ABSTRACT

Title of Dissertation: THE MEASUREMENT MANAGER:
MODULAR AND EFFICIENT
END-TO-END MEASUREMENT SERVICES

Pavlos Papageorgiou, Doctor of Philosophy, 2008

Directed by: Associate Professor Michael Hicks
Department of Computer Science
Department of Electrical and Computer Engineering

End-to-end network measurement is used to improve the precision, efficiency, and fairness for a variety of Internet protocols and applications. Whether streaming media files, constructing an overlay or picking a candidate server to download from, applications need to provide a good user experience. What constitutes a good user experience differs for each application but what is common is the need to discover and adapt to the current conditions of the network path. This is especially important since, due to the design of the Internet, network routers do not provide any feedback about these network properties.

Measurement is typically performed in one of three ways: (1) *actively*, by injecting specially crafted probe packets into the network, (2) *passively*, by observing existing data traffic, and (3) *customized*, where applications use use their own traffic to perform customized measurements. All current approaches suffer from drawbacks. Passive techniques are efficient but are constrained by the shape of the existing traffic, limiting the speed and accuracy of their measurements. Active techniques are faster, more accurate and more flexible but impose a significantly higher overhead by competing with applica-

tions for bandwidth. And finally, custom techniques combine flexibility with efficiency, but are so tightly coupled with each application that they are not reusable.

To address these shortcomings, we present the *Measurement Manager*, a practical, modular, and efficient service for performing end-to-end network measurements between hosts. Our architecture introduces a new *hybrid* approach to network measurement, where applications can pool together their data packets to be reused as padding inside network probes in a transparent and systematic way. We achieve this through the *Measurement Manager Protocol* (MGRP), a new transport protocol for sending probes that combines data packets and probes on the fly. In MGRP, active measurement algorithms specify the probes they wish to send using a *Probe API* and applications allow MGRP to use data from their own packets to fill the otherwise wasted probe padding. The ability of MGRP to *piggyback any data packet* on *any probe* is pivotal in making our measurement system unique in the sense that any measurement algorithm can now be written *as if active*, but implemented *as if passive*.

We have implemented the *Measurement Manager* inside the Linux kernel and have adapted existing applications and active measurement tools to use our system. Through experimentation we provide detailed empirical evidence that piggybacking data packets on measurement probes is not only feasible but improves source and cross traffic as well as the performance of measurement algorithms while not affecting their accuracy. We show that the Measurement Manager is an architecture with broad applications that can be used to build a generic *measurement overlay network* as well as expanding the solution space for estimation algorithms, since every application packet can now act as a potential probe.

THE MEASUREMENT MANGER: MODULAR AND EFFICIENT END-TO-END MEASUREMENT SERVICES

by

Pavlos Papageorgiou

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Michael Hicks, Chair/Advisor
Professor Mark A. Shayman
Professor Richard J. La
Dr. Mehdi Kalantari Khandani
Professor Rance Cleaveland

Table of Contents

List of Tables v

List of Figures vi

List of Abbreviations viii

1 Introduction 1
- 1.1 Motivating Scenario: Dynamic Adaption of Media Streams 2
- 1.2 Problem Description 5
- 1.3 The Measurement Manager 6
- 1.4 Contributions 9

2 Active Network Measurement Techniques 12
- 2.1 The *probe group* primitive 13
- 2.2 Examples of Active Measurement Tools 14
  - 2.2.1 Pathload (available bandwidth) 14
  - 2.2.2 Yaz (available bandwidth) 17
  - 2.2.3 Pathchirp (available bandwidth) 18
  - 2.2.4 Pathrate (capacity) 20
  - 2.2.5 Badabing (loss) 21

3 The Measurement Manager 24
- 3.1 Probe API 26
  - 3.1.1 Probe Transactions 26
  - 3.1.2 Receivers 30
- 3.2 MGRP Piggybacking 31
  - 3.2.1 Piggybacking by Example 31
  - 3.2.2 Piggybacking Effects on Application Data 34
  - 3.2.3 Piggybacking Effects on Measurement Tools 36
    - 3.2.3.1 Pathload 37
- 3.3 Piggybacking Policy Variations 39
  - 3.3.1 Standalone Packets 40
  - 3.3.2 Duplicate Piggybacked Packets 40
  - 3.3.3 Partial Piggybacking 41

4 The MGRP Implementation 42
- 4.1 The MGRP Linux Kernel Module 42
- 4.2 MGRP Header 43
- 4.3 Probe Generation 45
- 4.4 The Payload Buffer 47
- 4.5 Piggybacking 49
  - 4.5.1 TCP with MGRP Piggybacking 51
  - 4.5.2 MGRP Piggybacking Statistics 52

