3

Table 2.5: Multiple MPAT aggregates running simultaneously cooperate with each other (night-time).

(a) With Aggregation (night time)

N	# Fast	# Timeouts	Bandwidth (KBps)
2 (MPAT)	565	44	97.6
4 (MPAT)	564	34	98.5
6 (MPAT)	555	39	98.1
8 (MPAT)	535	38	99.3
10 (MPAT)	537	41	97.9
5 (TCP)	577	41	93.6

(b) Without Aggregation (night time)

N	# Fast	# Timeouts	Bandwidth (KBps)
2(TCP)	540	28	106.7
4(TCP)	542	25	110.9
6(TCP)	546	27	108.5
8(TCP)	539	25	106.4
10(TCP)	531	26	109.5
5(TCP)	538	30	107.1

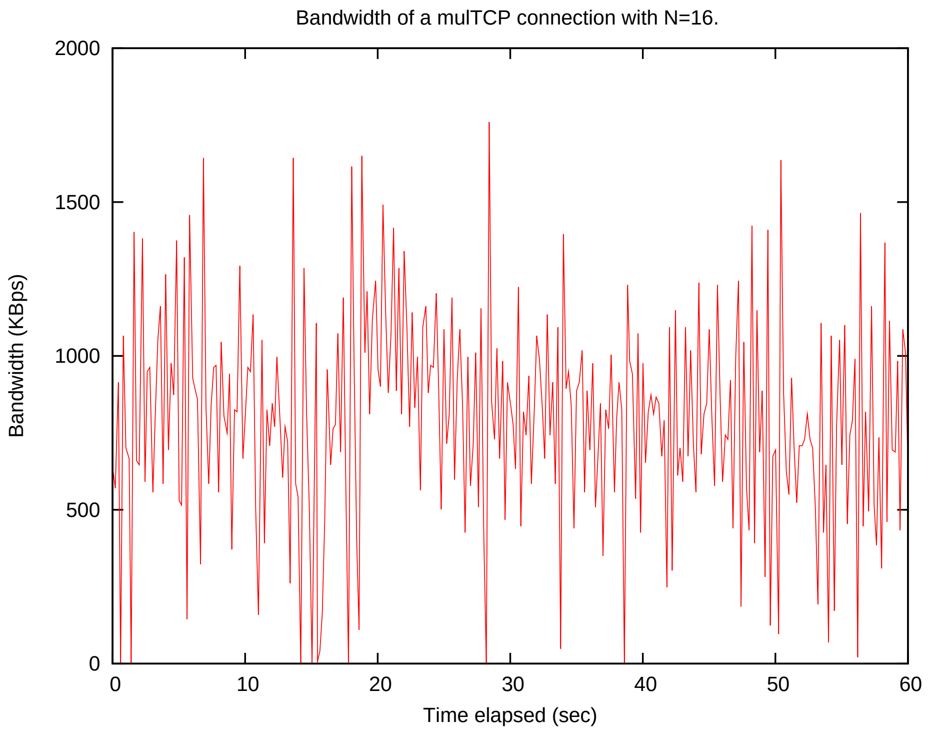


Figure 2.10: mulTCP scheme exhibits very high variance in bandwidth.

to dynamically changing performance requirements with time.

To test this claim, we ran an MPAT aggregate consisting of three flows between Utah and our campus network on the east coast, with 5 more long-running standard TCP flows in the background (in both directions). We changed the performance requirements every minute. Figure 2.13 shows how the absolute bandwidth for each of the three connections varies with time for a period of nine minutes. Data points have been sampled every 500 msec (approx 7 RTTs). Figure 2.14 shows relative bandwidth of the three flows at all times, along with the target performance differential. We can see that MPAT very quickly adapts to changing performance requirements. Note that during the time interval $t = 240$ to $t = 300$ sec, we were able to split bandwidth in the ratio 95:1.

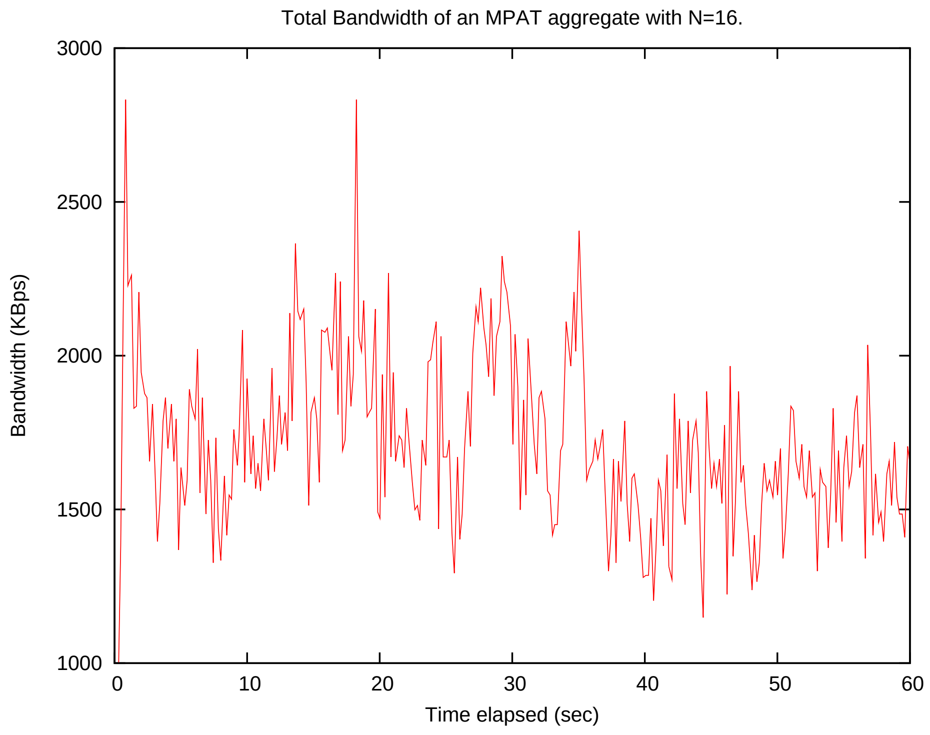


Figure 2.11: MPAT exhibits much lower variance due to sharing of losses.

2.4.6 Reduced variance

The total bandwidth of an MPAT aggregate is equivalent to the sum of N independent standard TCP flows, each of which uses an AIMD control loop. Multiplexing of these control loops smooths out the bandwidth variations within each AIMD loop, thus reducing the variance in throughput of an MPAT aggregate drastically. This is in contrast to mulTCP that exhibits increasing variance with N.

To demonstrate this, we ran an MPAT aggregate consisting of 16 flows between Utah and our campus network on the east coast. We had 10 long-running standard TCP flows running in the background. Figure 2.11 shows how the total bandwidth occupied by the MPAT aggregate varies with time. Figure 2.10 shows the bandwidth of a mulTCP flow (with $N = 16$) under similar conditions. Data points have been sampled every 200msec (3 RTTs).

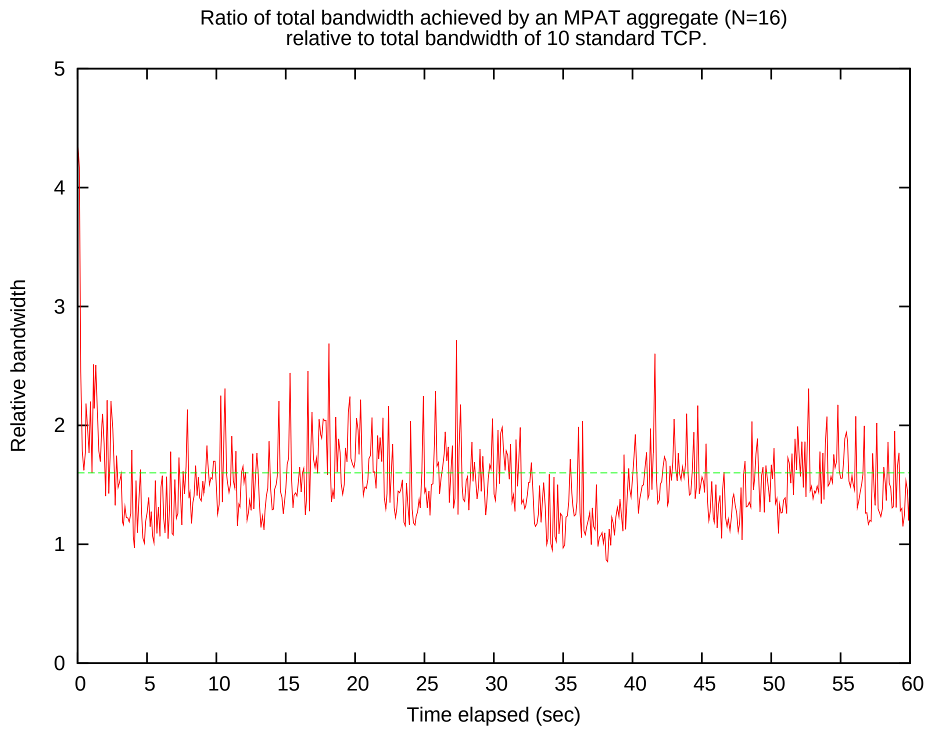


Figure 2.12: MPAT is friendly to the background traffic.

Comparing Figure 2.10 and Figure 2.11, we see that mulTCP has a very high variance as compared to that of MPAT. Please note that the y-axis in Figure 2.11 is shown for the range 1000-3000 KBps, while that in Figure 2.10 is for the range 0-2000 KBps. This is because a mulTCP connection with $N = 16$ gets much less throughput than the fair share of 16 TCP flows.

2.4.7 Adaptation to Transient Congestion

The MPAT scheme must be robust to transient congestion in the network, and more generally, to variations in the fair share of bandwidth available to the aggregate. With transient congestion, an MPAT aggregate must learn about a change in its total fair share using the AIMD algorithm. The change is apportioned among the aggregate's flows quickly through an adjustment of the MPAT windows A_i of all

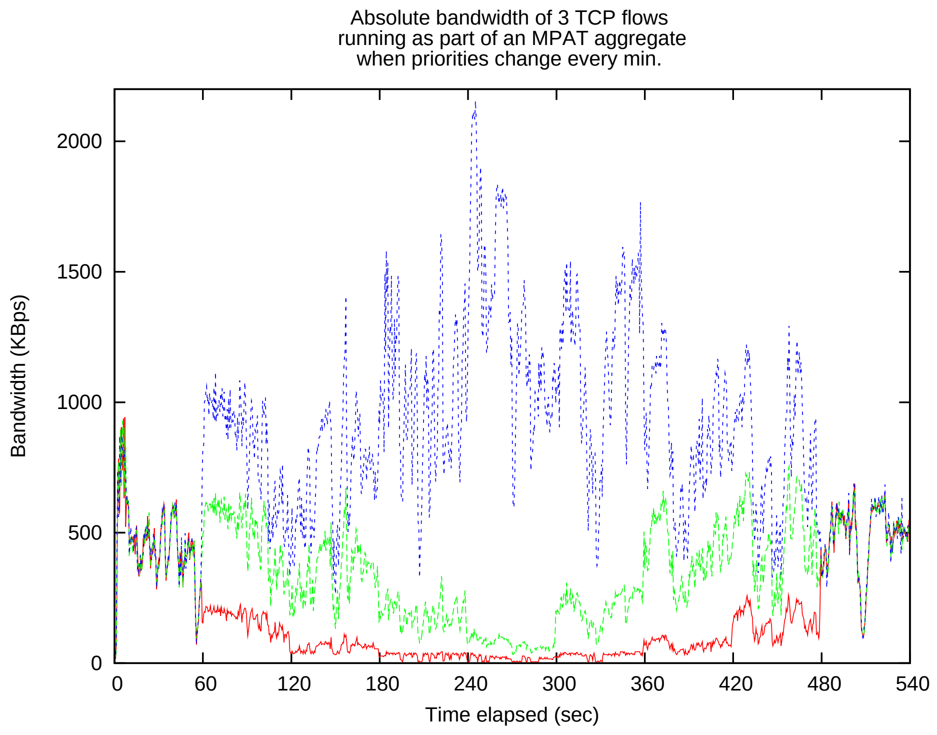


Figure 2.13: This figure shows the how absolute bandwidth of 3 MPAT flows adapts itself very quickly to dynamically changing performance requirements.

flows, ensuring that the desired bandwidth ratio is maintained at all times.

To study the responsiveness of MPAT to transient congestion, we conducted experiments on the Emulab network over an emulated link of bandwidth 90 Mbps and RTT 50msec. We ran an MPAT aggregate consisting of 10 flows with different performance requirements. Transient congestion was created by introducing constant bit rate (CBR) as well as TCP traffic in the background. We also used 10 long-running standard TCP flows in both directions. At t=120 sec, we introduce a CBR flow in the background with rate 30Mbps. As shown in Figure 2.16, MPAT cuts down its total bandwidth in about 4-6 RTTs (200-300 msec), while still apportioning bandwidth within the aggregate in the desired ratio at all times. At t=240sec, we removed the background CBR traffic. As seen in figure 2.17, MPAT

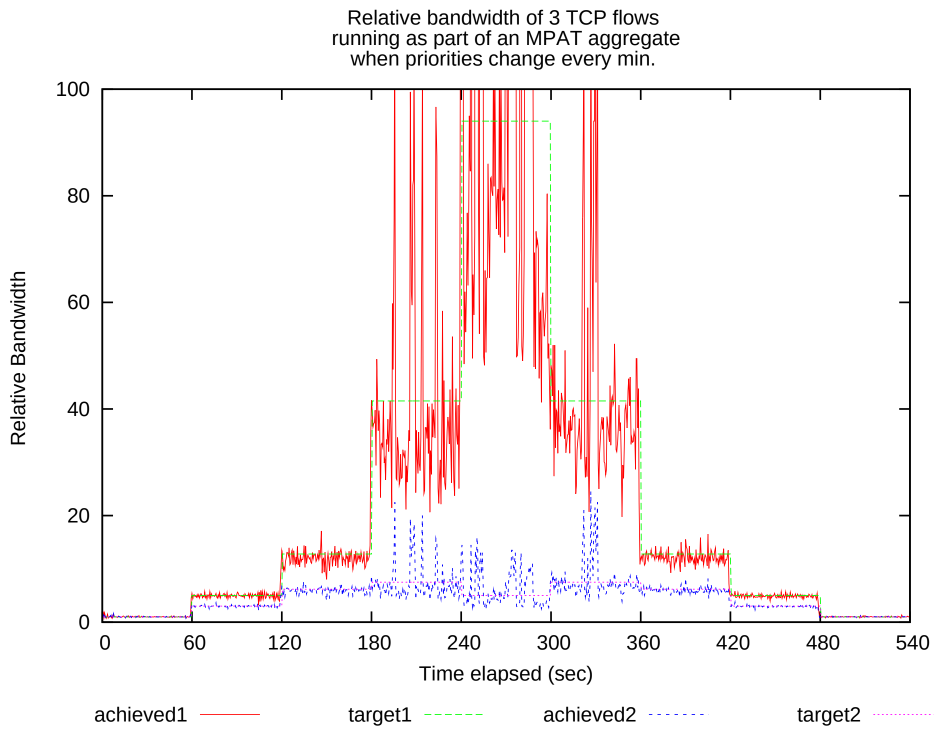


Figure 2.14: This figure shows the how relative bandwidth of 3 MPAT flows adapts itself very quickly to dynamically changing performance requirements.

reclaims its fair share of the remaining bandwidth.

When background TCP traffic was introduced to create transient congestion, it takes about 15-20 (1 sec) RTTs for both MPAT and the background TCP flows to settle into their respective fair shares, as seen in figures 2.19 and 2.20. This is because the new TCP flows begin in slow-start mode, and are also adapting to their fair share together with MPAT. As earlier, MPAT always apportioned bandwidth within the aggregate in the desired proportion.

2.4.8 Bursty background traffic

Most of the results we described above are for long-running TCP flows in the background. We also tested our scheme with bursty web traffic running in background.

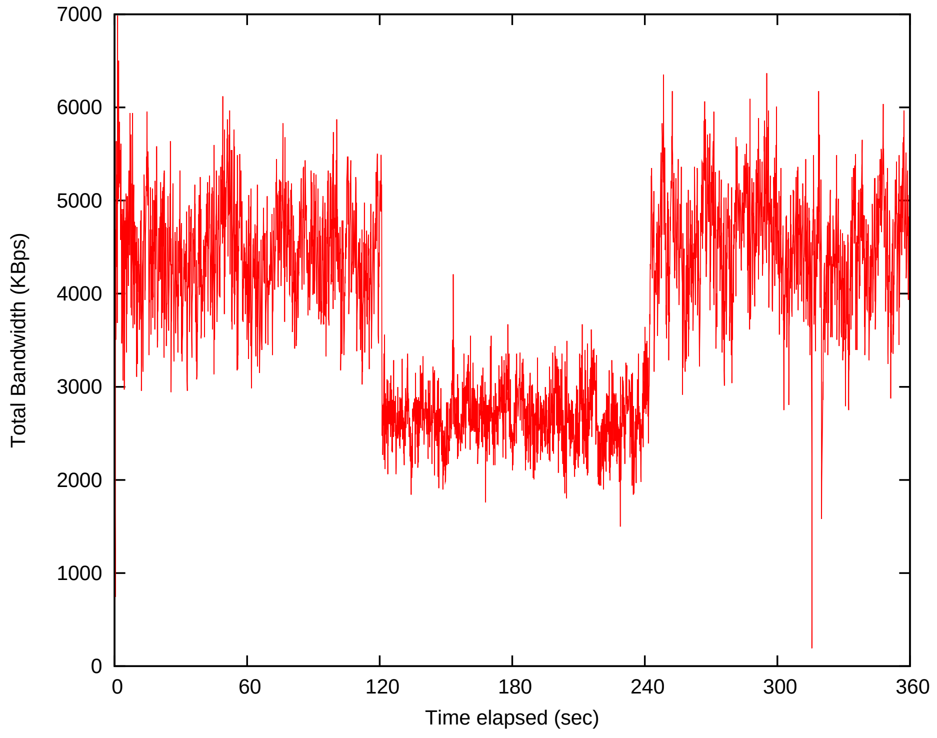


Figure 2.15: This figure shows how the total bandwidth achieved by an MPAT aggregate (N=16) adapts itself with additional **UDP** traffic introduced at t=120 and then removed at t=240.

We did this using a *wget* loop downloading web pages in the background, together with MPAT. While bursty traffic increased the absolute number of retransmits and timeouts *proportionally* for all aggregates and the background traffic, we did not see any qualitative change in the results. In other words, MPAT exhibited the same scaling, fairness and utilization properties with bursty traffic.

2.5 Conclusions

This chapter demonstrates for the first time the viability of providing differential services, at large scale, among a group of TCP flows that share the same bottleneck link. We demonstrate this through a range of experiments on the real

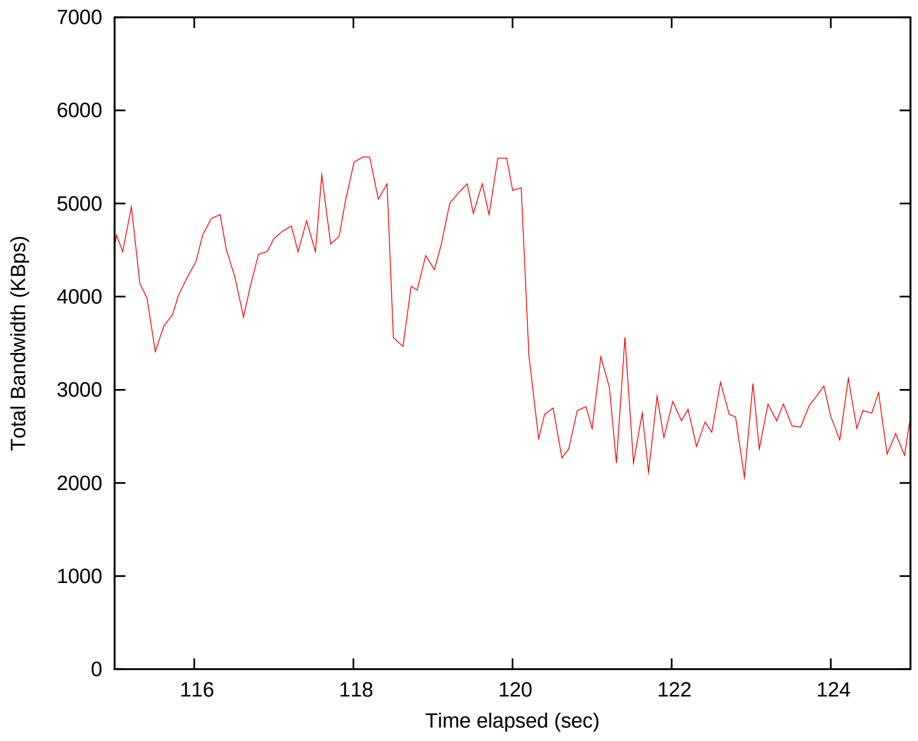


Figure 2.16: A deeper look into Figure 2.15 at t=120 to see the effect of introducing additonal **UDP** traffic.

Internet which show that an MPAT aggregate can hold its fair share of the bottleneck bandwidth while treating other flows fairly (either individual TCP flows or other MPAT aggregates). We also demonstrate that within an aggregate, MPAT allows substantial differentiation between flows (up to 95:1). This suggests that MPAT is therefore an appropriate candidate technology for broader overlay QoS services, both because MPAT provides the requisite differentiation and scalability, and because MPAT can co-exist with other TCP flows and with other MPAT aggregates. This result opens the door to experimentation with more easily deployable network QoS schemes such as the overlay.