4.6 Fragmentation and Reassembly . . . 53
4.6.1 The Reassembly Bucket . . . 55
4.7 MGRP Parameters . . . 59
4.7.1 For tools that send probes . . . 59
4.7.2 For applications that contribute payload . . . 60
4.7.3 For global MGRP behavior . . . 61

5 Experimental Evaluation 62
5.1 Experimental Setup . . . 62
5.1.1 Source traffic . . . 63
5.1.2 Cross traffic . . . 64
5.1.3 Probe Transactions . . . 65
5.1.3.1 Packet Trains . . . 66
5.2 Results . . . 67
5.2.1 STEP Experiment . . . 69
5.2.2 FIXED-5 Experiment . . . 75
5.2.3 WEB Experiment . . . 79
5.3 Discussion . . . 83

6 Case Study: MediaNet Overlay 86
6.1 MediaNet . . . 87
6.1.1 Active Measurement with MGRP . . . 91
6.2 Estimation Overlay . . . 92
6.3 MediaNet Experiments . . . 93
6.3.1 Experimental Setup . . . 93
6.3.2 Original MediaNet . . . 95
6.3.2.1 How Medianet Adaptation Works . . . 98
6.3.3 MediaNet with Active Measurement . . . 99
6.3.3.1 Summary of Results . . . 100
6.3.3.2 Using Original Pathload (pSLOW) . . . 104
6.3.3.3 Using Original Pathload (pSLOW) with MGRP . . . 107
6.3.3.4 Using Aggressive Pathload (pFAST) . . . 109
6.4 MediaNet lessons for the Estimation Overlay . . . 112

7 Related Work 114
7.1 Network measurement tools . . . 115
7.2 End-to-end Measurement Services . . . 116
7.3 Piggybacking measurements . . . 119
7.4 Measurement in TCP . . . 123
7.5 Passive Measurement . . . 124

8 Future Work 125
8.1 An Estimation Service . . . 125
8.2 Applications of the Measurement Manager . . . 128
8.2.1 Optimizing File Downloads . . . 128

8.2.1.1 BitTorrent 128
8.2.1.2 Download Managers 130
8.2.2 Optimizing TCP 131
8.2.2.1 Adaptive TCP 132
8.3 New Probing Paradigms 133

9 Conclusions 136

Bibliography 139

List of Tables

3.1 Details that MGRP returns about each probe transaction it sends. 37

4.1 Error in generating gaps between probes in MGRP 47
4.2 A view of MGRP's reassembly hash table in operation 58

5.1 Parameters for packet train experiments 67

6.1 MPEG frame distribution 95
6.2 Increase of decoded MPEG frames/second with MGRP 101
6.3 Increase of MPEG streaming rate with MGRP 102
6.4 Occurence of decoding failures in MediaNet with MGRP 103

List of Figures

1.1 The Measurement Manager 7

2.1 An Example of Pathload Operation 15
2.2 Burst Profile of one Pathload Run 16
2.3 Burst Profile of a Yaz session 18
2.4 Burst Profile of Pathchirp 19
2.5 Burst Profile of a Pathrate session 21
2.6 Burst Profile of a Badabing session 22

3.1 Position of MGRP in the Network Stack 25
3.2 The life of a probe transaction in MGRP 27
3.3 Pseudo code of a probe transaction in MGRP 28
3.4 Key differences between MGRP and UDP socket APIs 29
3.5 Pseudo code for receiving probes in MGRP 31
3.6 A step-by-step example of MGRP operation 32
3.7 Examples of adjusting pathload estimates for piggybacking 39

4.1 The position of MGRP in the Linux network stack 43
4.2 The MGRP Header 44
4.3 How MGRP extracts Transport Units 45
4.4 Examples of Piggybacked Packets 49
4.5 Examples of Fragmented Packets 50
4.6 TCP socket option for applications to opt-in to MGRP 52
4.7 The MGRP Fragmentation Header 54
4.8 The Reassembly Process 57

5.1 Experiment Topology 63
5.2 The three different types of cross traffic 64
5.3 STEP: Timeseries plot, with pk2 probes 69
5.4 STEP: Timeseries plot, with pFAST probes 70
5.5 STEP: Average per-second throughputs 70
5.6 STEP: Source throughputs while running packet train probes pk1, pk2, pk3 . . . 71
5.7 STEP: Source throughputs while running Pathload measurements 71
5.8 STEP: Completion times for pathload measurements. 72
5.9 Pathload accuracy plots for STEP experiment 73
5.10 FIXED-5: Timeseries plot: pk2 probes 75
5.11 FIXED-5: Average per-second throughputs. 75
5.12 FIXED-5: Combined source throughputs for pk1, pk2, pk3 76
5.13 FIXED-5: Source throughputs while running packet train probes pk1, pk2, pk3 . 76
5.14 FIXED-5: Source throughputs while running Pathload measurements 76
5.15 FIXED-5: Completion times for pathload measurements. 77
5.16 Pathload accuracy plots for FIXED experiment 78
5.17 WEB Average per-second throughputs 79

5.18 WEB: Source throughputs while running packet train probes pk1, pk2, pk3 . . . 80
5.19 WEB: Source throughputs while running Pathload measurements 80
5.20 WEB: Completion times for pathload measurements. 81
5.21 Pathload accuracy plots for WEB experiment 82
5.22 Effect of the piggybacking ratio on losses of the source traffic 84

6.1 The basic operation of MediaNet 89
6.2 MediaNet experiment 94
6.3 Cross traffic for the MediaNet experiment 95
6.4 Original MediaNet Operation Experiment 96
6.5 Medianet using Active Measurement but no MGRP (pathload SLOW) . . 105
6.6 MediaNet experiment with three MPEG streams using mgrp10/pSLOW . 108
6.7 MediaNet experiment with three MPEG streams using mgrpOFF/pFAST 110
6.8 MediaNet experiment with three MPEG streams using mgrp10/pFAST . 111

8.1 Adding an Estimator Service to the Measurement Manager 126

List of Abbreviations

CBR	Constant Bit Rate
CPU	Central Processing Unite
GS	(MediaNet) Global Scheduler
IP	Internet Protocol
LAN	Local Area Network
LKM	Linux Kernel Module
LS	(MediaNet) Local Scheduler
Mbps	Megabits Per Second
MGRP	Measurement Manager Protocol
MTU	Maximum Transmission Unit
OWD	One Way Delay
RTO	Retransmission Timeout
RTT	Round Trip Time
TCP	Transmission Control Protocol
TTL	Time To Live
TUN	(MGRP) Transport Unit
UDP	User Datagram Protocl

Chapter 1

Introduction

End-to-end measurement is an integral part of Internet applications and transport protocols. Whether streaming media files, constructing an overlay or picking a candidate server to download from, applications need to provide a good user experience. What constitutes a good user experience differs for each application but what is common is the need to discover and adapt to the current conditions of the network path.

The network properties of interest range from simple estimates of round trip time (RTT), latency, jitter (i.e., the uniformity of times between deliveries) and loss rate, to more involved estimates of available bandwidth, path capacity, and the detection of congestion and bottleneck links. Given network characteristics such as these, a video streaming application may prefer paths with low jitter and low loss rate so that its streams can be played with little buffering. For real-time video and audio-conferencing, end-to-end latency may also be important. On the other hand, a file sharing service like BitTorrent [1] may prefer paths that maximize the bandwidth available between a client and file-serving nodes to those that minimize latency and jitter.

Due to the design of the Internet, network routers do not provide any feedback about these network properties. Applications and transport protocols need to discover current network conditions on their own through end-to-end network measurement. Broadly speaking, existing techniques either *passively* observe existing traffic, or *actively* inject

probe packets to see how the network responds [2].

Transport protocols use passive techniques extensively. For example, TCP [3], the Transmission Control Protocol, estimates the Round-Trip Time (RTT) of a network path using its own packets. It then uses the RTT estimate to to perform Additive Increase Multiplicative Decrease (AIMD) probing to set its congestion window accordingly [4]. As another example, RTP/RTCP [5] monitors the delay characteristics of the packets of time-sensitive applications.

Active techniques are typically used by standalone tools or application-level services. For example, *pathload* [6] and *pathrate* [7] estimate available bandwidth and bottleneck capacity, respectively, by sending carefully-constructed bursts of packets into the network and observing the spacing and loss rates of those packets at the receiver end. As another example, the Resilient Overlay Network (RON), regularly sends *ping* packets between its nodes for establishing latency and connectivity characteristics between its nodes [8].

While both active and passive measurement techniques have their place, we find that neither class of techniques on its own is wholly satisfactory. To motivate why, we consider an example scenario: Internet media streaming.

1.1 Motivating Scenario: Dynamic Adaption of Media Streams

Consider applications that transfer voice and video over the Internet. According to the New York Times, 31% of AT&T's network is used by media streaming applications and that video traffic has now surpassed peer-to-peer traffic and is only second to web

traffic [9]. The popularity of video streaming sites, live streaming of concerts, sporting events, and presidential debates, and real-time person-to-person voice and video traffic have demonstrated the need for maintaining reasonable quality of service despite changing network conditions.