Having said that, this chapter does not make the broader conclusion that overlay QoS works. In order to do that, we must understand the end-to-end behavior

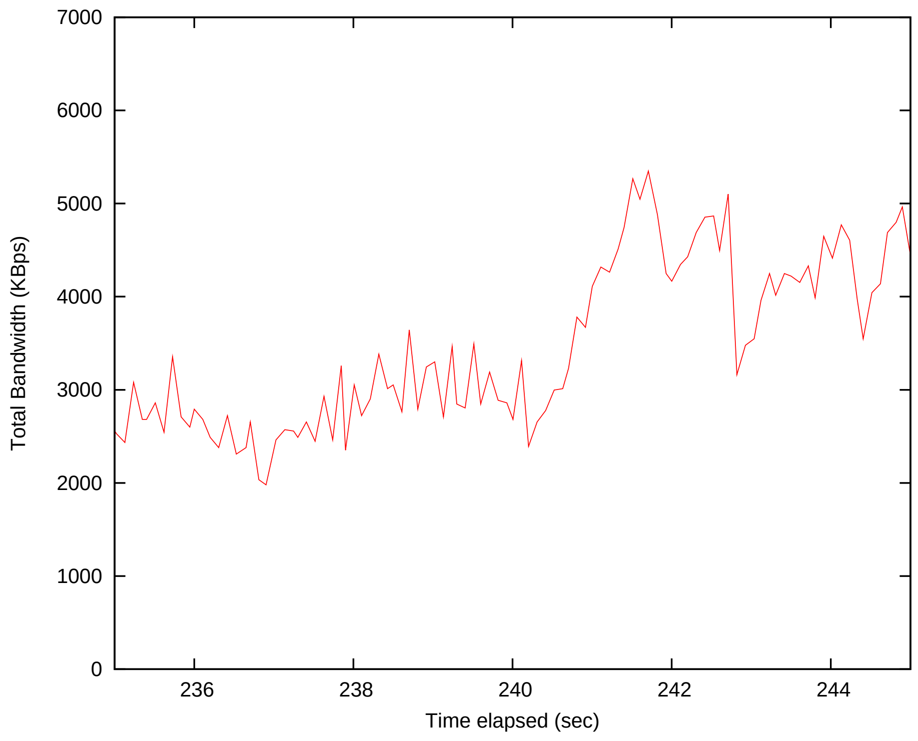


Figure 2.17: A deeper look into Figure 2.15 at t=240 to see the effect of removing additonal **UDP** traffic.

of flows (TCP and otherwise) over a multihop overlay. This means among other things that we must understand the interactions between the individual overlay hops (be they TCP or TFRC) of the same overlay, between aggregates on different overlays, and between all of these and the end-to-end flow control of the TCP connections running over the overlay. It is also important to understand how an overlay QoS could be combined with RON-type overlay functionality used to actually enhance performance (not just provide differentiation).

There may also be other uses for MPAT aggregation. For instance, in the back end systems of a web service data center, MPAT could be used to provide differential service among different kinds of customers (gold, silver, bronze). This is possible in part because MPAT only needs to be deployed at the sending end of a

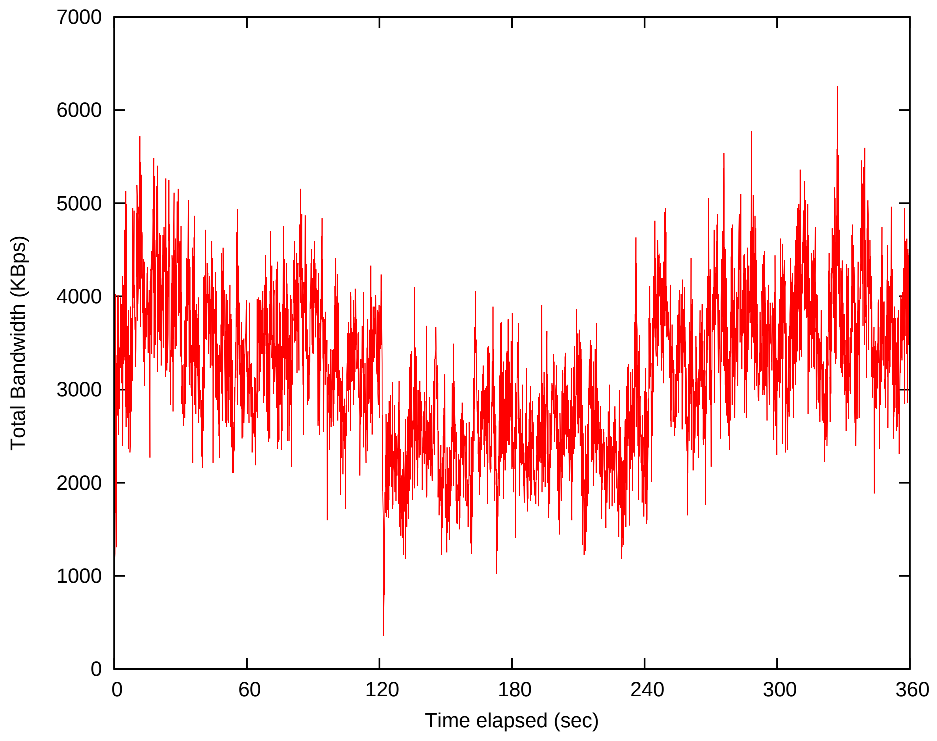


Figure 2.18: This figure shows how the total bandwidth achieved by an MPAT aggregate (N=16) adapts itself with additional **TCP** traffic introduced at t=120 and then removed at t=240.

TCP connection. For this to work, however, the TCP flows must share a bottleneck. This in turns requires that the server implementing MPAT can determine which flows, if any, share the bottleneck. These possible uses of MPAT provide exciting avenues for future research.

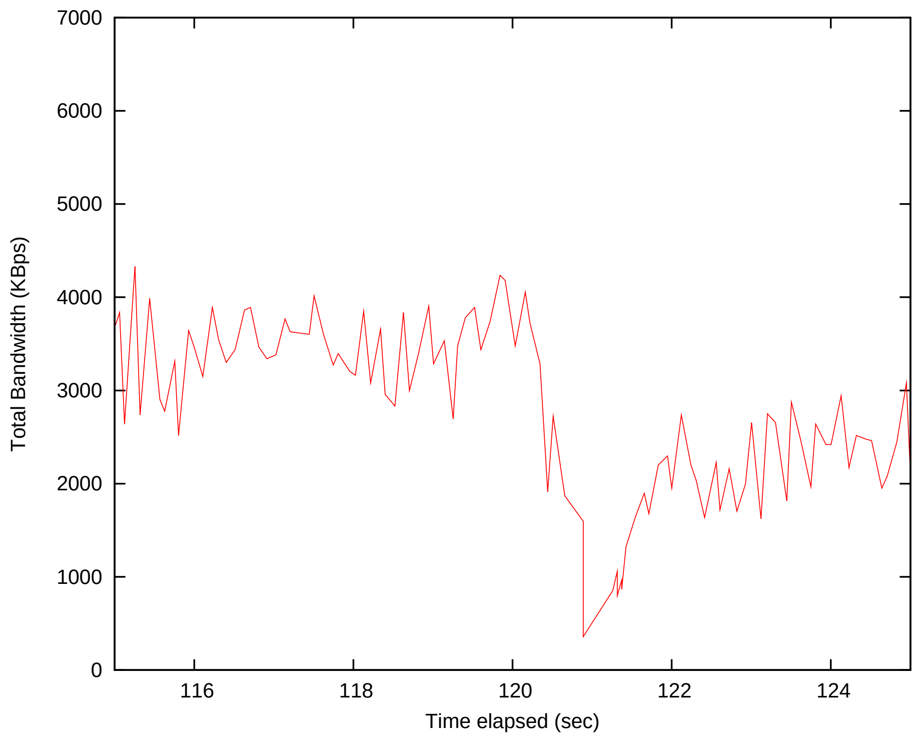


Figure 2.19: A deeper look into Figure 2.18 at t=120 to see the effect of introducing additonal **TCP** traffic.

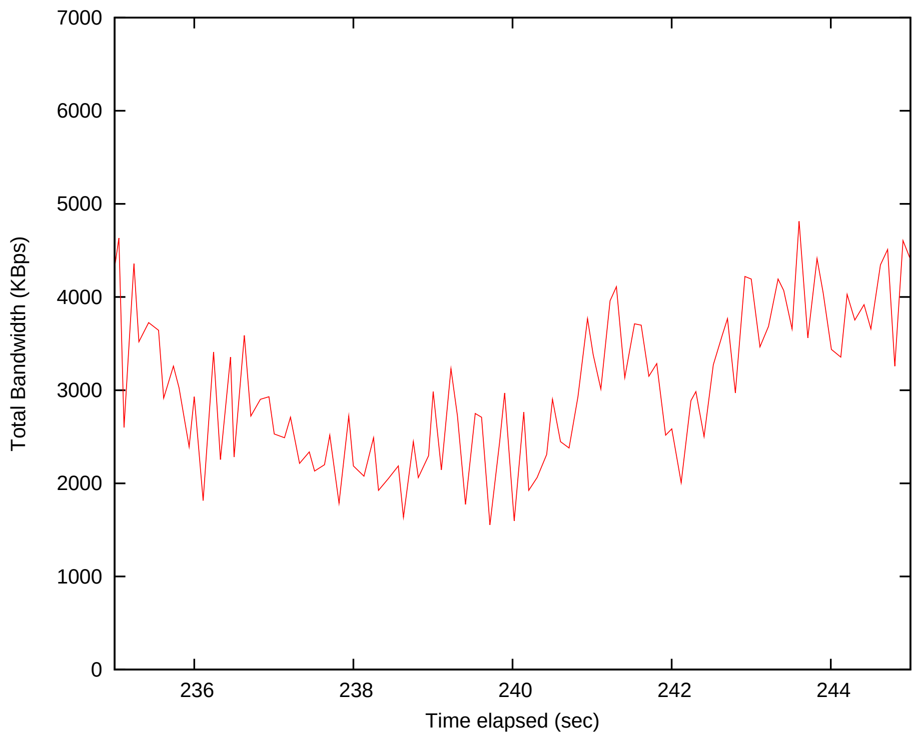


Figure 2.20: A deeper look into Figure 2.18 at t=120 to see the effect of introducing additonal **TCP** traffic.

Chapter 3

Netmapper: A network mapping and annotation service for distributed applications

3.1 Monitoring the network

Network performance is a key determinant of end-to-end performance and user experience for many applications [1, 2, 25, 38, 53]. However, managing network performance is fundamentally hard due to the distributed and decentralized nature of network resources. In spite of decades long research, providing better network performance to applications still draws considerable attention. The various proposed solutions range from ATM to RSVP to Intserv to Diffserv and many others [13, 14, 15, 16]. While these have not resulted in a completely satisfactory solution, we have learned that end-to-end approaches are generally more realistic and deployable compared to those that rely on explicit network support [40].

This chapter describes a network mapping and annotation service for distributed applications sensitive to bandwidth availability, latency, or loss. Applications (or resource managers working on their behalf) are expected to use this service to plan their network resource usage and fault response. We expose internal network state (e.g. bandwidth available on the edges, location of bottleneck links, etc) using only end-to-end measurements that are TCP-friendly. Knowledge of this internal state allows trend analysis, network performance debugging, network planning and exploitation of alternate routes, making such a service useful for both network providers and their ISP and enterprise customers.

The system (called Netmapper) takes as input the set of end-points of a distributed application, maps out the network connectivity between them, and annotates each edge (to the extent possible) with available bandwidth. Netmapper deploys end-to-end TCP probes between the end-points of an application to estimate the available bandwidth on various network paths. It then identifies the bottleneck links on these paths using a novel TCP-friendly bottleneck identification technique. Netmapper eventually annotates all the edges whose annotation is possible given the network topology and the available end-points. The end-to-end probes are intelligently planned so that the annotation requires a minimum number of probes.

Stability of network performance metrics over reasonable lengths of time [43] leads us to believe that Netmapper can provide a stable map of network performance that can be updated on a coarse time scale. We have built Netmapper, and tested it out on the real Internet. Our experience deploying Netmapper on the real Internet shows that the system consumes a very small amount of bandwidth, and converges fast even with dynamically varying network state. Both variability in available bandwidth and shift in location of bottleneck link are slow enough that they allow our system to converge, and maintain a consistent graph.

The rest of the paper is organized as follows: We discuss some of the related work and motivation for building a network monitoring service in Section 3.2. We summarize the primary contributions of our monitoring system in Section 3.3. We describe various algorithms to annotate a network graph with available bandwidth in Section 3.4. Section 3.5 presents a detailed performance evaluation of Netmapper using experiments over the real Internet. We conclude in section 3.6 by describing a set of applications that are using our system.

3.2 Motivation and related work

Many other research efforts have attempted to provide network monitoring services to applications [5, 38, 40, 51, 55], that they can query and take appropriate action. However, these efforts have not closed the loop between monitoring and using the collected data to set end-to-end performance control knobs, expecting the sophistication to do so from the application. It is simply too complex, and in some cases not possible for applications to plan and exercise these controls by themselves. Exercising end-to-end controls for performance management requires an application to understand the underlying network structure, monitor it, and map its performance requirements to the setting of an end-to-end control knob such as MPAT [**?**], multi-homing [60], overlay routing [5], or server selection [59]. There is a clear motivation for abstracting away the complexity of network monitoring and planning from the application into a service. Such a service can then also act as the bridge between applications and network providers by providing application traffic demand matrices to the network provider.

Several applications, such as server selection (Akamai [59], IBM Olympic hosting, etc.), multi-homing based route selection [60], and overlay network construction (RON [5], Narada [38], etc.) employ some form of end-to-end mechanisms to provide better network service. However there are two main issues with these approaches.

First, typically these end-to-end approaches treat the network as a black-box and provide purely end-to-end metrics such as delay, loss, bandwidth estimates agnostic of any network details. The major problem with this extreme black box solution is that by completely ignoring any network path or topology information, the underlying path sharing among different choices, common bottleneck points, or any topologically correlated properties are hidden from any planning or provisioning of the network. Note that sharing is critical for bandwidth planning. Providing a latency map of the network [51, 55] is not adequate for packing an application's bandwidth demand into the network. Bandwidth annotation is generally used to express capacity constraints and latency annotation is used to express performance constraints. Sharing also allows more efficient fault response compared to topology-unaware monitoring of liveness between the application's endpoints [38]. Overlay network studies also mostly treat the path between any two nodes as a single point-to-point link. Any resource provisioning in such a scenario could lead to bad network provisioning as many such paths may be sharing the same bottleneck link.

Second, these approaches tightly couple the end-to-end measurement solutions with the application/network provisioning decisions. As a result, it is often impossible to utilize these solutions for any other applications, reducing their approaches as point solutions. While the motivation for these studies is to better manage the network from an end-to-end perspective, often these solutions fall short of becoming a truly generic solution that application providers could use.

Some systems like Remos[40] require considerable support from within the network using SNMP data from routers. However, such tools can only be used by network operators (e.g., telecommunication companies) and/or on the subparts of the network owned by them. Thus, provider customers cannot easily make use of these approaches.

3.3 Our contributions

Netmapper addresses many limitations of existing monitoring solutions [38, 40, 5, 51, 55]. We briefly summarize below the contributions of our network mapping and annotation service.

1. Netmapper gathers all the information *without any additional support from the network* infrastructure itself. By conducting experiments between agents that run on the end points, our system can derive internal information about the network (e.g., internal link, available bandwidth at a link, locations of bottlenecks, etc.) using end-to-end measurements. This is useful because the provider of the service or application generally has no control over the network and probably has limited or no access to monitoring data of the network. For example, ISPs (such as AT&T, Sprint, etc.) typically have network management systems that internally monitor the performance of the network. However, the ISPs only have monitoring data for the parts of the network they own. Moreover, this information is generally not available to the applications running at the network endpoints.

2. Netmapper *explicitly represents network topology to capture sharing* properties. This makes it capable of supporting available bandwidth queries on the network for simultaneous flows. For example, the system can answer advanced queries like "determine the available bandwidth when node 1 sends to node 4 AND node 2 sends to node 10" which may not be correctly answered by a system that does not consider the annotated network topology. Note that sharing of paths can imply different answers for individual path queries and simultaneous multi-path query.

3. Using purely end-to-end measurements, Netmapper *provides bandwidth annotation of links inside the network*, which is an inherently more difficult problem compared to latency annotation. By determining such information, decisions can be made regarding routing a client to a particular server (e.g., which network paths to take and/or servers to which the client should be directed, etc.) that has the best network connectivity, the most bandwidth, etc.

4. We design a novel scheme to *identify the location of bottlenecks in a network* for a distributed networked application. The tool improves upon existing methods [53, 39] for bottleneck identification. Specific contributions over existing schemes include identifying bottlenecks with little to no impact on the regular traffic, and matching TCP's notion of bottleneck (as opposed to, say link-level definitions). We show using experiments (under controlled settings of the Emulab environment [56]) that existing bottleneck identification schemes [53, 39] detect incorrect bottleneck link if the bottleneck is due to transient congestion. This tool is described in detail in Chapter 4.

5. An advantage of Netmapper is that it is *not intrusive* in comparison to some of the existing approaches [53] that use UDP-based probes to determine the network capacity. The annotation process does not change the dynamics of the existing flows because the procedure used is based on TCP probes.

6. We use bottleneck identification as a fundamental primitive to *reduce the total amount of probe traffic* that we need to inject inside the network in order to annotate a given graph. This also significantly reduces the number of TCP flows that we need to open simultaneously.

7. We conduct extensive experiments on the real Internet (a collection of 17 nodes geographically distributed in the US that form the RON network[58]), and show that our system converges even under dynamically changing network state. Both variability in available bandwidth and shift in location of bottleneck link are slow enough that they allow our system to converge, and maintain a consistent graph.

3.4 Network annotation algorithms

Knowing the underlying network structure and link capacities that connect an application's endpoints is the key input needed for planning network resource usage. This input is provided by Netmapper, which takes as input the set of endpoints for a network application, and outputs an annotated graph representing the network connectivity and link capacities between these endpoints. Netmapper uses pairwise (layer-3) traceroutes between the endpoints to map out the network connectivity. It then annotates the edges of this graph, to the extent possible, with their available bandwidth and latency. The term "available bandwidth" refers to the amount of bandwidth that an application can get by opening one or more TCP flows. Note that available bandwidth as described herein is different from raw bandwidth of a link, or bandwidth obtained by a UDP flow. Assuming that we can get the latency annotation using per-hop delays reported by traceroute, we will focus on bandwidth annotation in the remaining part of this section.

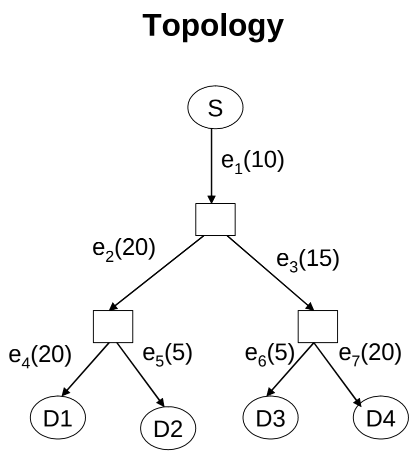
Netmapper must utilize end-to-end probe flows to annotate links inside the network. The extent to which Netmapper is able to annotate links depends upon the set of endpoints available to it for running probe traffic. As more endpoints become available from a set of applications, Netmapper is able to refine the annotation with more detail. The key challenge is to minimize the number of probes needed to annotate the whole graph.

3.4.1 Basic idea behind annotating a link

To understand Netmapper's operation, we must first understand the fundamental method to annotate internal links of a graph when we only know the bandwidth on a given set of paths in the graph. The way to annotate a link in the network with its capacity is to *saturate* the link such that no more traffic can be driven through it. The capacity of the link is then simply the amount of traffic that is being driven through it at saturation. The indication of saturation is that the total amount of bandwidth achieved at the receivers of the flows being driven through the link, is less than the total amount of traffic that senders are able to transmit. Probe flows can thus be orchestrated in a way that links of the network are successively annotated. The goal in the selection of these flows should be to minimize the overall amount of probe traffic in the network. Note that the share of network bandwidth of a set of TCP probe flows, and hence their intrusiveness, is proportional to the number of simultaneous flows. So, we also need to minimize the number of *simultaneous* probe flows in the network. In this section, we describe various algorithms to annotate each edge of the graph with available bandwidth.

3.4.2 Naive algorithm

A naive way to annotate a graph is to try sending as much traffic as possible through each edge. In other words, for each edge e of the graph, we find the set of paths that go through the edge (denoted by $P(e)$). We run a TCP flow along all these paths simultaneously, and annotate the edge e with total bandwidth that



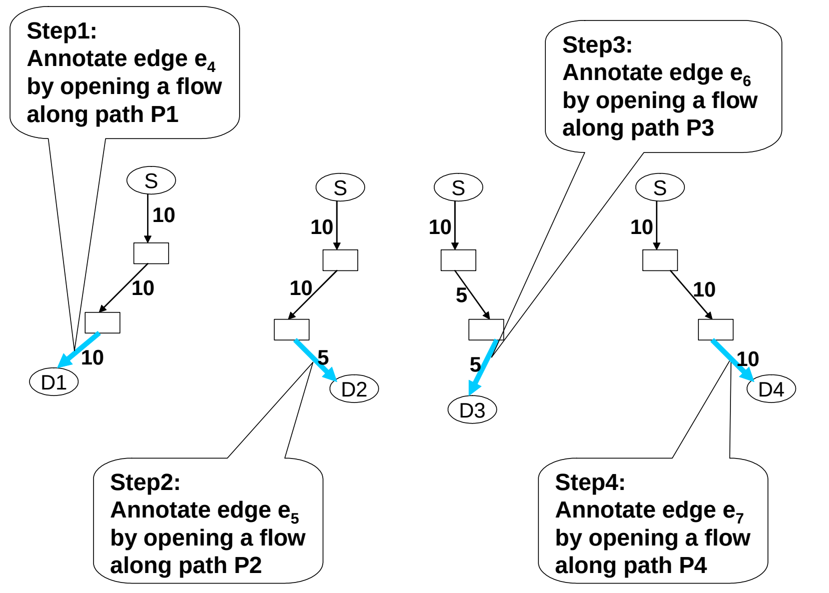
List of end-to-end paths for the topology

Path	Src	Dst	Edges
P1	S	D1	e_1, e_2, e_4
P2	S	D2	e_1, e_2, e_5
P3	S	D3	e_1, e_3, e_6
P4	S	D4	e_1, e_3, e_7

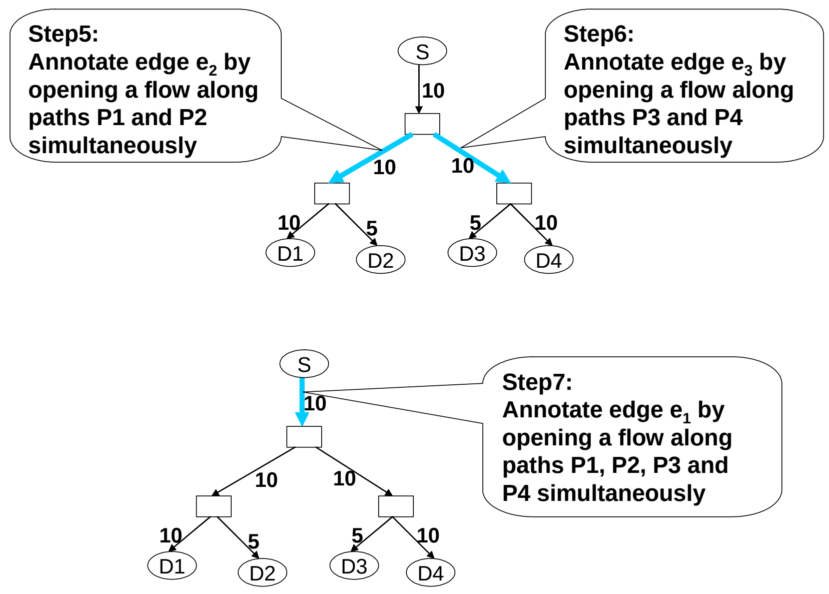
Figure 3.1: Example topology used to show how the improved algorithm with bottleneck identification reduces the amount of probe traffic needed to annotate a graph, as compared to a naive scheme.

all the TCP flows get. The cost of this algorithm (in terms of the number of TCP flows) is equal to sum total of the number of paths going through each edge of the graph. We call this as the *naive algorithm*, since it tries to pump the maximum possible flow through each edge in order to annotate it.