Maintaining a specific level of service for video streaming can become critical in health-related applications, as in the case of Telemedicine. Using the Internet to build Telemedicine applications can be practical [10], but has many challenges and requirements that differ from normal video streaming. For example, an empirical evaluation of using Internet-based Telemedicine in opthalmology revealed that it is more important for the medical provider to have fine-grained control over the video streaming quality of critical frames than maintaining a good average quality [11]. In short, on-demand, customizable quality of service can lead to better diagnoses. It is clear that the requirements of medical professionals making a diagnosis are different than the requirements of home users streaming their favorite movie.

Modern video and VoIP systems include some form of automatic rate determination at the beginning of sessions, but to adapt to variations in network conditions during a session requires some form of network measurement. There are three possible approaches: (1) active techniques that inject single or groups of special packets (probes) into the network; (2) passive techniques that observe existing traffic; (3) custom techniques that shape or alter the application's own traffic to probe the network. However, no current technique is completely satisfactory.

Using *passive measurement*, an application that uses playback buffers (as most video streaming applications do), may observe the rate at which its buffers fill or empty

[12] and keep track of packet and frame loss of its own traffic [13, 14]. Passive observations are particularly useful for signaling that a stream's rate should be reduced, but do little to suggest when its rate can be increased [13, 14], essentially because fixed-rate streams sent at rates below the available bandwidth say little about how much traffic the network can actually support.

Using *active measurement*, the application could try to infer available bandwidth by actively probing the network. This way, a stream could be upgraded when sufficient bandwidth becomes available. To do this, the application could use an active measurement tool, such as pathload [6] or pathchirp [15] to probe the available bandwidth, and only move to the higher rate if sufficient bandwidth is available. The drawback here is that the tool's probes compete with the application for bandwidth, which could again hurt quality.

To avoid the negative effects of active measurement probes, the application could implement a *custom measurement* algorithm in which it alters its own traffic to learn about existing conditions. In the simplest case, the application may attempt to periodically send higher-bandwidth, higher-quality data. If additional traffic can be supported, the application immediately takes advantage of this data at no overhead. But if insufficient bandwidth is available, session quality suffers every time the application blindly increases its rate. A more sophisticated approach would be for the application to infer available bandwidth by shaping its own traffic as a measurement tool would. The benefit is that measurement traffic and application traffic do not compete, because they are one and the same, but with the similar accuracy to the active approach. For example, VDN [16] periodically shapes its traffic into trains like those of pathload to infer rough bandwidth characteristics. Skype [17] also shapes its own packets to measure available bandwidth

and loss rate together [13]. The main drawback of these schemes is that they are not modular; measurement algorithms are tightly coupled with the applications that use them.

1.2 Problem Description

In summary, neither passive, active, nor custom measurement techniques are wholly satisfactory, for the following reasons:

1. **Passive measurement is not flexible and not adequate** Passive measurement techniques use existing traffic to infer properties about a network path. These techniques are efficient because they introduce no new traffic, but lack control over the probe sequence. This lack of control limits the network properties that we can estimate, slows down the estimation process and produces estimates that may be less accurate than in the active case. In our example, passive techniques are not able to detect when bandwidth becomes available after network conditions improve.

2. **Active measurement is not efficient and is disruptive** Because active measurement techniques have full control over the probe sequence, they are usually fast to produce an accurate estimate. Moreover, active techniques are modular: if a better tool or algorithm for, say, available bandwidth emerges, applications can easily switch to using the new tool instead of an existing one. However, active tools tend to be more intrusive and less efficient than passive techniques because active probe packets interfere with the traffic we are trying to measure, and themselves contain no useful payload. Thus, while active measurement can detect current network conditions, the cost of doing so may outweigh the benefits.

3. **Custom measurement is not modular** Custom measurement techniques use the application's own traffic to measure the network. This approach is attractive because it combines the best features of active and passive techniques: a customized algorithm is *efficient* when it can reuse application traffic, but potentially more *accurate* because it can schedule that traffic to its advantage. The main drawback of custom algorithms is that they are *not modular*. Custom techniques are often tailored to and tightly coupled with the applications using them: the same application cannot easily be adapted to use another algorithm, and conversely a single measurement tool cannot reuse packets from separate applications.

In effect, current measurement techniques are either fast and accurate but not efficient, or efficient but slower, less accurate and ad hoc. The goal of our research is to create a system for network measurement that, like the best custom approaches, combines the low-overhead of passive probing with the flexibility and accuracy of active probing, but does so in a way that is modular.

1.3 The Measurement Manager

In this dissertation we present the *Measurement Manager*, a kernel-level service for performing network measurements between hosts. Our architecture aims to address the shortcomings of current approaches to network measurement by using a *hybrid approach* that combines the best features of passive, active and custom approaches. The main premise of our approach is to leverage application traffic, like the custom and passive approaches, with the speed, accuracy and modularity of the active approach. The

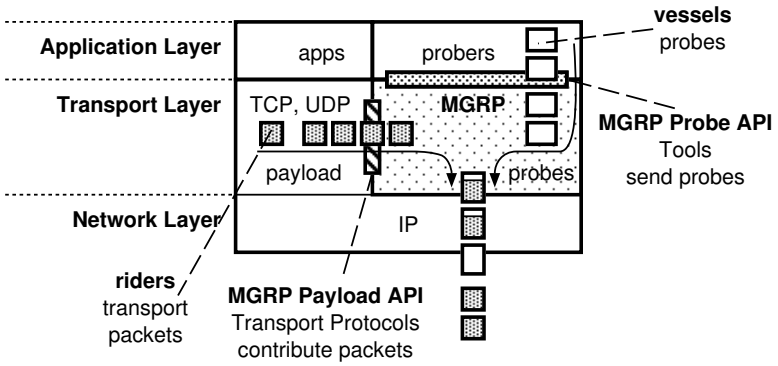


Figure 1.1: Measurement tools specify the characteristics of probe trains to be sent, including whether some portion of a probe packet contains empty padding. This portion may be filled by useful payload, drawn from existing TCP or UDP streams having the same destination as the probes, to lower the probing overhead.

Measurement Manager grew out of our work on merging network measurement with transport protocols [18] and we have presented parts of our system in work that is under submission [19]. The original idea was inspired by the Congestion Manager [20], an end-to-end system that coordinates congestion information between Internet hosts.

Figure 1.1 shows the overview of the Measurement Manager architecture. The main novelty of the Measurement Manager is that it enables network probes to reuse application traffic transparently and in a systematic way. We achieve this through the *Measurement Manager Protocol* (MGRP), an in-kernel service that combines probes and data packets. In MGRP, active measurement algorithms specify the probes they wish to send using the *Probe API*. With this API they specify the per-probe header, the payload size, and the time to wait between probe transmissions. The kernel then sends the probes according to the specification. However, rather than filling probe payloads with empty padding, as is normally done with active tools, the kernel attempts to fill these payloads with data from application packets having the same destination as the probes. The ability of MGRP to *piggyback any data packet* on *any probe* turns out to be pivotal in making

our measurement system unique in the sense that any measurement algorithm can now be written *as if active*, but implemented *as if passive*.

> The thesis of this dissertation is that the Measurement Manager provides four characteristics that make it superior to existing measurement services and techniques: *flexibility*, *efficiency*, and *modularity*.

To elaborate:

- **Flexibility** The Measurement Manager enables specifying network measurements using a Probe API that is sufficient to implement a range of active measurement algorithms accurately. Moreover, the API provides tools with flexibility to control how these probes are scheduled. For example, if no payload is found to fill the empty padding, probes go out empty. Entities that generate probes can determine whether, or for how long, their probes wait for piggybacking.

- **Efficiency** MGRP allows active measurements to be more efficient by piggybacking contributed application packets on empty probes. This reduction in probing overhead has three benefits. First, it helps the *source traffic* by reducing contention between probes and application packets, which makes more path bandwidth available for useful data. Second, it helps any *cross traffic* by again reducing interference and link contention from active measurements. Finally, the bandwidth savings enables measurement algorithms to be more aggressive in their approach, making measurements quicker to complete, and potentially more accurate.

- **Modularity** The Measurement Manager permits a clear separation of the imple-

mentation of measurement tools and services from applications that use them. It is not tied to any specific transport protocol, application, or estimation technique. Any tool that is built to use the Measurement Manager can take advantage of available packets from any number of applications, and any application can take advantage of active measurements collected using probes that had application traffic piggybacked.

We view the work in this dissertation as laying a foundation for a proper integration of network measurement with network services for a broad range of applications. In particular, the Measurement Manager makes it easier for applications to use network measurement, either by using existing techniques or by creating new schemes when they need highly customized measurements. When a new estimation algorithm is needed, the Measurement Manager makes it easier to develop and evaluate the new algorithm by providing a two-step development process. First, the algorithm is designed in a standalone *active-like* fashion without worrying about overhead. Then, once a proper algorithm has been found, we can enable probe reuse and minimize the overhead that it incurs. Moreover, the Measurement Manager opens the possibility of building a network measurement service to which applications transparently contribute their own traffic, while collectively taking advantage of the measurements performed. As more paths are covered by the service, more applications can benefit.