Figure 3.1 shows an example of a directed graph with a tree structure, consisting of single source S, and four destinations D_1, D_2, D_3 and D_4. The figure also shows the actual link capacities in the graph. Figure 3.2 shows the steps that the naive algorithm would take to annotate the graph. We consider all edges in the graph, in order of the increasing number of paths that go through the edge. Thus, we start by annotating edges e_4, e_5, e_6 and e_7, since there is only one path that goes


Step1:
Annotate edge e_4 by opening a flow along path P1
Step2:
Annotate edge e_5 by opening a flow along path P2
Step3:
Annotate edge e_6 by opening a flow along path P3
Step4:
Annotate edge e_7 by opening a flow along path P4
S
10
5
D1
D2
D3
D4

(a)


Step5:
Annotate edge e_2 by opening a flow along paths P1 and P2 simultaneously
Step6:
Annotate edge e_3 by opening a flow along paths P3 and P4 simultaneously
Step7:
Annotate edge e_1 by opening a flow along paths P1, P2, P3 and P4 simultaneously
S
10
5
D1
D2
D3
D4

(b)

Figure 3.2: Annotation steps with naive algorithm for the topology in Figure 3.1. The annotation requires a total of seven steps, three of which drive multiple probe flows simultaneously into the network.

through each of these edges. We next consider edges e_2 and e_3, each of which has two paths going through it. Finally, we annotate edge e_1, which has all the four paths traversing through it.

Assuming the naive algorithm picks edge e_4 in the first step, we start with a probe between S and D_1 in step1. The path bandwidth is 10, which means at least 10 units of bandwidth is available on edges e_1, e_2 and e_4. No more information can be gathered from this path. We annotate each of the three edges e_1, e_2 and e_4 with 10 units of bandwidth. The algorithm next tries to annotate edge e_5 in step2, opening a probe between S and D_2. This gives a bandwidth of 5, since the minimum bandwidth amongst the edges e_1, e_2 and e_5 is 5 units. This allows us to annotate edge e_5 with 5 units. Note that the annotation of edges e_1 and e_2 remains unchanged at 10 units. Similarly, the algorithm proceeds through steps 3 and 4 (shown in Figure 3.2(a)) to annotate edges e_1, e_3, e_6 and e_7. Running a probe between S and D_3 (step3) yields 5, which annotates e_3 and e_6 with 5. Running a probe between S and D_4 (step4) yields 10, which annotates e_1, e_3 and e_7 with 10.

Next, the algorithm tries to annotate edges e_2 and e_3, since there are two paths that go through each of these edges. In step5, we try to annotate edge e_2, which has two flows along paths P1 and P2 passing through it. We know the amount of bandwidth that each of these two flows get when we run them individually (10 and 5 respectively). So, the maximum amount of traffic than can be pumped through the edge e_2 is 15. However, current annotation of the edge is only 10. In order to find the actual bandwidth of the edge e_2 in the range of [10, 15], we must run the two flows simultaneously. Running both flows together yields 10. The two flows go through edges e_1 and e_2, and hence we annotate both these edges with 10. Similarly, in step6, the algorithm tries to annotate edge e_3. The two flows going through the edge, viz P3 and P4, get bandwidth of 5 and 10 units respectively when run independently. The maximum flow through the edge e_3 can be up to 15. Running both flows together yields 10, hence edges e_1 and e_3 are annotated with 10 as well.

Finally, edge e_1 must be tested for saturation, which has four paths P1, P2, P3 and P4 going through it. Current annotation for the edge is 10. We know that we can get 10 units of traffic by opening P1 and P2 simultaneously. We also get 10 units by opening P3 and P4 simultaneously. So, if we try to open all the four flows P1, P2, P3 and P4 simultaneously, we could get up to 20 units of bandwidth. Running the four flows together yields 10 as the final annotation for edge e_1. The total number of steps taken by the algorithm are 7, 3 of them requiring driving multiple probe flows simultaneously into the network.

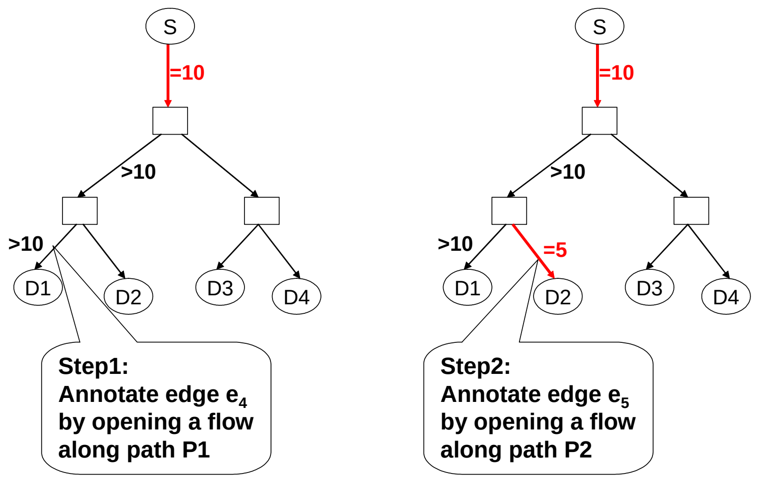
3.4.3 An improved algorithm with bottleneck identification

Netmapper uses a powerful primitive that can significantly reduce the number of probes and amount of probe traffic required by the basic algorithm. This primitive is bottleneck identification. By identifying the location of bottleneck on a path, we are not only able to annotate the bottleneck link in one step, we also eliminate a number of probe steps proportional to the size of the subgraph connecting the bottleneck link with various destination nodes. For this, we have developed a novel scheme to identify the location of bottlenecks in a network that we describe in detail in Chapter 4. The tool improves upon existing methods [53, 39] for bottleneck identification. Specific contributions over existing schemes include identifying bottlenecks with little to no impact on the regular traffic, and matching TCP's notion of bottleneck (as opposed to, say link-level definitions).

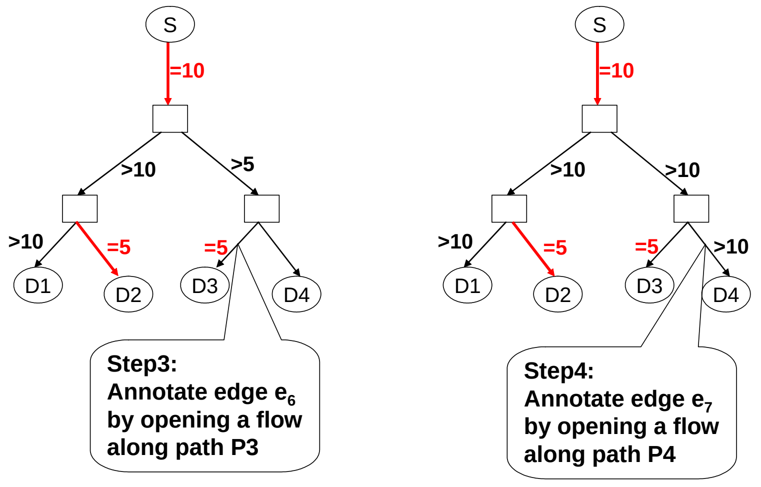
At each step, a network link may be annotated with an equality (=), representing that the link has been saturated, or an inequality (>), representing that the link's annotation could be further refined using subsequent steps in the algorithm. The equality results when the link was fully saturated in some measurement step in the algorithm. Inequalities indicate that the link has not been fully saturated during any measurement, and hence more capacity in the link could be available. The final annotation of the network, at the end of the algorithm's execution, could include both equalities (=) and inequalities (>). The final state of the network represents the maximum information that can be derived about the network's annotation from end-to-end measurements.

We illustrate this point using the example graph shown in Figure 3.1. The steps involved in annotation using the improved algorithm with bottleneck identification are shown in Figure 3.3. As in the case of naive scheme, we consider edges in increasing order of the number of paths that go through the edge. In step1, we try to annotate edge e_4 by starting a flow from S to D_1. We identify edge e_1 as the bottleneck, with a path bandwidth of 10 units. This means that the bandwidth on edge e_1 is equal to 10, while that on edges e_2 and e_4 is greater than 10. Thus, we annotate edge e_1 as "=10", and edges e_2 and e_4 with ">10". By identifying edge e_1 as the bottleneck link, we have been able to annotate it by opening only a single TCP flow, as opposed to the naive scheme, where we had to open four flows in order to annotate the edge. More importantly, this also significantly reduces the steps to annotate other parts of the graph, as we shall see shortly.

In step2, we start a flow from S to D_2 to annotate edge e_5. This gives edge e_5 as the bottleneck, with a bandwidth of 5 units. We annotate edge e_5 as "=5", and


S
=10
>10
>10
D1
D2
D3
D4
Step1: Annotate edge e_4 by opening a flow along path P1
S
=10
>10
>10
=5
D1
D2
D3
D4
Step2: Annotate edge e_5 by opening a flow along path P2

(a)


S
=10
>10
>5
>10
=5
=5
D1
D2
D3
D4
Step3: Annotate edge e_6 by opening a flow along path P3
S
=10
>10
>10
>10
=5
=5
>10
D1
D2
D3
D4
Step4: Annotate edge e_7 by opening a flow along path P4

(b)

Figure 3.3: Annotation steps for improved algorithm with bottleneck identification for the topology in Figure 3.1. The annotation requires only four steps (down from seven steps taken by the naive scheme), each requiring only one probe flow.

leave the annotations for edges e_1 and e_2 unchanged. Similarly, in step3 and step4 (shown in Figure 3.3), by running a single probe from S to D_3 and D_4 respectively, we annotate links e_6 and e_7 completely.

Next we consider edge e_2 which has a current annotation of ">10". Due to e_1 being bottlenecked at 10, there is no way to pump more than 10 units into the edge e_2, hence, we cannot update its annotation any further. The same holds true for edge e_3 as well. We already know the final annotation for edge e_1. This gives us the complete annotation for the whole graph in just 4 steps (down from 7 steps taken by naive algorithm), each requiring only one probe flow.

3.4.4 The general case

The complete algorithm for the general case can be simply stated as follows. When we identify a link as the bottleneck, with capacity C, we consider all paths of which the link is a part, and annotate each edge on such paths with C up to the point where another path is incident on this path. The rationale is that for edges in this part of the path, there is no way to drive more traffic into the edge than C. Since IP routing from a source to a set of destinations has a tree structure, this yields a significant reduction in the probe traffic that needs to be generated to annotate links downstream from a bottleneck link.

As an instance, consider the graph shown in Figure 3.4. If we identify edge e_1 as bottleneck for some end-to-end path, with capacity C, we can annotate edges e_2, e_3, e_4 and e_5 with C, since we cannot pump more traffic than C into these edges. However, for the edges e_6 and e_7, we can transmit traffic through two sources S1 and S2. So, we need to check if edges e_6 and e_7 have higher available bandwidth than C by running simultaneous probes from the two sources.

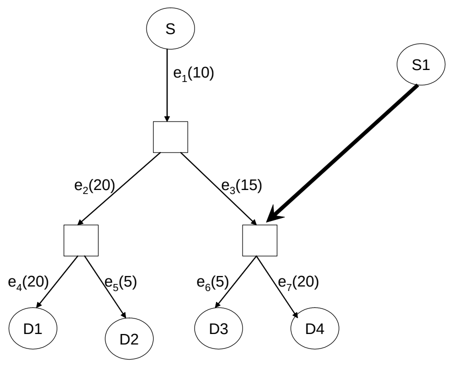
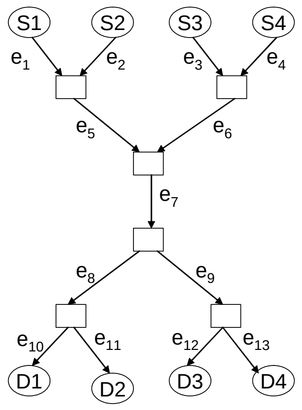


Figure 3.4: Example showing that when we identify a link as the bottleneck, we annotate each edge on all paths of which the link is a part, up to the point where another path is incident on this path.

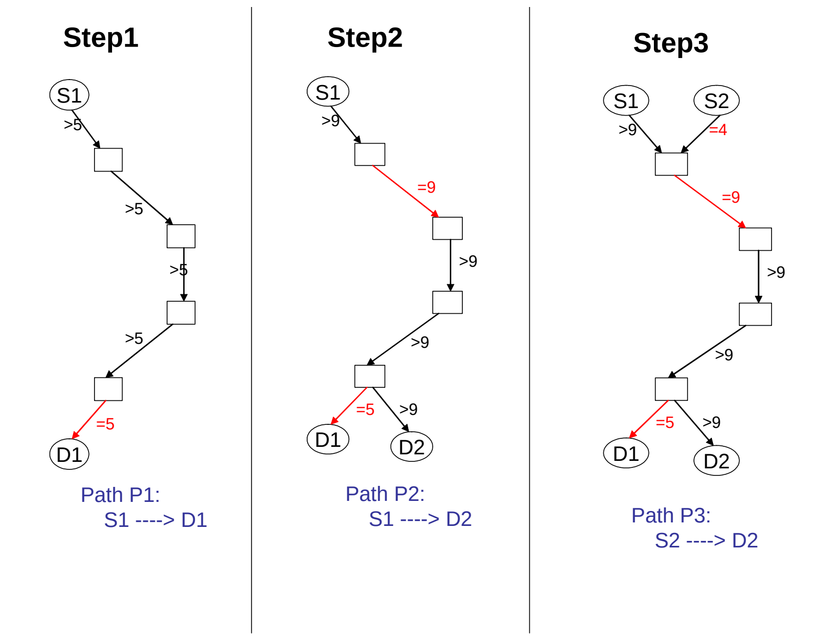
Topology



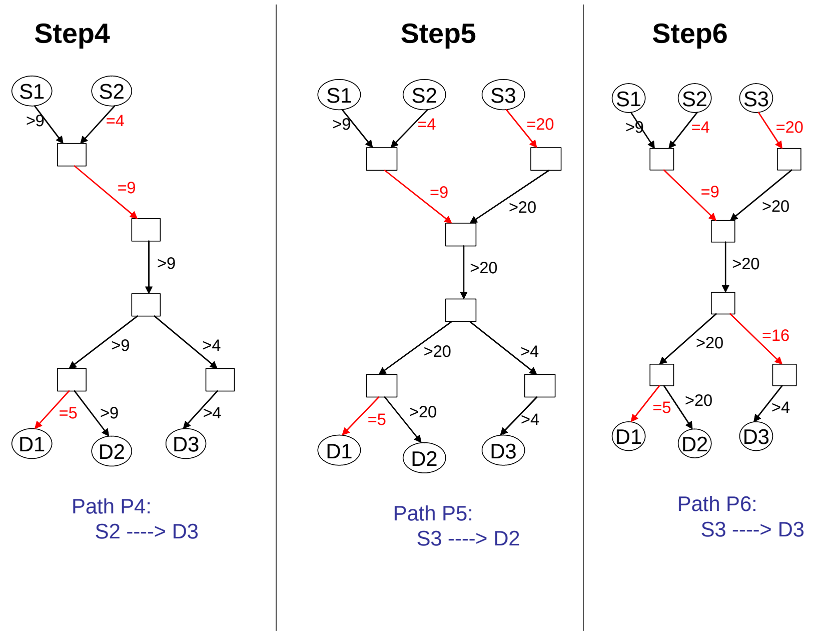
List of end-to-end paths for the topology

Path	Src	Dst	Edges
P1	S1	D1	e_1, e_5, e_7, e_8, e_{10}
P2	S1	D2	e_1, e_5, e_7, e_8, e_{11}
P3	S2	D2	e_2, e_5, e_7, e_8, e_{11}
P4	S2	D3	e_2, e_5, e_7, e_9, e_{12}
P5	S3	D2	e_3, e_6, e_7, e_8, e_{11}
P6	S3	D3	e_3, e_6, e_7, e_9, e_{12}
P7	S4	D3	e_4, e_6, e_7, e_9, e_{12}
P8	S4	D4	e_4, e_6, e_7, e_9, e_{13}

Figure 3.5: Topology and list of end-to-end paths used to illustrate the improved algorithm using bottleneck identification for the general case.



(a)



(b)

Figure 3.6: First six steps for the pre-processing stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5. Continued to Figure 3.7

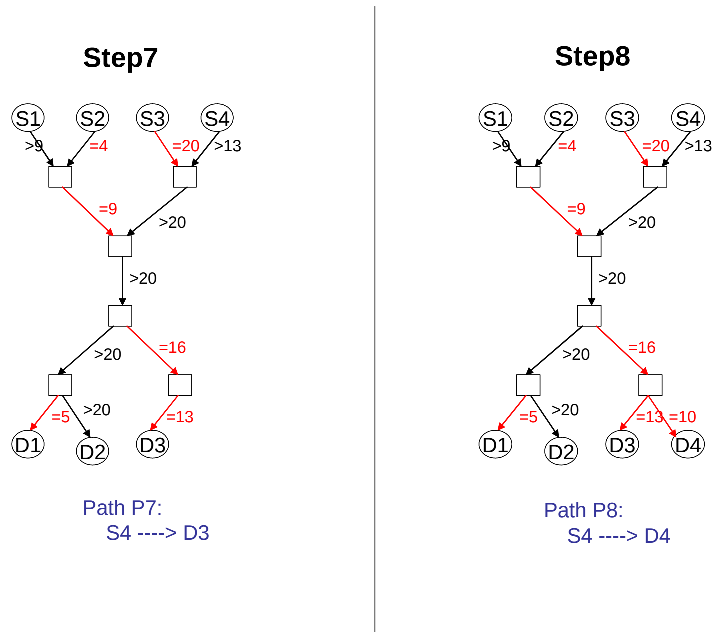


Figure 3.7: Steps 7 and 8 for the pre-processing stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5. Continued from Figure 3.6

Figure 3.11 shows the pre-processing stage of the improved algorithm for determining annotated network topology between network endpoints. The system takes a set of network endpoint addresses as input, for which we need to obtain an annotated network topology. The system starts by running layer-3 traceroute between each pair of endpoints to obtain an un-annotated network topology. The topological map of the network includes constraints on routing and exposes path sharing. The edges of the resulting topological map are then annotated with available bandwidth information. We initialize the annotation of each edge in the un-annotated topology as ">0".

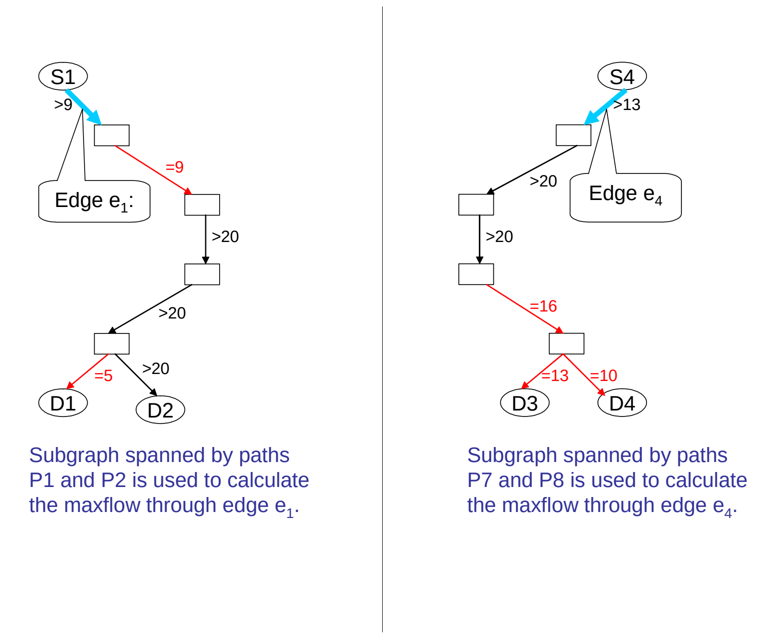
For each pair of endpoints, we open a TCP flow in order to determine the available bandwidth "B" and the location of bottleneck link on the network path connecting the endpoints. We update the annotation of each edge in the network path with the maximum of its current annotation and "B". We annotate the edge identified to be the path bottleneck with maximum available bandwidth "=B".