1.4 Contributions

In summary, this dissertation makes the following contributions:

- We perform an analysis of existing active measurement tools and provide a detailed profile of the types of probing currently used. Based on our analysis we derive a generic, suitably-expressive Probe API that the Measurement Manager implements. (Chapter 2).

- Our main contribution is to design and build a practical, modular and efficient architecture that provides a systematic way for applications to pool together their data packets and use them as network probes (Chapter 3).

- We design a new transport protocol, the Measurement Manager Protocol (MGRP), that transparently combines measurement probes and application data and implement it inside the Linux kernel (Chapter 3 and Chapter 4).

- Through micro-benchmarks we provide detailed, experimental evidence that piggybacking data packets on measurement probes is not only feasible but improves source and cross traffic as well as the performance of measurement algorithms while not affecting their accuracy (Chapter 5).

- We show that the Measurement Manager can be used to build a *measurement overlay network* that runs alongside existing applications. Applications let the measurement overlay use their traffic as piggybacking inside probes and in turn the overlay provides the applications with measurement estimates. We show how MediaNet [21], an overlay network for distributing streaming media at various quality-of-service levels, can take advantage of measurements, provided by the measurement overlay, to maximize the streaming rate of its media streams (Chapter 6).

- We show that the Measurement Manager is an architecture with many applications. We show how the measurement overlay we built for Chapter 6 can be extended to become an integral part of the Measurement Manager, providing a broad range of estimation services. We also explore how the Measurement Manager expands the solution space for estimation algorithms, since every application packet can now act as potential probe (Chapter 8).

In the process of doing this work, we also developed a series of tools for testing and analyzing network measurement tools, we built infrastructure for conducting distributed experiments and we automated the data collection process so that we could visualize network conditions in real time.

Chapter 2

Active Network Measurement Techniques

Active measurement techniques can be quite accurate and provide timely estimates, but are not efficient because they generate empty probes that consume network bandwidth. In spite of the higher overhead, applications benefit from active measurement because they can be proactive and gracefully adapt to changing network conditions.

For example, a video streaming application can seamlessly switch to a lower rate without any interruption when conditions deteriorate, or increase the quality of the video stream when bandwidth becomes available again. If the same application used only passive measurement to infer properties of its own traffic, it could react, possibly abruptly, to deteriorating conditions such as loss or lower bandwidth but it could not take advantage of improving network conditions. So applications are better off using active measurement even if that means that they are injecting additional traffic in the network. We demonstrate this in our case study of the MediaNet Streaming Overlay in Chapter 6.

In this chapter we describe examples of existing active measurement tools to motivate the design of MGRP. The objective of this dissertation is not to build a better tool or a better estimation algorithm. Our objective is to build a common probing infrastructure that active measurement tools can use so that they incur passive-like overhead. To that purpose, we demonstrate that since most tools use a common probing primitive, and our system implements this primitive very efficiently, then most existing tools can be built on

top of MGRP and incur much less overhead with minimal changes.

2.1 The *probe group* primitive

Crovella and Krishnamurthy [2] (section 5.3) provide an extensive classification of active measurement tools based on the techniques used to estimate network properties. But our insight is that aside from the inference algorithms, tools use a common basic primitive when sending probes. This primitive is to inject probes into the network in groups with the following characteristics: (a) probe groups are generated independently, (b) the number of probes in each group can range from 1 or 2 to tens or hundreds of probes, (c) each probe has a specific packet size (not necessarily the same), (d) probes are spaced apart with precise timing, (e) probes can be normal UDP packets or special packets (such as ICMP or TTL-limited packets) that incite responses from intermediate routers. In Chapter 3 we present our Probe API which presents an implementation of this *probe group* primitive.

The probe group primitive can model tools that send packet pairs (pathchar [22]), fixed-gap packet trains (pathload [6], yaz [23]), variable-gap packet trains (pathchirp [15]), back-to-back packet trains (pathrate [7], badabing [24]), and recursive packet trains with TTL-limited probes (pathneck [25]). Additionally, this primitive can model the APIs used by a number of measurement services to send probes, such as MAD [26], Periscope [27], Scriptroute [28], precision probing [29] and pktd [30]. We briefly describe these measurement services when we discuss related work in Chapter 7.