We illustrate this with the help of a sample topology shown in Figure 3.5 with four sources and four destinations. The figure also shows the list of end-to-end paths. Note that in practice, there could be a path from each source to each destination. But for the ease of illustration, we consider only a subset of the paths. Figure 3.6 and Figure 3.7 show the pre-processing stage of the algorithm for the given topology.

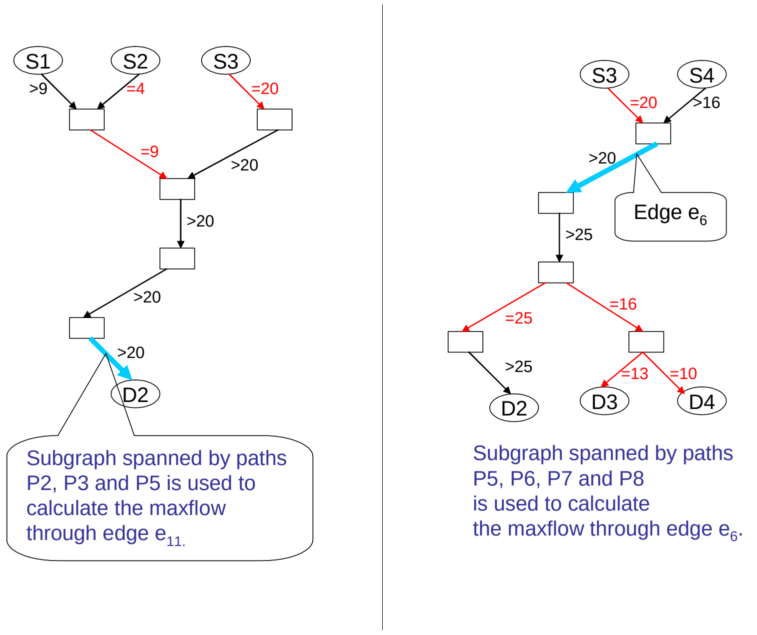
In step1, we open a TCP flow along the path P1 from source S1 to destination D1. We identify edge e_{10} as the bottleneck, with a bandwidth of 5 units. This means that we have been able to saturate the edge e_{10} with 5 units of bandwidth, and hence, we annotate it with "=5". For all other edges on the path, viz, e_1, e_5, e_7 and e_8, we annotate them with ">5", since these edges have at least five units of bandwidth available. In step2, we start a TCP flow along path P2, and identify edge e_5 as bottleneck with 9 units of bandwidth. We annotate edge e_5 with "=9". We also update the annotations of edges e_1, e_7 and e_8 from their current annotation of ">5" to ">9". Similarly, in steps 3 to 8, we open a single TCP flow along a given end-to-end path, identify the location of bottleneck link and bandwidth, and update the annotations of various edges.

The annotation of the network topology obtained in the pre-processing stage is further refined until it cannot be further refined. Figure 3.12 shows the recursive stage of the algorithm for determining annotated network topology. The algorithm starts at the innermost loop and iterates over all the edges in the graph to determine if annotation of any edge can be further refined. If it is determined that annotation of an edge can be further refined, then the variable "*changed*" is set to true, and we proceed further. Otherwise the method terminates when an iteration over all edges does not change the annotation of any edge.

In the next step, for each edge e that has an inequality (">") annotation, the amount of data that can be sent through the edge e is determined from the various paths that feed into the respective edge e. Note that the maximum amount of traffic which can be driven through an edge e depends not only on its own available bandwidth, but also on the available bandwidth of other edges in the graph through which a subset of the flows (that go through the edge e) pass. Given the capacity constraints of other edges in the graph, we find the maximum amount of traffic that can be driven through the edge. We define this as "maxflow computation", and refer to it as "maxflow"(M). For this, we construct the sub-graph G_e spanned by all the paths that go through the edge e. Figure 3.8 shows few sample sub-graphs for various edges in the topology shown in Figure 3.5.


S1
>9
Edge e1:
=9
>20
>20
=5
>20
D1
D2
Subgraph spanned by paths P1 and P2 is used to calculate the maxflow through edge e1.
S4
>13
>20
Edge e4
>20
=16
=13
=10
D3
D4
Subgraph spanned by paths P7 and P8 is used to calculate the maxflow through edge e4.

(a)


S1
S2
S3
>9
=4
=20
=9
>20
>20
>20
>20
D2
Subgraph spanned by paths P2, P3 and P5 is used to calculate the maxflow through edge e11.
S3
S4
=20
>16
>20
Edge e6
>25
=25
=16
>25
=13
=10
D2
D3
D4
Subgraph spanned by paths P5, P6, P7 and P8 is used to calculate the maxflow through edge e6.

(b)

Figure 3.8: Examples showing the subgraph spanned by paths through a given edge. This is used to calculate the Maxflow through the edge.

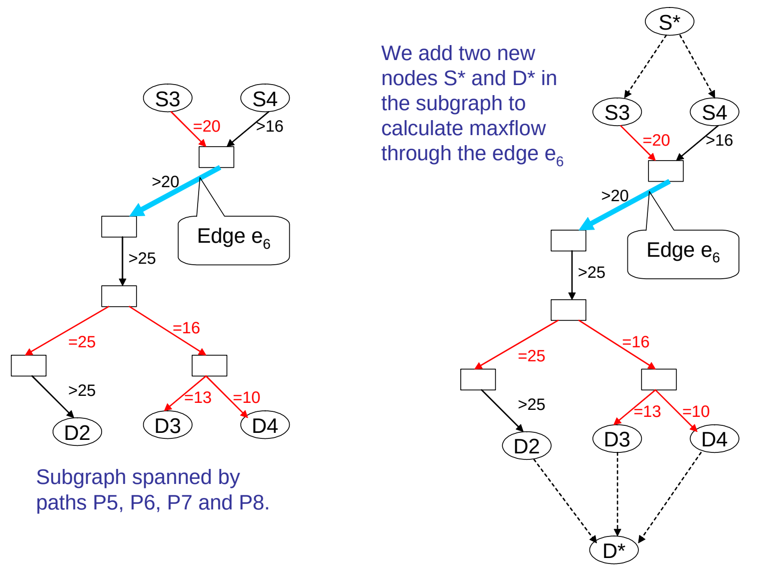

S3
S4
=20
>16
>20
Edge e_6
>25
=25
=16
>25
=13
=10
D2
D3
D4
Subgraph spanned by paths P5, P6, P7 and P8.
We add two new nodes S* and D* in the subgraph to calculate maxflow through the edge e_6
S*
D*

Figure 3.9: Example showing how we add two new nodes in the subgraph, in order to calculate the max-flow edge e_6.

Most of the standard max-flow algorithms [65] calculate the maximum amount of traffic between a given pair of source and destination nodes. In order to make our system compatible with the these algorithms, we introduce two new nodes S^* and D^* in the subgraph G_e described above. We add edges from the new node S^* to all sources in the graph G_e, and edges from all destinations in the graph to the new node D^*. We assign infinite capacity to these new edges, so that they do not become bottlenecks in any max-flow computation. Figure 3.9 shows an example of the two new nodes added to the subgraph spanned by paths that go through the edge e_6. Assume that we know the available bandwidth (using only edges in the set $P(e)$) through all edges in the graph G_e other than the edge e. We calculate the maximum possible flow that we can send from S^* to D^* using the standard max-flow algorithm [65].

Note that if the value of maxflow computation M is equal to the current annotation of the edge e, then we cannot update the annotation of the edge any further. So, we leave the edge e unchanged, and proceed to the next edge. This is an advantage of the bottleneck identification scheme. For example, if all flows going through the current edge e also pass through another common edge in the graph that has already been saturated (identified as bottleneck), then we know the maximum amount of traffic that can be pumped through the edge. If this is equal to the current annotation of the edge, then we leave the edge e unchanged. This is illustrated in step9 of Figure 3.10, where we are trying to annotate edge e_1. Using the subgraph shown in Figure 3.8(a), we see that the maxflow through edge e_1 is 9 units. However, we have already annotated the edge with "> 9", hence, we cannot refine its annotation any further by running flows along the paths P1 and P2 simultaneously. Next consider the step10 shown in the same figure, where we are considering to refine the annotation of edge e_4. The maxflow through the edge is 16, which is more than its current annotation. Hence, we open a TCP flow along the paths P7 and P8 simultaneously in order to annotate edge e_4.

If the maxflow M is greater than the current annotation, we choose a subset of paths S that pass through edge e such that bandwidth of the paths is greater than or equal to M. We later describe in Section 3.4.6 on how to pick the subset S that contains minimum number of flows. Next we activate the paths in the set S, i.e., we start TCP flows between the paths in order to drive an amount of traffic M through the edge e. We measure the total throughput achieved by all the TCP flows, and also identify the location of bottleneck link. If the total throughput T is less than the maxflow computation, we mark edge e as a bottleneck, and annotate it with "=T", indicating that the edge has been saturated. Otherwise the edge e is still not saturated. If the total throughput T is greater than the current annotation, we update the annotation for edge e to be “>T”.

This is illustrated in Step11 and Step12 of Figure 3.10(b). We are trying to annotate edge e_{11} in Step 11, which has three paths P2, P3 and P5 going through it. We calculate the maxflow through edge e_{11} as 29. However, we do not need to activate all the three flows in order to pass a flow of 29. We can achieve this by opening only two flows, viz along the paths P2 and P5. Similarly, in step 12, we can achieve the maxflow of 36 by activating only three of the four end-to-end flows, viz, along paths P5, P7 and P8.

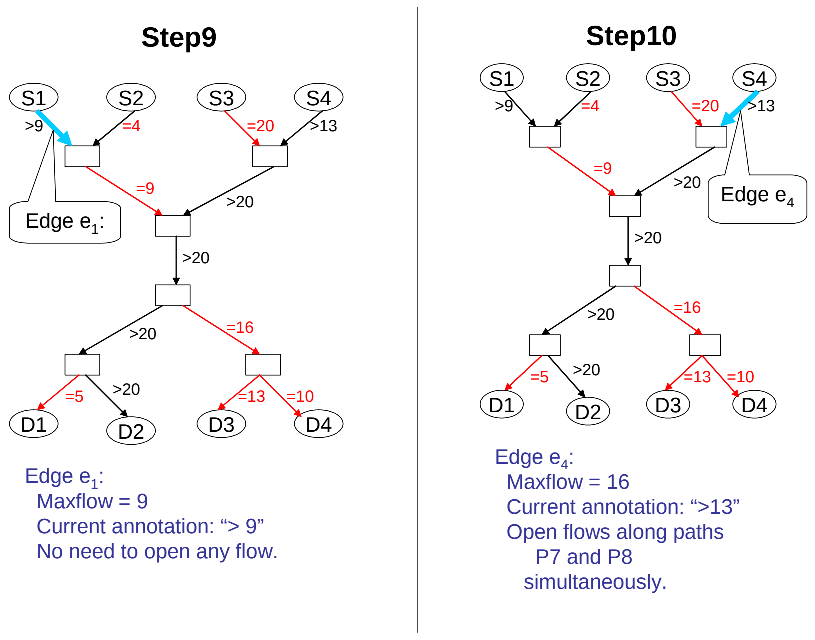
In this manner, the algorithm in Figure 3.12 iterates over all edges with annotations of inequalities. At the end of loop, if any annotation was refined, we iterate through all the steps again. Otherwise, the algorithm terminates and delivers the current network annotation as output.

3.4.5 Use of Dynamic Programming

The implementation of the naive algorithm described above is easily amenable to dynamic programming, where the maximum amount of traffic that can be driven into a link from a set of annotated subgraphs can be computed, and then the flows can be activated to check if they saturate the link. Indeed this is the approach proposed independently by [41]. However, this algorithm does not identify the location of bottleneck links along a path, and hence generates progressively more simultaneous flows in the network as the size of the subgraphs increases.

3.4.6 Choosing the minimum number of flows

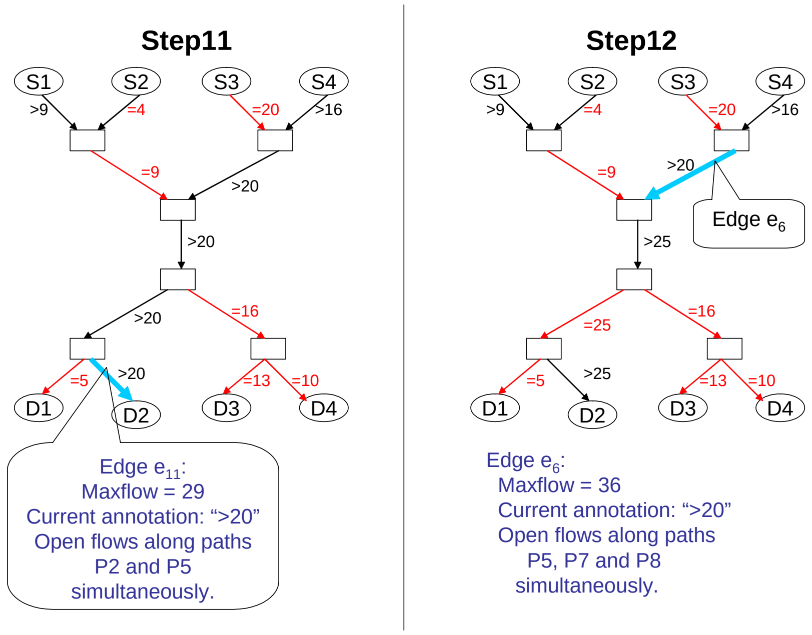
In order to annotate a given edge e, we first calculate the maximum amount of traffic that can be driven through it, given the capacity constraints of other edges



Edge e_1:
Maxflow = 9
Current annotation: “> 9”
No need to open any flow.

Edge e_4:
Maxflow = 16
Current annotation: “>13”
Open flows along paths
P7 and P8
simultaneously.

(a)



Edge e_{11}:
Maxflow = 29
Current annotation: “>20”
Open flows along paths
P2 and P5
simultaneously.

Edge e_6:
Maxflow = 36
Current annotation: “>20”
Open flows along paths
P5, P7 and P8
simultaneously.

(b)

Figure 3.10: Partial steps for the recursive stage of the improved algorithm with bottleneck identification on the topology shown in Figure 3.5.

```
Input: A set of network endpoint addresses.
Output: Network topology connecting the end-points,
    where each edge is annotated with maximum available bandwidth.
1. Run traceroute between each pair of endpoints to derive an
    un-annotated network topology.
2. Annotate each edge on the un-annotated topology with ">0".
3. For each pair of end-points,
    3a. Run a TCP flow between the end-points to identify the location of
        bottleneck link, and measure the bandwidth "B" of the path.
    3b. Annotate each link on this path with ">X",
        where X = the maximum of its current annotation and B.
    3c. Annotate the edge identified to be the path bottleneck with "=B".
```

Figure 3.11: Pre-processing stage for the improved annotation algorithm with bottleneck identification (contd. to Figure 3.12)

in the graph that share one or more end-to-end paths going through the edge e. Note that there are various path combinations that can drive total flow equal to the value given by max-flow algorithm described above. Our goal is to select the minimum number of flows that can drive this traffic through the edge. All flows through the edge e need not be activated to saturate it, since some of the paths leading into e already get saturated upstream. So, we only need to activate enough flows that can carry the saturated bandwidth down into e. The key is how to select this subset of flows. One option is to pick flows greedily, in decreasing order of their individual bandwidth when run independently. However, this scheme does

```
4. Set *Changed* to true.
5. While (*Changed*),
    5a. Set *Changed* to false.
    5b. For each edge $e$ in the graph, not annotated with "=",
        (Let the current annotation be "$> N$")
        5b.i: Compute the "maxflow" $M$ of the edge $e$ (defined as the
            maximum amount of flow that can be driven through an edge)
        5b.ii: If $M = N$ then go to step 5b.
        5b.iii: Choose a subset S of the paths passing through $e$
            whose total flow is $\geq$M.
        5b.iv: Run a TCP flow on each of the paths in set S simultaneously.
            Let the total throughput be T.
        5b.v: For an edge $e1$ identified to be the bottleneck link
            in this measurement, set the annotation of $e1$ to "=T".
            Set *changed* to true.
        5b.vi: If $T < M$, annotate edge $e$ with "=T".
            Set *changed* to true.
        5b.vii: If $T = M$ and current annotation was ">N" and $N < T$,
            then annotate edge $e$ with ">T", and set *changed* to true.
```

Figure 3.12: Recursive stage for the improved annotation algorithm with bottleneck identification (contd. from Figure 3.11)

not work, since the flows with large individual bandwidth might pass through a common bottleneck upstream. So, if we try to run them simultaneously, we may not be able to pass enough traffic through the edge e.

Our algorithm to select the minimum number of flows is as follows: We initialize the subset S with flow f_1 that gets maximum bandwidth when run independently. In the next step, amongst all the flows belonging to $P(e)$, we pick flow f_2 such that when f_1 and f_2 are run simultaneously, we get maximum total bandwidth, and add it to the set S. We find this maximum total bandwidth using standard max-flow algorithm on the subgraph spanned by edges of flows f_1 and f_2. In general, at any stage, we pick the next flow such that when the new flow is run in conjunction with the flows already in the subset S, we get the maximum increment in the total traffic that can be driven through the edge e. We repeat this process until the set of flows in S can saturate the edge e completely, i.e., the total bandwidth of all flows in S when run simultaneously is equal to the max-flow of edge e.

3.4.7 Identifying bottleneck link with simultaneous flows

If we run multiple TCP flows simultaneously, all of which pass through a common edge e, then we need to identify the location of the bottleneck link only along the flow (denoted by f_{max}) that gets maximum bandwidth when run independently. In other words, we can make edge e the bottleneck by running multiple flows simultaneously if and only if e is the bottleneck link along the flow f_{max}.

To understand this, consider the graph shown in Figure 3.13. Let f_1 denote the TCP flow from S to D_1, and f_2 denote the TCP flow from S to D_2. Let the bottleneck bandwidth of edges e, e_1 and e_2 be B, b_1 and b_2 respectively. Without loss of generality, assume that $b_1 \leq b_2$. Also assume that when we run the flows

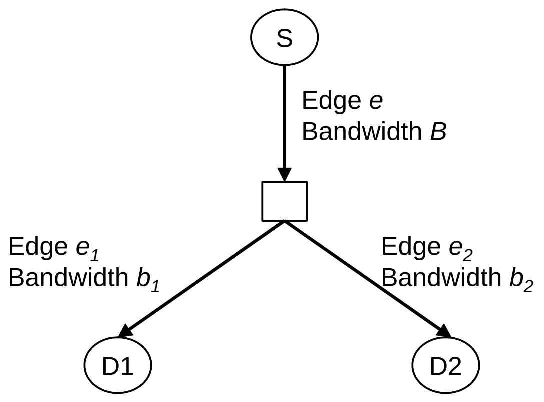


Figure 3.13: Example showing how to identify bottleneck link with simultaneous flows.

f_1 and f_2 independently, we identify edges e_1 and e_2 as bottlenecks respectively. This means that $b_1 \leq b_2 \leq B$. Let us consider what happens when we run the two flows f_1 and f_2 simultaneously.

- **Case1:** $B > b_1 + b_2$**.** In this case, the bottleneck bandwidth for common edge e is more than the maximum amount of traffic that the two flows can pump together. So, we cannot saturate edge e by running the two flows simultaneously. Edges e_1 and e_2 will continue to be the bottlenecks for flows f_1 and f_2 respectively.

- **Case2:** $B < b_1 + b_2$**.** Since $b_1 \leq b_2$, this implies that $b_2 \geq B/2$, hence, edge e will be the bottleneck for flow f_2 when the two flows are run in parallel. Note that edge e may or may not be the bottleneck for flow f_1, depending upon whether b_1 is greater than $B/2$ or not.

This means that the available bandwidth of the edge e is less than $b_1 + b_2$ if and only if the flow f_2 gets bandwidth less than b_2 when both the flows are run simultaneously.

3.4.8 Estimating bandwidth for multiple TCP flows

The algorithm described above estimates the amount of bandwidth that a single TCP flow would get on a given end-to-end path. This can easily be extended to compute the bandwidth we would get by opening multiple TCP flows through a given edge. For this, we maintain a rough estimate of the number of background flows bottlenecked at a link.

Assume that raw bandwidth of the bottleneck link is C, and that there are N background TCP flows currently going through the edge. First, we open a single TCP flow through the edge, and label its bandwidth as B_1. Assuming that all the flows going through the edge have similar RTT, each of them would get an equal fair share of the bottleneck bandwidth. Thus, $B_1 = \frac{C}{N+1}$. Next, we open k TCP flows ($k > 1$) through the edge, and denote the average bandwidth obtained by each of these as B_k. Assuming stationarity in the behavior of background flows, we can say that $B_k = \frac{C}{N+k}$. The above two equations can be solved to estimate the number of background flows N, and raw bandwidth C. Finally, if our application wants to open i TCP flows, we can estimate the amount of bandwidth obtained by each as $\frac{C}{N+i}$.

Note that this is a very rough estimation of the amount of background traffic. Firstly, a single edge may not be the bottleneck for all the flows going through it, since these flows may have different source/destination, and hence take different paths in the network. In this case, we could think of N as the number of background flows that are being bottlenecked by the edge, and C as the total bandwidth obtained by all these N flows. The above two equations still hold true.

Secondly, opening more TCP flows could increase the number of flows that are being bottlenecked at the edge. For example, consider an edge e with available bandwidth of 10 Mbps. Assume that there is another edge e_1 on the path with available bandwidth of 9 Mbps. If we open multiple TCP flows, the available bandwidth on the edge e reduced to 8 Mbps. So, all the flows that were earlier being bottlenecked on the edge e_1 will start getting bottlenecked on edge e. In practice, we assume that N is usually big enough that this effect is not very pronounced.

3.5 Experimental Results

We conducted extensive experiments on the real Internet to study the behavior of Netmapper. We used a collection of 17 nodes geographically distributed in the US that form the RON network [58]. Table 3.1 gives a list of nodes (along with their IP address and geographical location) that we used in various experiments of Netmapper. In order to have shared paths, we picked source nodes located on the West Coast, and destination nodes near the East Coast.

The primary goal of our experiments is to see how the system behaves under dynamically changing network state. The two main parameters that affect network dynamics are variability in available bandwidth, and shift in location of bottleneck link along a path. In order to understand the behavior of Netmapper under the dynamics of real Internet, we performed an extensive study on how quickly the available bandwidth and location of bottleneck link along a given path change with time. We also study how the modified annotation algorithm reduces the amount of traffic that we need to inject inside the network in order to annotate a given graph.

Table 3.1: Information about various nodes used for the Netmapper experiments.

No.	**Name**	**IP Address**	**City**	**State**
S1	ana1-gblx	64.215.37.170	Anaheim	CA
S2	digitalwest	65.164.24.58	Los Angeles	CA
S3	ucsd	132.239.17.240	La Jolla	CA
S4	msanders	205.166.119.19	San Mateo	CA
S5	utah	155.98.35.100	Salt Lake City	UT
S6	cybermesa	65.19.5.18	SantaFe	NM
D1	nyu	216.165.108.39	New York	NY
D2	webair	209.200.18.252	New York City	NY
D3	cornell	128.84.154.59	Ithaca	NY
D4	cmu	128.2.185.85	Pittsburgh	PA
D5	mit-main	18.7.14.168	Cambridge	MA
D6	roncluster1	18.31.0.181	Cambridge	MA
D7	vineyard	204.17.195.103	Boston	MA
D8	coloco	199.34.53.174	Laurel	MD
D9	mvn	66.232.160.65	Mount Vernon	IL
D10	chi1-gblx	64.215.37.86	Chicago	IL
D11	umich	141.212.113.77	Ann Arbor	MI

A brief summary of our experimental results is as follows:

1. **Bandwidth variation**: Using a 30-day long trace on 72 different paths on the Internet, we show that average bandwidth along a given path remains bounded by a factor of 2 over a period of 1.5-2 hours with high probability.

2. **Bottleneck variation**: Using a similar experimental setup as above, we have shown that the location of a bottleneck link along a path does not change at a very fast rate. The average length of time over which the bottleneck link along a path remains constant is 2-3 hours with high probability.

3. **Fast-converging algorithm**: Our experience in deploying Netmapper on the real Internet shows that it converges even under widely varying network dynamics. Both variability in available bandwidth and shift in location of bottleneck link are slow enough that they allow our system to converge, and maintain a consistent graph.

   In order to eliminate short-term variations in bandwidth/bottleneck, we show using experiments that it is sufficient to average 3-6 samples of bandwidth and bottleneck location along each path. We also validate the bandwidth annotations of Netmapper by comparing with a naive algorithm that most accurately estimates the true state of the network.

4. **Consistency across time**: We ran the netmapper algorithm on a given set of nodes for a period of 30 days, and studied how the resulting graph annotations change with time. We see that the relative change in resulting annotations is less than 0.5 with more than 80% probability.

5. **Reduced load on the network**: We show that our modified annotation algorithm reduces the total amount of probe traffic (as compared to a naive

algorithm) by approximately 50%, and significantly reduces the number of TCP flows that we need to open simultaneously.

This section is organized as follows: Section 3.5.1 studies the variation in available bandwidth on a path as a function of time. Section 3.5.2 shows how frequently the location of the bottleneck link along a path changes with time. Section 3.5.3 shows how the Netmapper algorithm behaves under the dynamics of the real Internet. Section 3.5.4 tests the consistency of our results over time. Section 3.5.5 shows how our improved annotation algorithm (using dynamic programming and bottleneck identification as a fundamental primitive) reduces the amount of probe traffic that we need to inject inside the network in order to annotate a given graph.

Note that we could not use Planet-lab testbed for these experiments because we require special privileges on the nodes. We need unconstrained TCP send/recv windows in order to make sure that bottleneck is inside the network. We use a modified version of traceroute (that sends all the ICMP probes in parallel) in order to detect the location of bottleneck links, as will be discussed later in Chapter 4. Finally, our goal is to test the system under the dynamics of the real Internet. Most inter-node paths on Planetlab use Internet-2 and Abilene, giving us a fairly over-provisioned network with less variability compared to the Internet, thus stressing our techniques less.

3.5.1 Variability in bandwidth

On a short time scale, Paxson et al. have shown [43] that bandwidth just changes so quickly that it is not possible to model it. However, we show that over a bigger time scale (order of few minutes to an hour), taking an average over few samples bounds the relative change in available bandwidth over a small time interval (typically, 1-

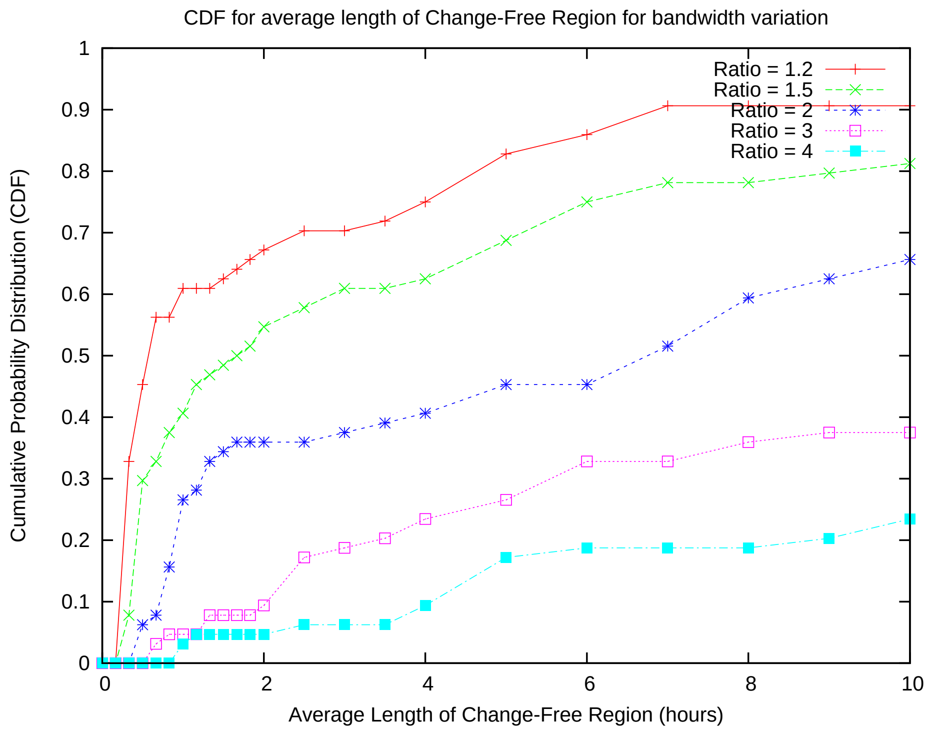


Figure 3.14: CDF for *average* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation.

2 hours) with very high probability. Paxson et al. [43] have also conducted such studies in the past, and reported similar results. But most of those studies are 5-8 years old. We performed similar experiments again to ensure that the stationarity properties of bandwidth variability hold true today.

We took 72 different paths on the Internet. On each path, we found the available bandwidth by running a TCP flow, transferring 256K of data in each sample. We took one bandwidth sample on each path every 3 minutes. We combine every three consecutive samples into one bin using the average.

We define a Change-Free Region (CFR) as the time-interval in which ratio of maximum bandwidth to minimum bandwidth remains less than some pre-specified parameter (named as ρ). Figure 3.14 shows the average length of CFR for various

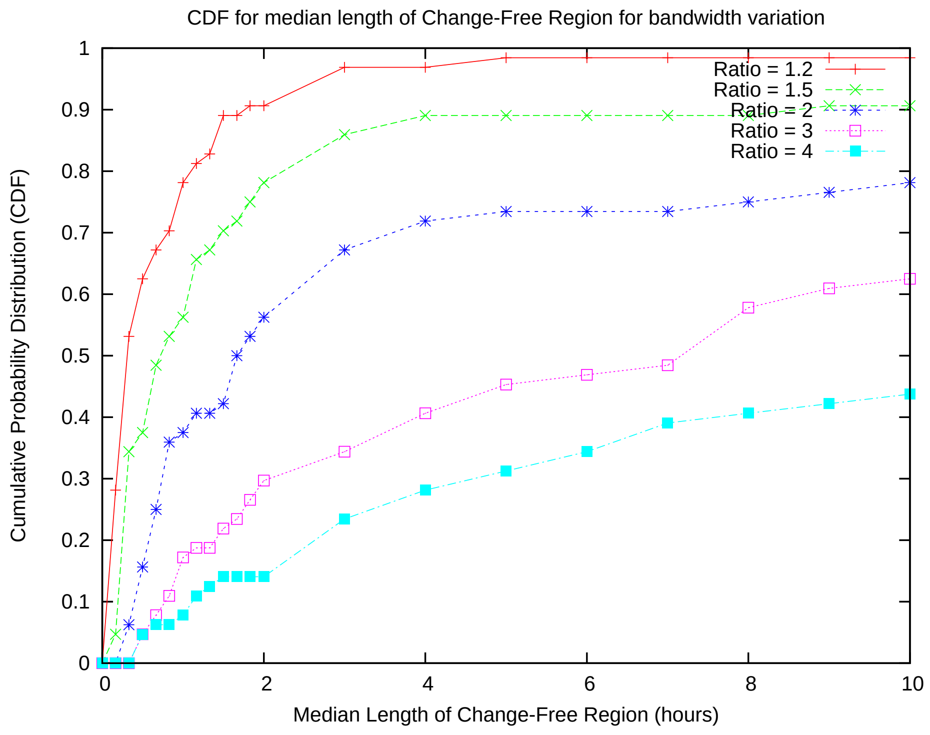


Figure 3.15: CDF for *median* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation.

values of ρ (referred to as "ratio=1.2", "ratio=1.5", etc in the figure). We see that with 65% probability, average bandwidth remains within a factor of 2 over a period of length 1.5-2 hours.

Since average value gets affected by some really large values of CFR, we also plot the CDF for *median* length of CFR along each path in Figure 3.15. Similarly, Figure 3.16 shows the CDF for *maximum* length of CFR along each path. The results for the median value of CFR are similar to that of average CFR.

3.5.2 Shift in location of bottleneck link

The goal of this section is to study how frequently the location of the bottleneck link along a path changes with time. Prior studies [53, 39] have shown steady-state

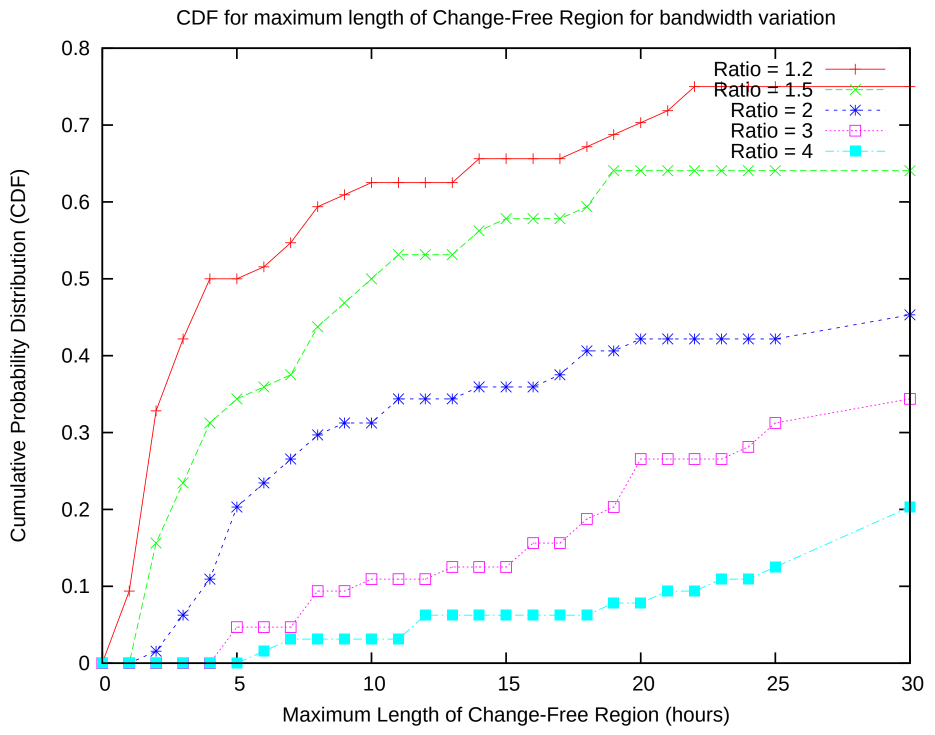


Figure 3.16: CDF for *maximum* length of Change-Free Region amongst 72 paths on the Internet for bandwidth variation.

distribution of bottlenecks, but do not consider the dynamics involved in the shift of location of bottleneck link over time. We show using experiments on the real Internet that the location of bottleneck link shifts slow enough that it allows our algorithm to converge.

We took 72 different paths on the Internet, and identified the location of bottleneck link along each path, once every few minutes, for a period of 30 days. We define a Change-Free Region (CFR) as the time-interval over which the location of bottleneck link along a path does not change. We found CFRs along all the 72 paths. We plot the CDF for the average length of CFR along each path in Figure 3.17. The blue curve (labeled as "No averaging") represents the CDF without averaging. In order to find how averaging a number of samples helps, we tried

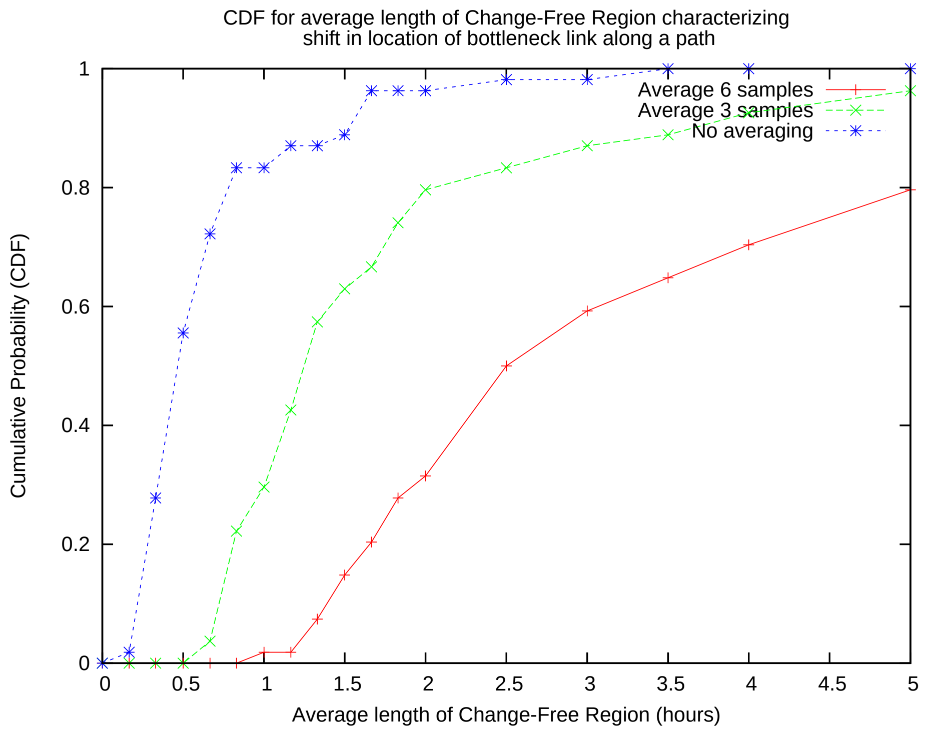


Figure 3.17: CDF for *average* length of Change-Free Region for shift in location of bottleneck link along a path.

merging 3 consecutive samples (Green curve, labeled as "Average 3 samples") and 6 consecutive samples (Red curve, labeled as "Average 6 samples"). We can see from the figure that the average length of CFR along a path is more than 30 minutes with 50% probability. Taking the average of six consecutive samples increases the average length of CFR to more than 2.5 hours with 50% probability, and more than 100 minutes with 80% probability. Note that we have no way of validating the shift in location of bottlenek link on the real Internet. We have validated our bottleneck identification tool under diverse network conditions using extensive set of experiments on the emulab [56] testbed: bottleneck due to raw bandwidth, or due to transient congestion, multiple bottlenecks on a path, location of bottleneck changing dynamically with time, bottleneck oscillating between two links, etc.

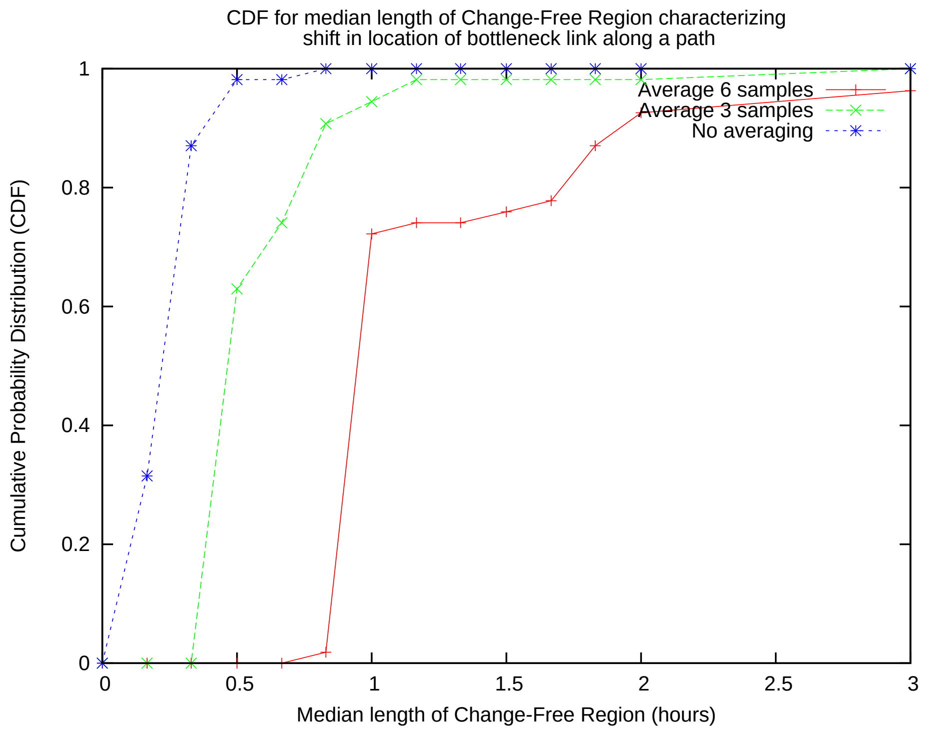


Figure 3.18: CDF for *median* length of Change-Free Region for shift in location of bottleneck link along a path.

Figures 3.18 and Figure 3.19 show the median and maximum length of CFR (respectively) along each path. The results for both median and maximum value of CFR are very similar to that of average CFR. In other words, if we average 3-6 consecutive samples of bottleneck estimation, it eliminates short-term fluctuations, and significantly increases the length of Change-Free Region.

3.5.3 Fast converging algorithm and consistent graph

Our experience in deploying Netmapper on the real Internet shows that it converges even with network dynamics. Both variability in available bandwidth and shift in location of bottleneck link are slow enough that they allow our system to converge, and maintain a consistent graph. We define a graph to be consistent if

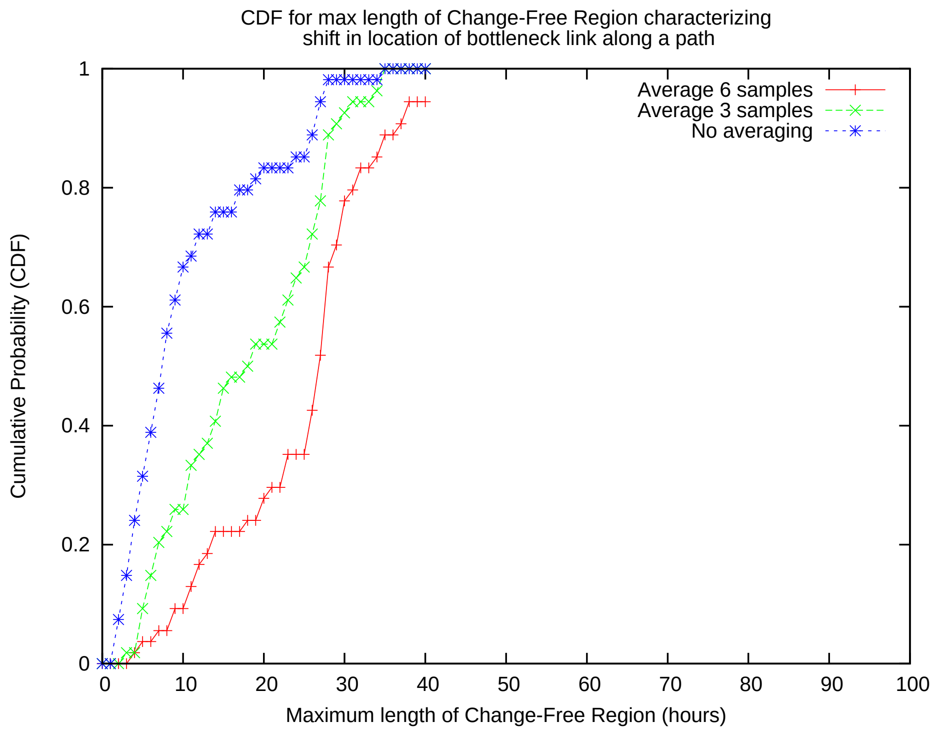


Figure 3.19: CDF for *maximum* length of Change-Free Region for shift in location of bottleneck link along a path.

the bandwidth annotation of each edge is less than both the total influx into the edge, and the total outflux out of the edge. Taking the average over a few samples (typically 3-6) of bandwidth and bottleneck estimation along a path removes most of the inconsistencies in the resulting annotated graph. Figure 3.20 shows the final annotated graph output by Netmapper for four source nodes and eight destination nodes. To visually simplify this, Figure 3.21 shows Netmapper results for a single source (located at Los Angeles), and nine destinations (around the East Coast). We can see from the figure that almost all the bandwidth annotations are consistent, except for the edge 2–>4.