2.2 Examples of Active Measurement Tools

This section presents a sampling of active measurement tools (pathload, pathchirp, yaz, pathrate, and badabing) along with their network usage characteristics. We chose these tools in an attempt to cover a large number of the network properties that are usually measured (available bandwidth, capacity, and loss rate), along with the probe techniques that are typically used (fixed-gap packet trains, variable-gap packet trains, and back-to-back packet trains). Most of the tools that we present in this section are tools that require the cooperation of both sides of the end-to-end path. They typically comprise of a *sender* and a *receiver* that send probes between them, usually using UDP, with a possible TCP control channel for exchanging commands and results.

2.2.1 Pathload (available bandwidth)

Pathload [6] is an active measurement tool that estimates the available bandwidth of an end-to-end path. Pathload uses self-induced congestion in the form of *Self-Loading Periodic Streams (SLoPS)* [31] that temporarily build up the queue at the link with the least amount of available bandwidth. Pathload aims at finding the minimum probe rate that causes congestion. It reaches this minimum rate by changing the probe rate in a binary search fashion.

Figure 2.1 shows the operation of pathload in the presence of cross traffic. In this example pathload is running continuously and keeps generating estimates. We see that pathload generates an estimate every 20-30 seconds. Our setup is simple. The only cross traffic present is UDP step traffic, which we show in the top figure. In the bottom figure

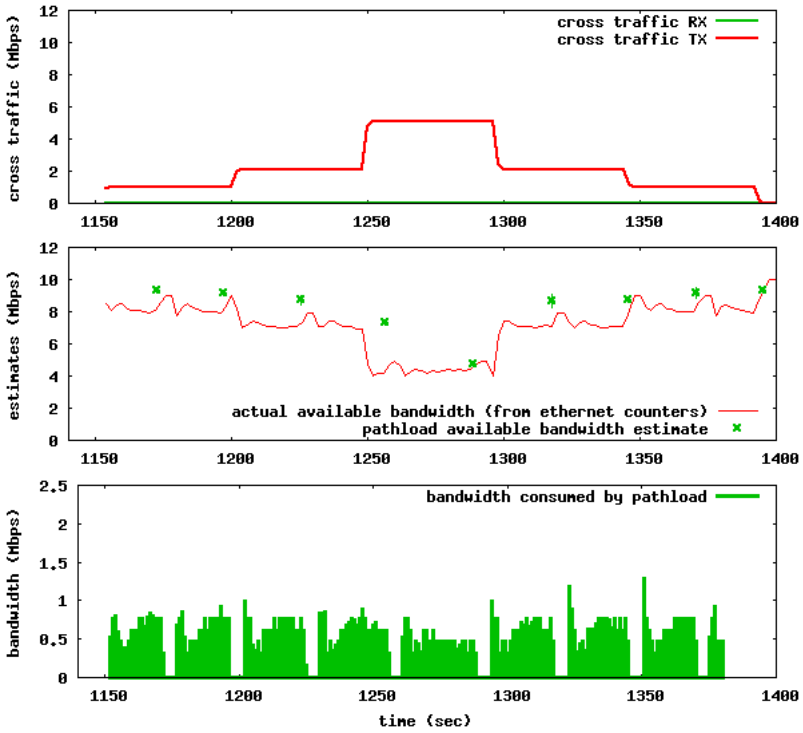


Figure 2.1: An Example of Pathload Operation. Pathload keeps running and generating estimates (the green points). In this case we see that pathload is pretty accurate in estimating the available bandwidth.

we see one green point for every pathload estimate and with the red line we show the actual available bandwidth derived from the Ethernet counters.

Pathload operates in rounds. In each round, it generates trains of probes, called *streams*, that it sends from source to destination. Each stream consists of K probes (default 100) that are equally sized (L bytes) and equally spaced (T sec between successive probes). Pathload sends one fleet of N streams (default 12) in each round. The values of K, L and T remain fixed during a round and correspond to a certain instantaneous probing rate $R = L/T$. Figure 2.2 shows the burst profile of an actual Pathload run. We derived these plots from the packet trace of the example we showed in Figure 2.1. The plots show (a) the number of probes per burst, (b) the average probe packet size and (c) the gap between probes. We can also see how these variables affect the instantaneous