In order to determine the accuracy of Netmapper's output, we compared the bandwidth annotations produced by Netmapper with the output of naive algo-

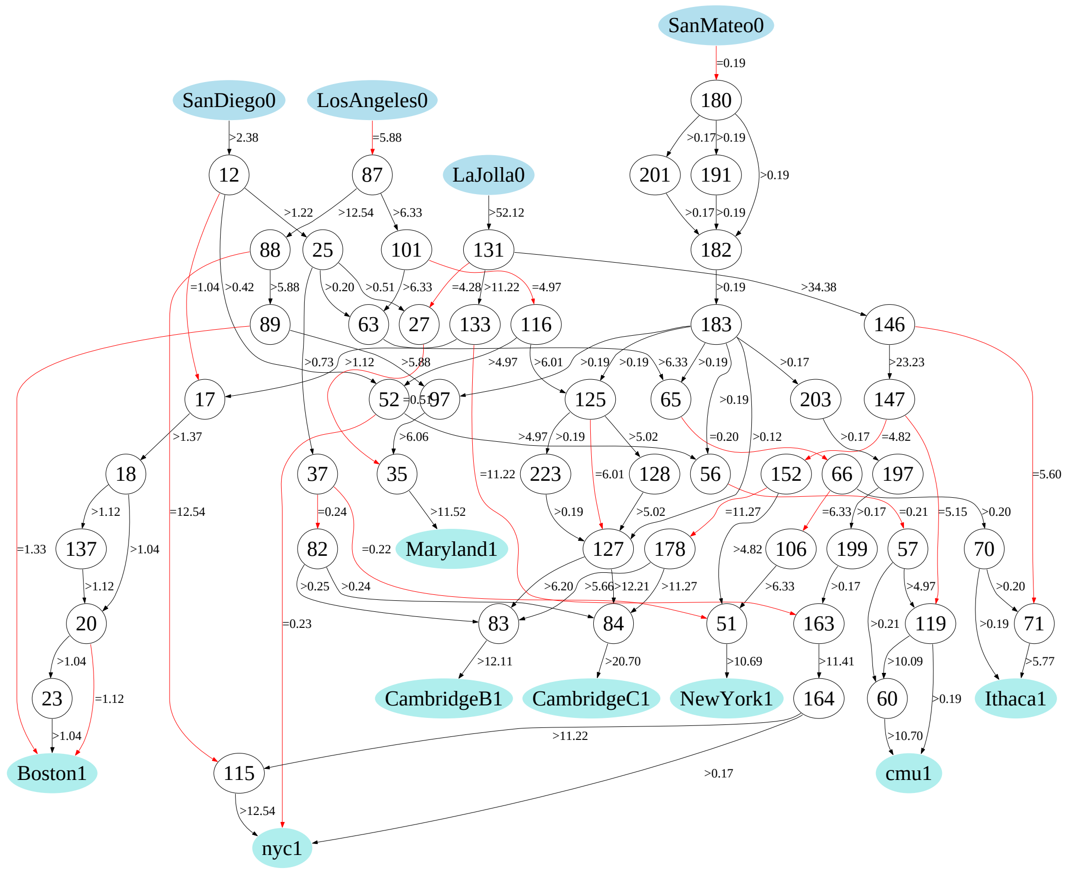


Figure 3.20: Final annotated graph output by Netmapper for four source nodes (around West Coast) and eight destination nodes (around East Coast).

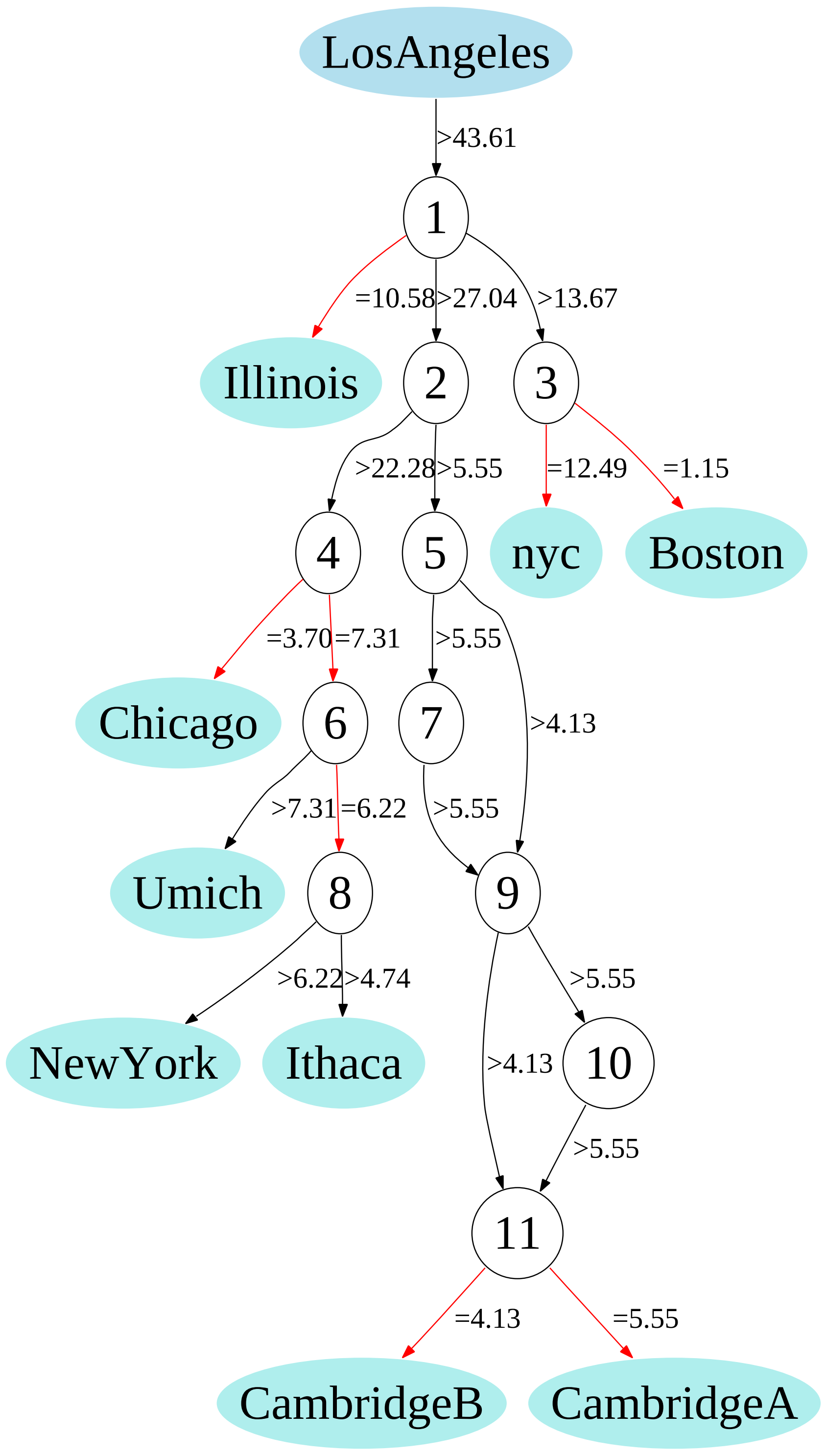


Figure 3.21: Final annotated graph output by *Netmapper* for single source node and eight destination nodes.

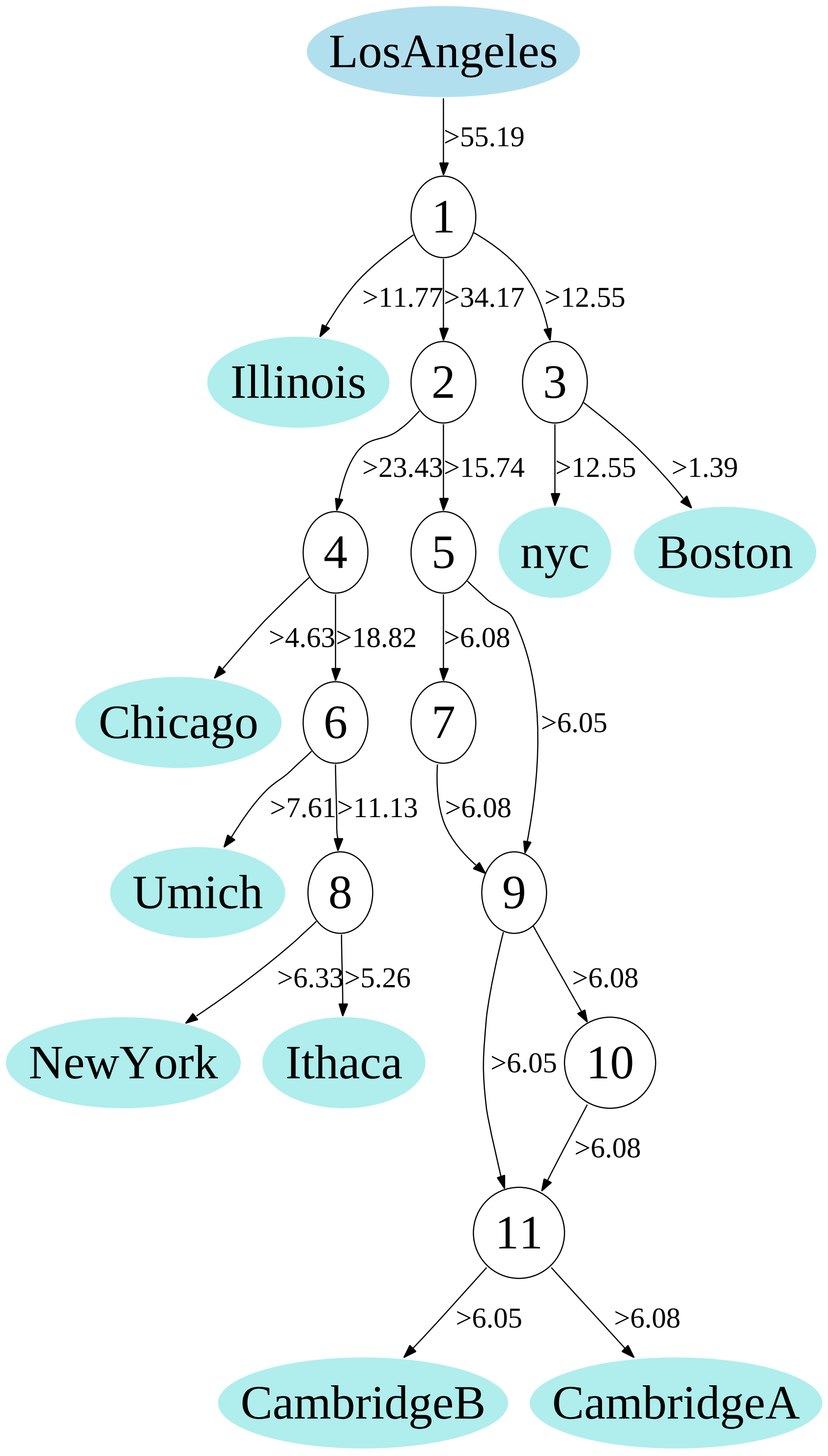


Figure 3.22: Final annotated graph output by *Naive algorithm* for single source node and eight destination nodes.

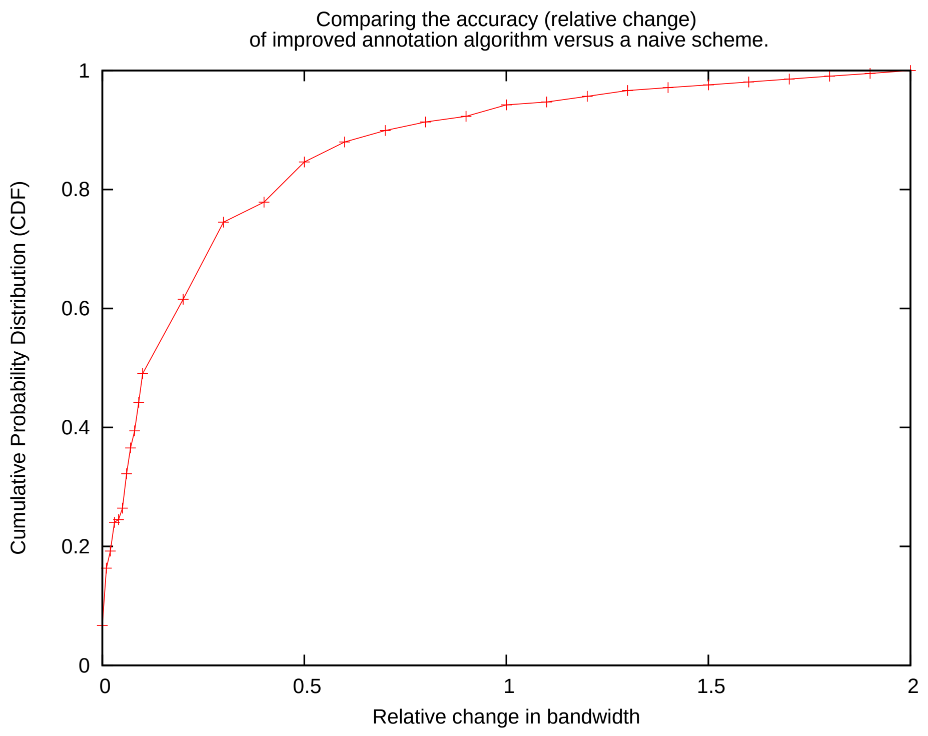


Figure 3.23: Validating the accuracy (*relative change*) of bandwidth annotations output by our improved algorithm (using bottleneck identification), with reference to the naive scheme.

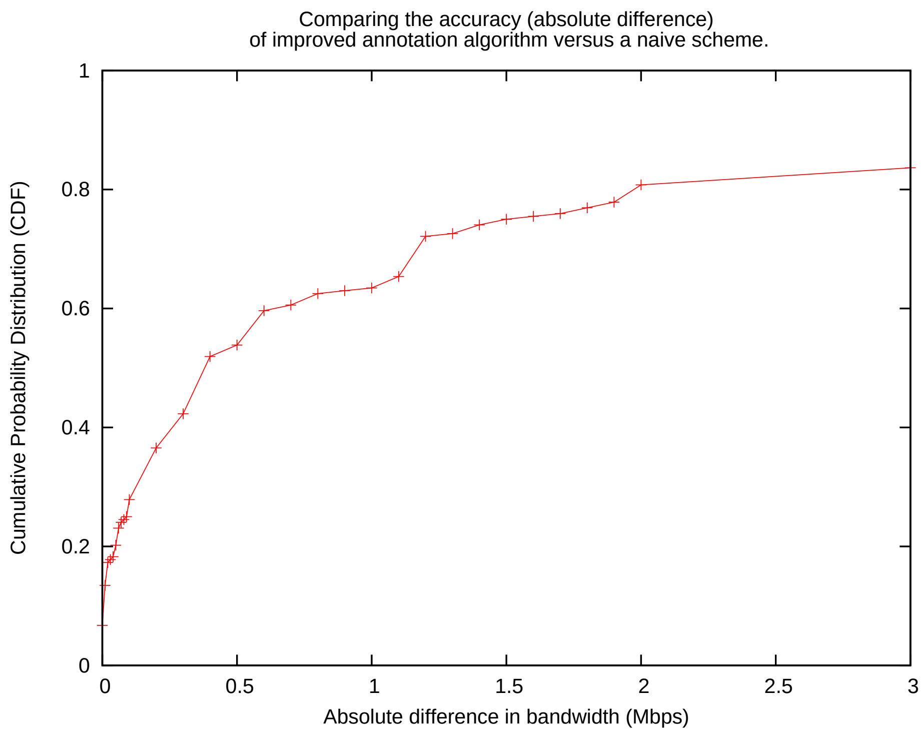


Figure 3.24: Validating the accuracy (*absolute difference*) of bandwidth annotations output by our improved algorithm (using bottleneck identification), with reference to the naive scheme.

rithm, which is the best estimator for the actual network state. Recall that the naive algorithm (See Section 3.4.2) annotates each edge by simultaneously running a TCP flow along all the paths that go through the edge. Figure 3.23 shows the Cumulative Distribution Function (CDF) for relative change in bandwidth between the output of Netmapper and naive scheme. We can see that the bandwidth annotation on each edge is very close (within 30% with 75% probability). This much variation is inherent due to the short-term fluctuations in available bandwidth, and shift in location of bottleneck link. We show later in Section 3.5.4 that relative change between the output of Netmapper at two different times is less than 0.3 with approx. 75% probability. As a sample illustration, Figure 3.21 and Figure 3.22 show the graphs output by Netmapper and the naive algorithm respectively for a system with 1 source and 9 destinations. Bandwidth annotations for most of the edges are very close, except for edges 4->6, 6->8 and 2->5.

Note that on an average, there are more bottleneck links closer to the source. We believe this is because most of the regular nodes on the Internet are provisioned asymmetrically (with downlink bandwidth significantly higher than uplink bandwidth). We also ran the Netmapper experiments on nodes that form part of IBM Intra-grid network, we found that more bottleneck links were closer to the destination. IBM nodes were provisioned in “server” style, with higher uplink bandwidth than downlink.

3.5.4 Consistency across time

Our next goal is to test the consistency of Netmapper results over time. In other words, if we run Netmapper on the same set of nodes at two different times, how similar do the resulting graphs look ? We ran the netmapper algorithm on a graph with four sources (nodes S1..S4 shown in Table 3.1), and eight destinations (nodes D1..D8 shown in Table 3.1) for a period of 30 days (once per day), and performed a pair-wise comparison of the resulting graphs. We classify each edge into one of the three classes. The first class (labelled as *both edges inequality*) consists of edges that were annotated with inequality information in both the graphs. In other words, we could not saturate these edges even by running the maximum possible traffic through them. Second class (labelled as *both edges bottleneck*) consists of edges that were identified as bottleneck in both the graphs. If an edge was identified as bottleneck in only one of the two graphs, we classify it into the third category (labelled as *one bottleneck*). We compare the resulting graphs along the following parameters:

(a) We computed *relative change in bandwidth* annotation of an edge across two different graphs. Figure 3.25 shows the CDF for the relative change in bandwidth across all pairs of graphs (amongst 30 days, approx 900 comparisons) for all the three categories of edges defined above. We see from the figure that if an edge has been identified as a bottleneck in both the graphs, the relative change in its bandwidth annotation is less than 0.5 with 83% probability. This probability increases to 89% for edges that could not be saturated in either of the two graphs. We believe this is due to the large amount of available bandwidth on these edges, and we have not been able to saturate them using our limited number of flows. On the other hand, this probability decreases to 66% for edges that were identified as bottleneck in one graph, but not the other. This could be because of relatively high variability in bandwidth across these edges. We also compared the Netmapper graphs that were a fixed time period apart, and found results to be very similar to that of comparing all the graphs.

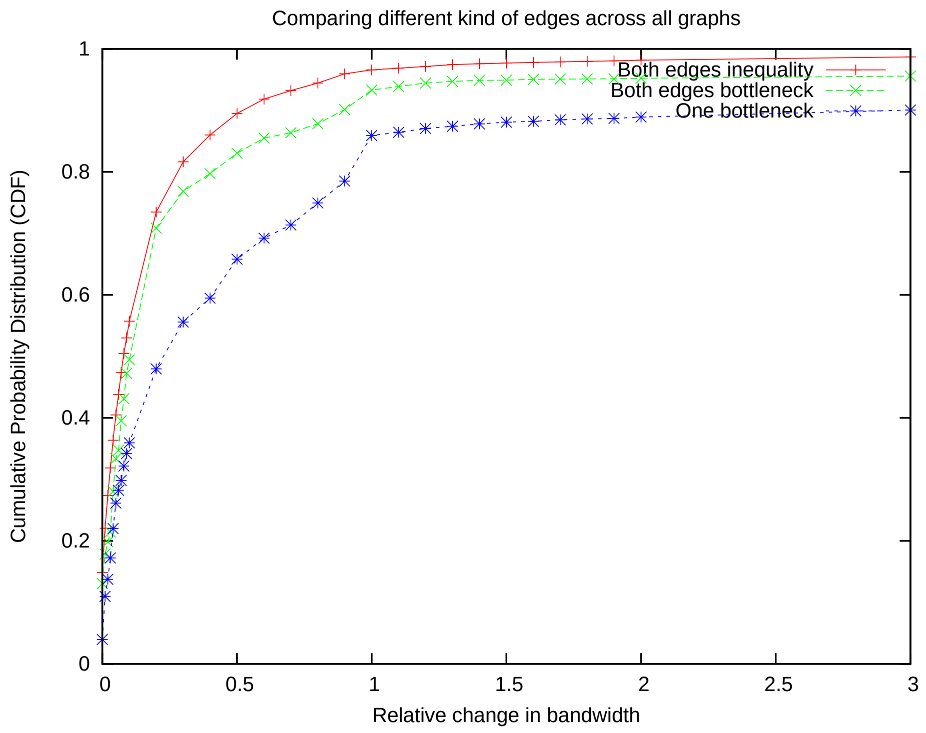


Figure 3.25: CDF for the *relative change* in bandwidth annotation of a link at two different times.

(b) Figure 3.26 shows the CDF for *absolute difference in bandwidth* annotation of a link across two different graphs. The trends are very similar to the relative change in bandwidth. Probability that difference in bandwidth annotations is less than 1Mbps is 78%, 82% and 61% for first, second and third category of edges respectively. We see that if an edge has been identified as a bottleneck in only one of the two graphs, then there is a higher chance that its absolute bandwidth differs by more than 1 Mbps.

(c) We tried to estimate the stability of our bottleneck estimation process over time. In other words, if we identify a link as bottleneck at one time, what are

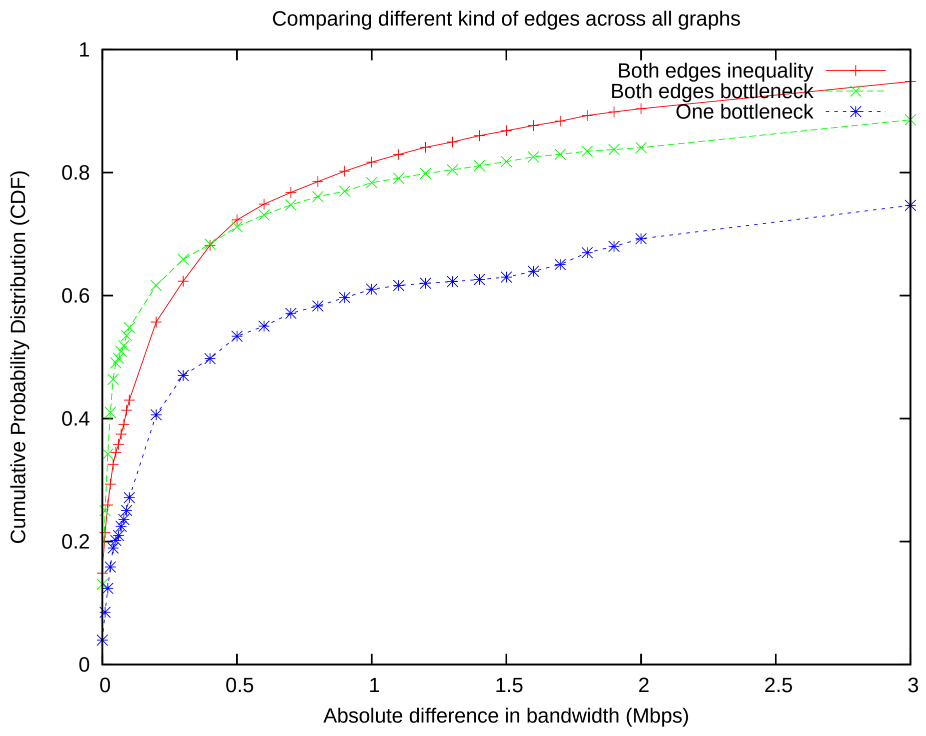


Figure 3.26: CDF for the *difference* in bandwidth annotation of a link at two different times.

the chances that we will annotate it as bottleneck again at a later time. For this, we performed a pair-wise comparison of all 30 graphs, and computed the number of links that were identified as bottleneck in one graph, but not another. We found that if we identify a link as bottleneck in one graph, with 83.5% probability, we have also identified it as bottleneck in another graph. In other words, if we could not saturate an edge at one time, there is 16.5% chance that we may be able to saturate it at some other time.

We also computed this probability for graphs that are a fixed length of time apart. For example, we compared graphs that are one day apart, two days apart, and so on. Figure 3.27 plots the probability as a function of the time interval between two runs. We see that the probability increases a little bit over time, but remains stable (less than 25%) overall.

3.5.5 Reduction in the amount of probe traffic

Our final goal is to see how much improvement our new annotation algorithm (using bottleneck identification as a fundamental primitive, as described in Section 3.4) offers, as compared to a naive scheme (which runs a TCP flow along all possible paths that go through each edge). We conducted a series of experiments with different source/destination nodes to test this. Table 3.2 shows the list of source/destination nodes used in various experiments. As described earlier, source nodes are located on the West Coast, while destination nodes are located on the East Coast (or Mid-West) in order to ensure sharing. Note that experiments *1a, 1b* and *1c* have the same set of source nodes, while experiments *1a, 2a* and *3a* have the same set of destination nodes. The goal is to see how adding a source/destination node affects the amount of probe traffic.

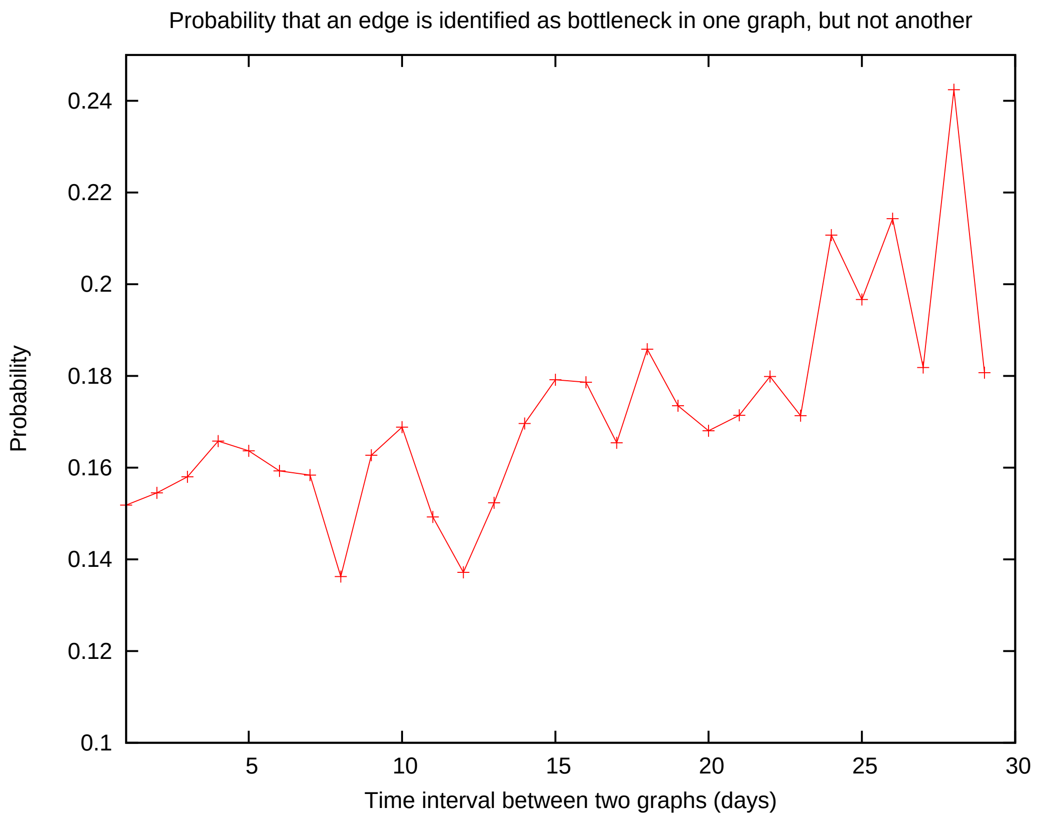


Figure 3.27: Probability that an edge is identified bottleneck in one graph, but not another as a function of the time interval between runs.

Table 3.2: List of nodes used in various experiments of Netmapper.

Expt No.	**No. of Sources**	**No. of Dests**	**List of Sources**	**List of Destinations**
1a	3	6	S1..S3	D1..D6
1b	3	8	S1..S3	D1..D8
1c	3	11	S1..S3	D1..D11
2a	4	6	S1..S4	D1..D6
2b	4	8	S1..S4	D1..D8
2c	4	11	S1..S4	D1..D11
3a	5	6	S1..S5	D1..D6
3b	5	8	S1..S5	D1..D8
3c	5	11	S1..S5	D1..D11
4a	6	6	S1..S6	D1..D6

Table 3.3 compares the total number of TCP flows that we need to open using both the improved algorithm (labeled as *Netmapper*) and the naive (labeled as *Naive*) scheme. We see that Netmapper reduces the number of TCP flows by 45-55%, as compared to the naive scheme.

Table 3.3: Comparison of the probes for various experiments of Netmapper versus Naive algorithm.

Expt	**Total No. of**	**TCP**	**Flows**	**Total No. of**	**Steps**	
No.	**Netmapper**	**Naive**	**Ratio**	**Netmapper**	**Naive**	**Ratio**
1a	53	100	0.53	28	47	0.59
1b	89	140	0.63	44	71	0.62
1c	91	188	0.48	51	84	0.61
2a	83	162	0.51	42	76	0.55
2b	97	212	0.45	49	106	0.46
2c	116	256	0.45	65	109	0.59
3a	98	201	0.48	50	80	0.62
3b	166	307	0.54	74	137	0.54
3c	175	399	0.43	93	156	0.59
4a	131	241	0.54	64	97	0.65

Note that adding a source node increases the number of TCP flows by much larger amount, as compared to adding a destination node. This could be because in our experiments, we find more bottleneck links closer to the source. As described earlier, this is due to asymmetric provisioning of nodes on the Internet.

Table 3.4 shows a breakup of the number of parallel probes for each length that we had to open. For example, if L3 = 5, it means that five steps required us to open three flows in parallel. We see that with Netmapper, there is a significant reduction in the number of parallel probes. This is really useful, since in a working system, we often have a limit on the maximum amount of probe traffic we can inject inside the network at a given time. The share of network bandwidth of a set of TCP probe flows, and hence their intrusivenes, is proportional to the number of simultaneous flows.

3.6 Conclusions

In this chapter, we have described a service called Netmapper that distributed applications can use to monitor available bandwidth in the network. The service uses sophisticated techniques to map out the connectivity and capacity of the network connecting these endpoints. We describe how the network monitoring and annotation algorithm improves upon existing monitoring solutions in several quantifiable ways. Using the annotated network topology produced by Netmapper as input, the application may use any end-to-end control knob (like MPAT[48], multi-homing[60], overlay routing[5], server selection[59], etc) to manage its performance. We have built Netmapper, and deployed it on a collection of nodes that form part of the RON[5] network. Our experience testing Netmapper on the real Internet shows that network dynamics are slow enough that they allow our system to converge.

Applications can use Netmapper at different levels of abstraction. They can use its annotated network representation natively to make their own decisions, or abstract away network monitoring and control completely in Netmapper. Multiple applications sharing a network can use Netmapper, which can also act as a resource

Table 3.4: Comparison for the breakup of the total number of probes for various experiments using Netmapper and Naive algorithms.

Expt No	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**	**L8**	**L9**	**L10**
1a-Netmapper	17	3	5	1	1	1	0	0	0	0
1a-Naive	23	9	8	2	3	2	0	0	0	0
1b-Netmapper	24	10	5	1	2	0	0	2	0	0
1b-Naive	41	13	10	1	3	0	0	3	0	0
1c-Netmapper	32	8	7	1	1	1	1	0	0	0
1c-Naive	44	16	14	2	2	1	2	0	0	1
2a-Netmapper	23	9	3	4	1	2	0	0	0	0
2a-Naive	37	17	9	6	2	5	0	0	0	0
2b-Netmapper	30	8	2	5	2	0	1	1	0	0
2b-Naive	62	19	10	7	3	0	1	4	0	0
2c-Netmapper	40	13	4	5	1	1	1	0	0	0
2c-Naive	53	24	12	8	4	1	3	1	0	1
3a-Netmapper	27	12	3	3	4	1	0	0	0	0
3a-Naive	28	22	11	5	10	3	0	1	0	0
3b-Netmapper	38	14	6	6	7	0	1	2	0	0
3b-Naive	69	28	16	6	9	2	3	4	0	0
3c-Netmapper	52	24	5	6	3	1	1	1	0	0
3c-Naive	77	30	12	12	10	2	4	3	1	0
4a-Netmapper	34	14	7	1	4	4	0	0	0	0
4a-Naive	36	24	14	10	5	7	0	1	0	0

broker when multiple applications contend for network resources.

Currently we are interfacing two applications with Netmapper. One is a peer-to-peer voice conferencing service, which stresses our algorithms both in terms of managing bandwidth and latency for the application. We are driving this study using actual traces from a production voice conferencing service. The second application is peer-to-peer network gaming to reduce game server hosting costs, where gaming peers are selected on the basis of network performance between them. This study is also being driven using traces from a production gaming service. The characteristics of these applications are significantly different, allowing us to demonstrate the broad applicability of Netmapper as a network monitoring service for different kinds of applications. Ultimately we plan to deploy a pilot of the service over the real Internet as part of a production application deployment.

Chapter 4

Tneck: A tool to identify bottlenecks

4.1 Bottleneck Identification

In this chapter, we describe our bottleneck identification mechanism, which is a component of the network mapping and annotation algorithm discussed in Chapter 3. Bottleneck identification is a method to identify which link on a network path has the least available bandwidth. The term "available bandwidth" refers to the amount of bandwidth that an application can get by opening one or more TCP flows. Note that available bandwidth as described herein is different from raw bandwidth of a link, or bandwidth obtained by a UDP flow. Using this technique, if a certain edge e in a network path is identified to be the bottleneck link, it is annotated with "=B" as edge e is the constraining link in the end-to-end path and the bandwidth of the end-to-end path is B. Our goal is to identify the bottleneck link faced by one or more TCP flows along a network path, without perturbing other traffic sharing the same path. The key challenge in bottleneck identification is that we can only rely on end-to-end measurements to isolate an internal network property.

An interesting approach to the problem, known as BFind [53] was to correlate an increase in the traffic rate of a flow with an increase in the per-hop delay reported along its path by traceroute, and identify the hop showing the maximum increase in delay as the bottleneck link. A key limitation of the approach is that it uses a congestion insensitive UDP flow to ramp up the traffic on a path. Though UDP is the only viable choice when we have control over only one of the endpoints of the path, several undesirable properties result from this choice. Firstly, since the

UDP flow is congestion insensitive, it may push away background traffic from the bottleneck link, especially TCP flows with small windows. This causes the probe flow to "open up" the bottleneck, potentially leading to incorrect identification of the bottleneck. Secondly, as mentioned in Chapter 3, we attempt to annotate a network with the bandwidth available to a TCP flow, assuming that the application will use network-friendly TCP or TFRC flows. A link being the bottleneck for a TCP flow depends upon several complex factors, for instance RTTs and window sizes of the background TCP traffic. Hence, in general, the bottleneck for a UDP flow may not be the bottleneck for a TCP flow. Finally, a collection of UDP probes can be extremely intrusive to other traffic in the network, and may even flood the network.

TReno [45] tries to adapt its probe traffic to network congestion by using the Additive Increase Multiplicative Decrease (AIMD) algorithm. However, TReno uses ICMP packets as probe packets, and hence converges very slowly (since ICMP packets are sent on the slow path of the router).

4.1.1 TCP-based bottleneck estimation

A more desirable solution is to use TCP's own bandwidth ramp-up to induce and detect a bottleneck in the network. Figure 4.1 illustrates our scheme for detecting the location of bottleneck link along a network path. The TCP protocol between a sender and receiver connected over a network path operates by increasing the sending rate, thus probing for available bandwidth in the network. When the TCP protocol detects a packet loss, the sending rate is decreased and the probing process is repeated by increasing the sending rate.

The intuition behind developing a TCP-friendly variant of BFind [53] is derived

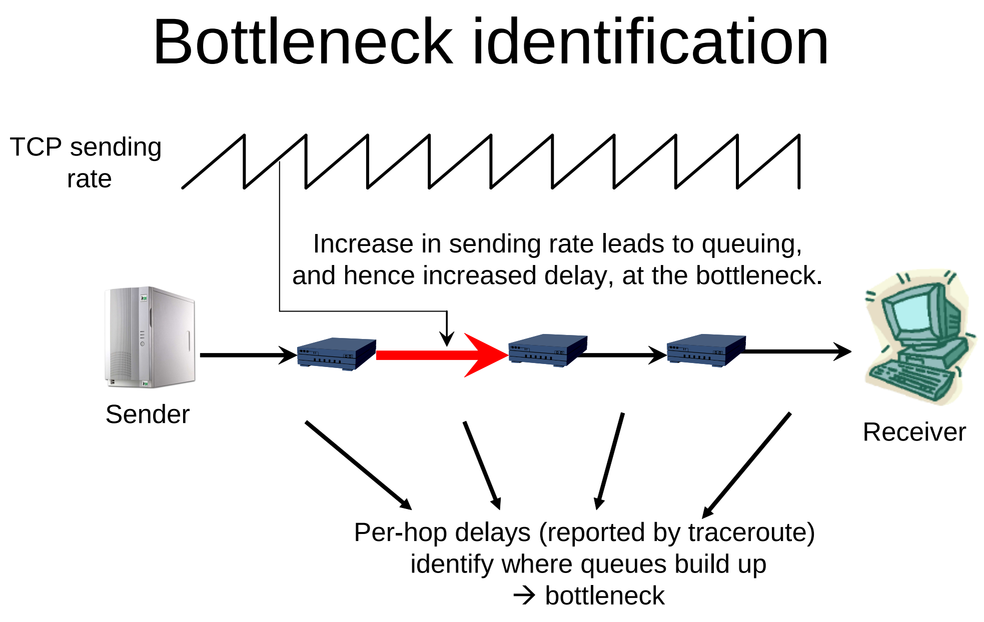

Bottleneck identification
TCP sending rate
Increase in sending rate leads to queuing, and hence increased delay, at the bottleneck.
Sender
Receiver
Per-hop delays (reported by traceroute) identify where queues build up → bottleneck

Figure 4.1: TCP-based identification of the bottleneck link along a path.

from TCP Vegas [46]. TCP Vegas reacts proactively to impending congestion by reacting to an increase in RTT before incurring a loss event. The premise is that there is a stable, detectable increase in queuing delay before a loss occurs. When the sending rate is increased, the delay experienced by packets at each link increases. However, the property of the bottleneck link on a path is that it shows the highest increase in delay. This delay can be measured by running the traceroute utility on the path together with the TCP flow, as traceroute reports the per-hop delay experienced by packets. Thus, our system utilizes traceroute and TCP in conjunction in order to infer the bottleneck link on a network path. We refer to this tool as *Tneck*, which stands for Tcp-friendly bottleNECK estimation.

To implement Tneck, we start by measuring the latency of each link along the

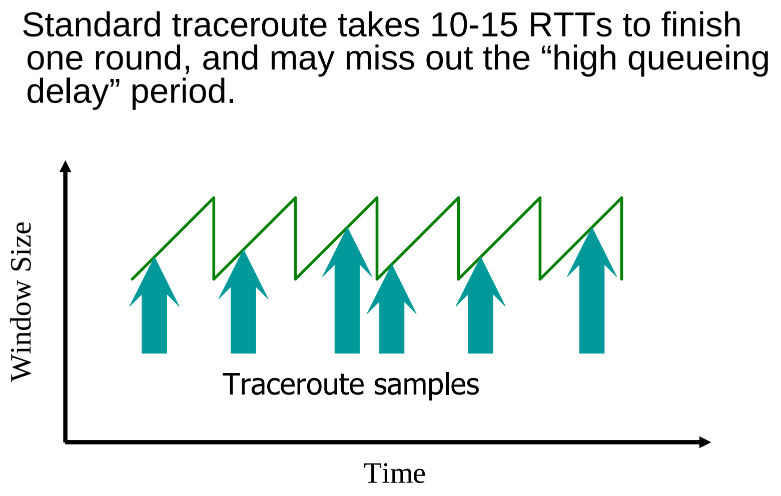

Standard traceroute takes 10-15 RTTs to finish one round, and may miss out the “high queueing delay” period.
Window Size
Traceroute samples
Time

Figure 4.2: This figure shows how using standard traceroute may not provide accurate information about the queues being built at the bottleneck link.

path using traceroute. We call this as *base latency* for the link. We take a number of traceroute samples (typically five), and record the minimum latency on each link as its base latency. Note that base latency intuitively represents propagation delay of the link. Next, we open a TCP connection between the two end-points. As TCP ramps up its congestion window, we start running traceroute to sample the latency on each link. We find the increase in delay on each link as compared to the base latency. This increased delay is primarily caused by the increased queuing delay. We mark the link that shows maximum increase in delay as the bottleneck link. We wait for four consecutive samples during which we see the increased delay before marking a given link as bottleneck.

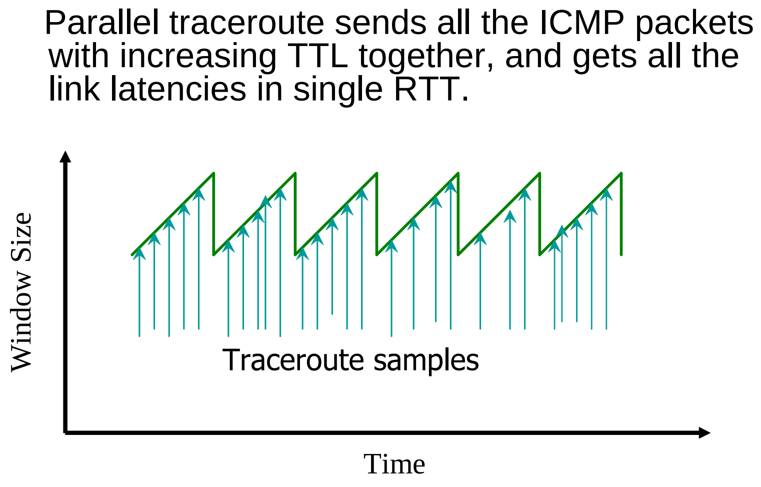

Parallel traceroute sends all the ICMP packets with increasing TTL together, and gets all the link latencies in single RTT.
Window Size
Traceroute samples
Time

Figure 4.3: This figure shows how the modified (parallel) traceroute can capture link latencies very quickly and frequently.

4.1.2 Need for parallel traceroute

As we know from standard queuing theory, significant increase in queuing delay occurs in a narrow range of high utilization (for a very small time interval). Hence, one must respond to the increased delay within a small number of RTTs. Thus, in order to identify bottlenecks in conjunction with TCP, traceroute must be able to sample the increased delay quickly. To get its result, however, standard traceroute can take time proportional to the number of hops in the path times the RTT. It starts by sending ICMP packets with Time-To-Live (TTL) value of 1, waits for the reply, then sends packets with TTL of 2, and so on. Figure 4.2 shows how standard traceroute could miss information about the queues being built at the bottleneck link.

In order to quickly sample the increased queuing delay, we modify standard traceroute to send multiple ICMP packets with increasing TTL values simultaneously. We refer to this as *parallel traceroute.* This allows us to get the per-hop delays in one RTT. Figure 4.3 shows how parallel traceroute provides the sampling frequency needed for bottleneck identification to work with TCP.

4.2 Experimental results

There has been a lot of work done recently on detecting bottlenecks. The work that is most closely related to our tool is BFind [53] and Pathneck [39]. The two key distinguishing features of our tool *Tneck*, as compared to BFind and Pathneck are as follows. First, *Tneck* sends TCP packets as probing traffic, as opposed to UDP traffic. Second, *Tneck* uses parallel traceroute while BFind and Pathneck use standard traceroute.

In order to understand how these fundamental differences manifest themselves in the real world, we implemented *Tneck* on the Linux operating system, and conducted extensive experiments with Tneck over the real Internet using the IBM IntraGRID [63] nodes, and the RON nodes [58] as the endpoints. The IBM IntraGRID is a network of nodes for Grid computing that has over 50 nodes spread all over the world. The RON network consists of 20 nodes all over the U.S. We also performed an extensive set of experiments on the Emulab [56] testbed to *validate our tool* under diverse network conditions: bottleneck due to raw bandwidth, or due to transient congestion, multiple bottlenecks on a path, location of bottleneck changing dynamically with time, bottleneck oscillating between two links, etc.

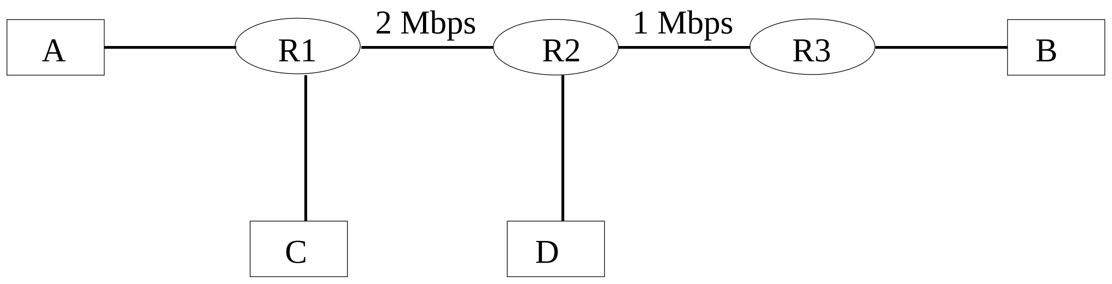


Figure 4.4: Emulab topology used to show that existing tools detect an incorrect bottleneck link if the bottleneck is due to transient congestion.

4.2.1 Bottleneck due to transient congestion

We conjecture that both BFind and Pathneck incorrectly identify the bottleneck link if the bottleneck is due to transient congestion, and not due to raw bandwidth. The intuition being that BFind uses a congestion insensitive UDP flow to ramp up the traffic on a path. To validate this, we setup the topology shown in figure 4.4 on Emulab[56]. Nodes A, B, C and D are end-hosts, while R1, R2 and R3 are intermediate routers. Unless explicitly shown in the figure, all links have bandwidth of 100 Mbps. We open two long-running TCP flows from C to D. A TCP flow between A and B has R1-R2 as the bottleneck link, since 1/3rd of the link bandwidth (0.67 Mbps) is available to all 3 flows. However, running BFind[53] between A and B reports R2-R3 as the bottleneck, because it pushes away the traffic of the 2 TCP flows between C and D on the link R2-R3, occupying more than its fair share of 0.67 Mbps on R2-R3. The Pathneck tool also suffers from a similar problem if it sends its packet train at a rate higher than the available bandwidth.

4.2.2 Effect on background traffic

The fact that *Tneck* opens a TCP flow between the two end-points makes our tool non-intrusive to the background flows. *Tneck* induces much less probe traffic inside the network, as compared to BFind. This also allows us to keep *Tneck* running over a long period of time. We ran both Tneck and BFind over the real Internet, and found that due to the aggressive nature of UDP flows, BFind converged to a bottleneck at a probe bandwidth 3-4 times higher than the TCP-based variant. In many cases, BFind was not able to converge to a bottleneck because either standard traceroute was not able to sample the induced increase in delay, or it pushed away most of the competing background traffic. In fact, during one of our BFind experiments, two of the IBM IntraGRID nodes in Europe were taken off the network by AT&T since their network surveillance system classified the excessive bandwidth being consumed by the UDP traffic as worm-like behavior by the endpoints!

4.2.3 The role of TCP and parallel traceroute

After deploying both *Tneck* and BFind on the real Internet, we found that our tool *Tneck* can identify bottleneck links on paths where BFind does not. This is especially true if the RTT between the two end-hosts is very large, for example between New York and China. We believe that this is primarily due to the fact that *Tneck* uses parallel traceroute. BFind does not see the increased delay for sufficient number of traceroute samples, and hence misses out the bottleneck link. We also observed that many routers which usually respond back to ICMP packets, stop doing so when we send too many UDP packets. Due to this, BFind often does not report any bottleneck link. *Tneck*, however, gets all the ICMP responses back (since it uses TCP packets), and hence identifies a bottleneck link. Even if both *Tneck* and BFind detect the same bottleneck link, our tool *Tneck* converges faster.

4.2.4 Limitations of Tneck

The biggest drawback of *Tneck* is that it requires control at both source and destination. Both BFind and Pathneck require control at only one of the end-points. However, if all the end-points are part of a distributed application (as is the case with our Netmapper system described in Chapter 3), we believe it is reasonable to assume control over both ends.

Tneck would not be able to identify the bottleneck link if the underlying TCP flow is limited by its receive window. Similarly, BFind would not converge if the available bandwidth on the path is more than the maximum rate at which BFind sends its UDP traffic. There are other drawbacks of our bottleneck estimation tool (in general, any tool that relies on the latency information reported by traceroute). Firstly, traceroute sends ICMP packets, and many routers simply ignore all the ICMP packets (or, at least do not send a response back for the ICMP packets whose TTL expires). Due to this, traceroute output does not report latency for some of the links, and hence tools like *Tneck* and BFind do not identify a bottleneck link. Secondly, ICMP packets lie on the slow path of intermediate routers. Note that the physical path taken by ICMP packets is same as that taken by UDP and TCP packets. The only difference is that most routers give lower priority to ICMP packets (i.e., drop ICMP packets with higher probability, or delay the response to ICMP packets). This leads to inaccuracy in the latencies reported by traceroute.

4.3 Conclusions

In this chapter, we described a novel scheme to identify the location of bottlenecks in a network for a distributed networked application. The tool improves upon existing methods [53, 39] for bottleneck identification. Specific contributions over existing schemes include identifying bottlenecks with little to no impact on the regular traffic, and matching TCP's notion of bottleneck (as opposed to, say link-level definitions). We show using experiments (under controlled settings of the Emulab environment [56]) that existing bottleneck identification schemes [53, 39] detect incorrect bottleneck link if the bottleneck is due to transient congestion.

We used the bottleneck identification tool as a key primitive in the network annotation algorithm described in Chapter 3. The applicability of the tool is broad enough that end-users can use it to estimate the performance of the network path to a given destination, while ISPs can use it to quickly locate network problems, or to guide traffic engineering both at the interdomain and intradomain level.

Chapter 5

Conclusions and future work

In this chapter, we summarize our contributions, and point out directions for future research, both in addressing the limitations of this work, and in pursuing new avenues. Network performance is the key determinant of end-to-end performance and user experience for various applications[1, 2, 25, 38, 53]. However, most applications do not have explicit mechanisms to manage the network performance. We endeavored in this thesis to develop techniques for exposing the network as a managed resource to applications without requiring any explicit support from the network elements like routers, gateways, ISPs, etc. As part of this broad initiative, we have built systems to monitor (Netmapper, Tneck) and control (MPAT) the network from end-hosts, and tested them on the real Internet.

In Chapter 2, we described MPAT, the first truly scalable algorithm for fairly providing differential services to TCP flows that share the same bottleneck link. The TCP protocol state at an end-system captures rich information about the state of the network, such as flow's fair share of network bandwidth at its bottleneck, network delays, loss characteristics, and so on. While an individual flow's TCP state information may have limited value, the combined state of a large number of TCP flows can be a rich indicator of the state of the network. Internet servers typically have a very large number of TCP connections active simultaneously. Grouping these connections logically into flows that share a bottleneck link or a path can provide a comprehensive view of the internal network state, at the ends of the network. Managing this aggregate congestion state provides us a level of control within the network, without having control mechanisms implemented inside the network.

Unlike known schemes, our approach preserves the cumulative fair share of the aggregated flows even where the number of flows in the aggregate is large. Specifically, we demonstrated, primarily through experiments on the real Internet, that congestion state can be shared across more than 100 TCP flows with throughput differentials of 95:1. This is up to five times better than differentials achievable by known techniques. Indeed, MPAT scalability is limited only by the delay-bandwidth product of the aggregated flows. With this tool, it is now possible to seriously explore the viability of network QoS through overlay network services.

The next goal is to build some real-world applications that can benefit out of this work. The two killer applications that we have in mind are the enterprise network and grid applications. Companies like IBM need to give more bandwidth to gold-customers as compared to best-effort customers in the enterprise network. Grid applications have lot of background transfers (like file system backup) that need less bandwidth. We would like to look more generally at problems associated with designing an overlay network to provide general QoS service to a wide community of users (the QISP model discussed in Chapter 1).

We would also like to look more at the problem of how to provide differential QoS from a single server. For this, we need to develop some novel techniques for detecting whether multiple connections to a single server share the same bottleneck link in the network or not. Schemes like negative delay/loss correlation have been proposed in the literature, but are not very effective in a generalized setting. For the proposed service differentiation scheme, we intend to analyze, both analytically and empirically, the impact of aggregating flows with widely varying RTTs.

As of now, our system MPAT can provide bandwidth apportionment only among long-running TCP flows. Most of the traffic on the Internet consists of web traffic that has short-running TCP flows. In order to integrate MPAT into real-world applications, we plan to develop mechanisms to benefit short flows. One possible approach we are investigating is to map multiple short flows into a single long flow, or to use a higher value of initial congestion window (known as warm start).

Our current implementation of MPAT requires us to make changes at the server side. We are looking to develop a middle box that could possibly be placed near the gateway router, or firewall box of the site. The goal is that we deploy our MPAT system at these middle boxes, and all the server machines inside a site remain unchanged. This will ease out the deployment of our system in the real world, especially in enterprise settings. Our approach is to have the middle box maintain multiple congestion windows and do the bandwidth apportionment. The middle box could in turn control the individual senders by advertising different value of receiver window size.

In Chapter 3, we described a network mapping and annotation service for distributed applications sensitive to bandwidth availability, latency, or loss. Applications (or resource managers working on their behalf) are expected to use this service to plan their network resource usage and fault response. We expose internal network state (e.g. bandwidth available on the edges) using only end-to-end measurements that are TCP-friendly. Knowledge of this internal state allows trend analysis, network performance debugging, network planning and exploitation of alternate routes, making such a service useful for both network providers and their ISP and enterprise customers.

The system (called Netmapper) takes as input the set of end-points of a distributed application, maps out the network connectivity between them, and annotates each edge (to the extent possible) with available bandwidth. Netmapper deploys end-to-end TCP probes between the end-points of an application to estimate the available bandwidth on various network paths. It then identifies the bottleneck links on these paths using a novel TCP-friendly bottleneck identification technique. Our scheme to identify the location of bottleneck links (described in Chapter 4) improves upon existing methods for bottleneck identification, which detect an incorrect bottleneck link if the bottleneck is due to transient network congestion. Netmapper eventually annotates all the edges whose annotation is possible given the network topology and the available end-points. The end-to-end probes are intelligently planned so that the annotation requires a minimum number of probes. We have built Netmapper, and tested it out on a network of machines that form the RON network. Our system consumes very small amount of bandwidth, and converges fast even with dynamically varying network state.

Currently, we are interfacing three applications with MPAT and Netmapper. One is a peer-to-peer voice conferencing service, which stresses our algorithms both in terms of managing bandwidth and latency for the application. The second application is a distributed network gaming service that needs network-aware placement and mapping of servers to clients. The third one is a distributed data mining application in grid computing. All these studies are being driven by traces from production conferencing and gaming services. The characteristics of these applications are significantly different, allowing us to demonstrate the broad applicability of MPAT and Netmapper as part of a network performance management service for different kinds of applications.

BIBLIOGRAPHY

[1] V. Padmanabhan, *Addressing the Challenges of Web Data Transport*, Ph.D. Thesis, University of California at Berkeley, Sept 1998. http://www.cs.berkeley.edu/ padmanab/phd-thesis.html

[2] H. Balakrishnan and H. S. Rahul and S. Seshan, *An Integrated Congestion Management Architecture for Internet Hosts*, in Proceedings of ACM SIGCOMM, pages 175-187, Sept 1999.

[3] J. Crowcroft and P. Oechslin *Differentiated End-to-End Internet Services using a Weighted Proportional Fair Sharing TCP*, ACM Computer Communication Review, 28(3):53-69, July 1998.

[4] V. Padmanabhan and R. Katz, *Addressing the Challenges of Web Data Transport*, Unpublished, Jan 1998. http://www.research.microsoft.com/ padmanab/papers/thesis-paper.pdf

[5] D. Andersen and H. Balakrishnan and M. Kaashoek, and R. Morris, *Resilient Overlay Networks*, in Proceedings of 18th ACM SOSP, Oct 2001.

[6] V. Padmanabhan, *Coordinating Congestion Management and Bandwidth Sharing for Heterogeneous Data Streams*, in Proceedings of NOSSDAV'99, Basking Ridge, NJ.

[7] P. Gevros and F. Risso and P. Kirstein, *Analysis of a method for differential TCP service*, in Proceedings of IEEE GLOBECOM, pages 1699-1708, Dec 1999.

[8] L. Subramanian and I. Stoica and H. Balakrishnan and R. Katz, *OverQoS: An Overlay Based Architecture for Enhancing Internet QoS*, First Symposium on Networked Systems Design and Implementation (NSDI), March 2004.

[9] J. Touch, *TCP Control Block Interdependence*, RFC-2140, April 1997.

[10] D. Andersen and D. Bansal and D. Curtis and S. Seshan and H. Balakrishnan, *System Support for Bandwidth Management and Content Adaptation in Internet Applications*, in Proceedings of the 4th Symposium on Operating Systems Design and Implementation, pages 213-226, Oct 2000.

[11] 3rd Generation Partnership Project 2 (3GPP2), *Quality of Service: Stage 1 Requirements,* 3GPP2 S.R0035, www.3gpp2.org

[12] A. Bestavros and O. Hartmann, *Aggregating Congestion Information Over Sequences of TCP Connections*, In BUCS-TR-1998-001, Boston University, Dec 1998.

[13] L. Berger, *RSVP over ATM Implementation Guidelines*, RFC 2379, Aug 1998.

[14] L. Zhang and S. Deering and D. Estrin and S. Shenker and D. Zappala, *RSVP: A new resource ReSerVation Protocol*, IEEE Network, 7:8-18, Sept 1993.

[15] R. Braden and D. Clark and S. Shenker, *Integrated services in the Internet architecture: an overview*, RFC 1633, June 1994.

[16] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss, *An architecture for differentiated services*, RFC 2475, Oct 1998.

[17] E. Rosen and A. Viswanathan and R. Callon, *A Proposed Architecture for MPLS*, Internet Draft, draft-ietf-mpls-arch-00.txt, July 1997.

[18] A. Akella and S. Seshan and H. Balakrishnan, *The Impact of False Sharing on Shared Congestion Management*, in Proceedings of ACM SIGCOMM Computer Communication Review, Jan 2002.

[19] J. Padhye and J. Firoiu and J. Kurose, *Modelling TCP Throughput: A Simple Model and its Empirical Validation*, in Proceedings of ACM SIGCOMM, pages 303-314, Aug 1998.

[20] J. Semke and J. Mahadavi and M. Mathis, *Automatic TCP Buffer Tuning*, in Proceedings of ACM SIGCOMM, pages 315-323, Aug 1998.

[21] H.T. Kung and S.Y. Wang, *TCP Trunking: Design, Implementation and Performance*, in Proceedings of 7th International Conference on Network Protocols (ICNP'99), pages 222-231, 1999.

[22] S. Savage, *Sting: a TCP-based network measurement tool*, in Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), Oct 1999.

[23] S. Floyd and M. Handley and J. Padhye and J. Widmer, *Equation-based Congestion Control for Unicast Applications*, in Proceedings of ACM SIGCOMM, pages 43-56, Aug 2000.

[24] P. Pradhan and T. Chiueh and A. Neogi, *Aggregate TCP congestion control using multiple network probing*, in Proceedings of ICDCS 2000, pages 30-37, April 2000.

[25] H. Balakrishnan and V. Padmanabhan and S. Seshan and M. Stemm and R. Katz, *TCP behavior of a busy web server*, in Proceedings of IEEE INFOCOM, pages 252-262, March 1998.

[26] A. Venkataramani and R. Kokku and Mike Dahlin, *TCP NICE: A mechanism for background transfers*, in Proceedings of the 5th Symposium on Operating Systems Design and Implementation, pages 329-344, Dec 2002.

[27] C. Jin, D.X. Wei and S.H. Low, *FAST TCP: Motivation, Architecture, Algorithms, Performance*, to be published in Proceedings of IEEE Infocom, March 2004.

[28] *The Daytona User-Level TCP Stack*, Reference hidden for anonymity.

[29] D. Chiu and R. Jain, *Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks*, Journal of Computer Networks and ISDN Systems, 17(1):1-14, June 1989.

[30] V. Jacobson, *Congestion Avoidance and Control*, in Proceedings of ACM SIGCOMM, pages 314-329, Aug 1988.

[31] H. Y. Hsieh and K.H. Kim and R. Sivakumar, *On Achieving Weighted Service Differentiation: an End-to-end Perspective*, in Proceedings of IEEE IWQoS, June 2003.

[32] P. Gevros, *Internet Service Differentiation using Transport Options: the case for policy-aware congestion control*, in ACM Workshop on Revisiting IP QoS (RIPQoS), Aug 2003.

[33] B. Davie, *Deployment experience with Differentiated Services*, in ACM Workshop on Revisiting IP QoS (RIPQoS), Aug 2003.

[34] P. Hurley and J.-Y. L. Baudec and P. Thiran and M. Kara, *ABE: Providing a Low-Delay Service with Best Effort*, IEEE Network Magazine, Special Issue on Control of Best-Effort Traffic, 15(3):60-69, May 2001.

[35] C. Dovrolis and P. Ramanathan, *A case for Relative Differentiated Services and the Proportional Differentiation Model*, IEEE Network, 13(5):26-34, Sept 1999.

[36] T. Nandagopal and K.W. Lee and J. R. Li and V. Bharghavan, *Scalable Service Differentiation Using Purely End-to-End Mechanisms: Features and Limitations*, in Proceedings of IEEE IWQoS, June 2000.

[37] D. Lin and R. Morris, *Dynamics of Random Early Detection*, in Proceedings of ACM SIGCOMM, pages 127-137, Sept 1997.

[38] Y. Chu, S. Rao, S. Seshan and H. Zhang, *Enabling Conferencing Applications on the Internet Using an Overlay Multicast Architecture*, SIGCOMM 2001.

[39] N. Hu, L. Li, Z. Mao, P. Steenkiste, J. Wang, *Locating Internet Bottlenecks: Algorithms, Measurements, and Implications*, Sigcomm 2004.

[40] N. Miller and P. Steenkiste, *Collecting Network Status Information for Network-Aware Applications*, INFOCOM 2000.

[41] S. Ganguly, A. Saxena, R. Izmailov, S. Datta and S. Roy, *Path Induced Construction of Available Bandwidth Map for an Overlay Network*, unpublished.

[42] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt and L. Zhang, *IDMaps: A global internet host distance estimation service*, IEEE Transactions on Networking, Oct 2001.

[43] Y. Zhang, N. Duffield, V. Paxson and S. Shenker, *On the constancy of internet path statistics*, IMW 2001.

[44] Y. Chen, D. Bindel, H. Song and R. H. Katz, *An Algebraic Approach to Practical and Scalable Overlay Monitoring*, SIGCOMM 2004.

[45] M. Mathis and J. Mahdavi, *Diagnosing Internet Congestion with a Transport Layer Performance Tool*, INET 1996.

[46] L. Brakmo, S. O'Malley, and L. Peterson, *TCP Vegas: New techniques for congestion detection and avoidance*, SIGCOMM 1994.

[47] J. Jannotti, D. Gifford, K. Johnson, F. Kaashoek, and J. O'Toole, *Overcast: Reliable Multicasting with an Overlay Network*, OSDI 2000.

[48] M. Singh, P. Pradhan and P. Francis, *MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS*, ICNP 2004.

[49] D. Clark, J. Wroclawski, K. Sollins and R. Braden, *Tussle in Cyberspace: Defining Tomorrow's Internet*, SIGCOMM 2002.

[50] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, S. Surana, *Internet Indirection Infrastructure*, SIGCOMM 2002.

[51] Y. Chen, D. Bindel, H. Song and R. H. Katz, *An Algebraic Approach to Practical and Scalable Overlay Monitoring*, SIGCOMM 2004.

[52] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, *Measuring ISP Topologies with Rocketfuel*, Sigcomm 2002.

[53] A. Akella, S. Seshan and A. Shaikh, *An Empirical Evaluation of Wide-Area Internet Bottlenecks*, IMC 2003.

[54] N. Hu, L. Li, Z. Mao, P. Steenkiste, J. Wang, *Locating Internet Bottlenecks: Algorithms, Measurements, and Implications*, SIGCOMM 2004.

[55] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt and L. Zhang, *IDMaps: A global internet host distance estimation service*, IEEE Transactions on Networking, Oct 2001.

[56] Emulab test-bed. http://www.emulab.net

[57] Planetlab test-bed. http://www.planet-lab.org

[58] RON test-bed http://www.datapository.net

[59] http://www.akamai.com

[60] A. Akella, B. Maggs, S. Seshan, A. Shaikh and R. Sitaraman, *A Measurement-Based Analysis of Multihoming*, SIGCOMM 2003

[61] S. Savage, *Sting: a TCP-based network measurement tool*, USITS 1999.

[62] S. Chen, K. Nahrstedt, *An Overview of Quality-of-Service Routing for the Next Generation High-Speed Networks: Problems and Solutions*, IEEE Network Magazine, Dec 1998.

[63] https://intragrid.webahead.ibm.com

[64] M. Kodialam, T. Lakshman and S. Sengupta, *Online Multicast Routing with Bandwidth Guarantees: A New Approach using Multicast Network Flow*, SIGMETRICS 2000.

[65] T. Corman, C. Leiserson, and R. Rivest. Introduction to Algorithms. MIT Press, Cambridge, MA, 1990.