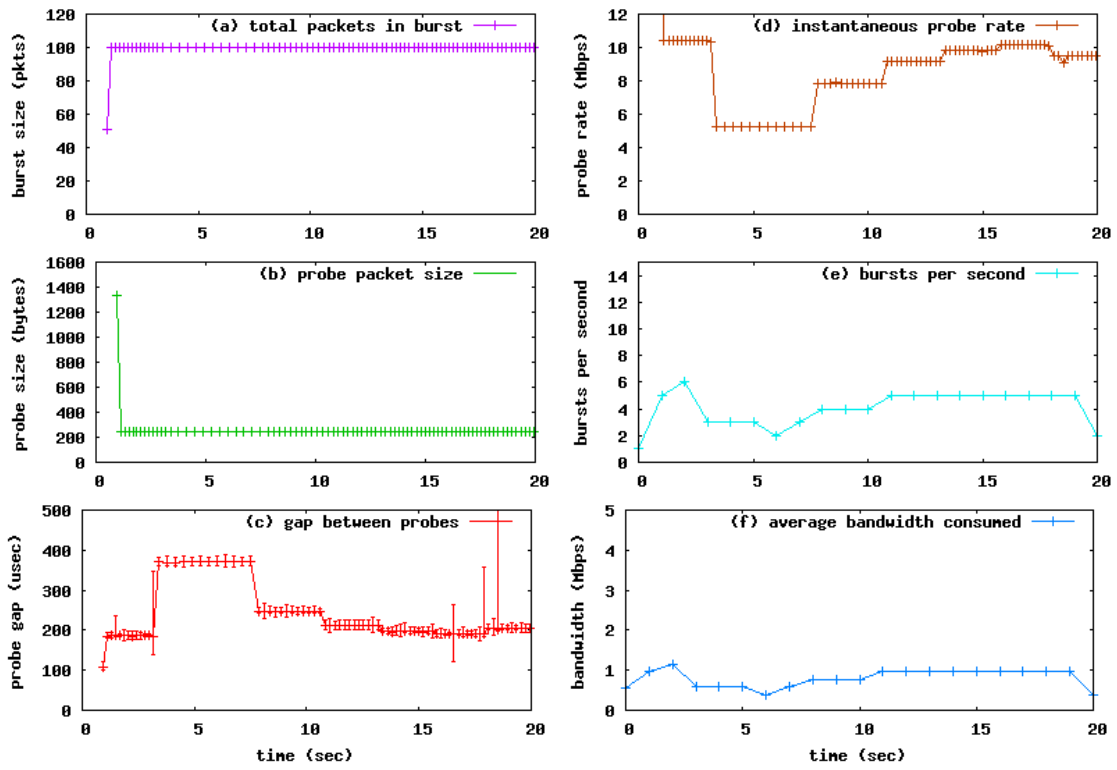


Figure 2.2: Burst Profile of one Pathload Run. These plots have one point for every burst that pathload sends. We see that pathload (a) uses 100 packets in every burst except for the first one, (b) the packet size is fixed around 200 bytes, (c) the gap between probes is fixed for the every burst but varies across bursts, (d) the probe packet size and the probe gap control the instantaneous probe rate, (e) the number of bursts per second varies between 2 and 6, and (f) the number of bursts per second along with the probe size and number of probes per burst control the average bandwidth consumed.

probe rate (d) and the average consumed bandwidth (f).

After it receives every stream, pathload uses the relative difference in the arrival times of probe packets (i.e.,the one way delays or OWDs) to determine whether the instantaneous probing rate of the last stream induced queuing delays. Pathload tries to detect an increasing trend in the one way delays which it interprets as an indication of congestion.

Pathload analyzes each stream by collecting the K one way delays and by partitioning them into $G = \sqrt{K}$ groups ($G = 10$ in the default case). It then factors out outliers by taking the median OWD of each group, and plugs the values into two tests, that de-

termine (a) whether the stream experienced a *consistent* increasing OWD trend for the whole stream, and, (b) whether the stream experienced a *net* increasing OWD, by using only the values from the first and last OWD group. Pathload repeats these calculations for every stream in the fleet and then deems that a fleet has an overall increasing OWD trend if both (a) and (b) are above certain thresholds for the majority of the streams. If the OWD trend for a fleet is inconclusive, pathload keeps sending fleets until it can converge. Jain and Dovrolis describe in detail the complete pathload algorithm [6, 31] and Sommers discusses calibration issues along with the metrics used by pathload [32] (Section 2.2.1.3).

2.2.2 Yaz (available bandwidth)

Yaz [23] is an active measurement tool that estimates the available bandwidth of an end-to-end path. It is based on pathload and can be thought of as a *calibrated* version of that tool, in that it uses empirical measurements to determine the least number of probes needed in a probe stream so that the measurement error remains below a pre-configured threshold. Figure 2.3 shows the burst profile of a sample yaz session over a 10 mbps bottleneck link with 1 mbps CBR UDP cross traffic.

Yaz operates in a similar fashion to pathload. It iteratively sends streams of probes of equal size, with equal spacings between probes. Like pathload, the probe size and the interval between probes determines the probing rate. Unlike pathload, yaz determines during run time what is the best size for the probe stream by alternating calibration periods (the spikes in Figure 2.3) with measurement periods. In the example session shown, the

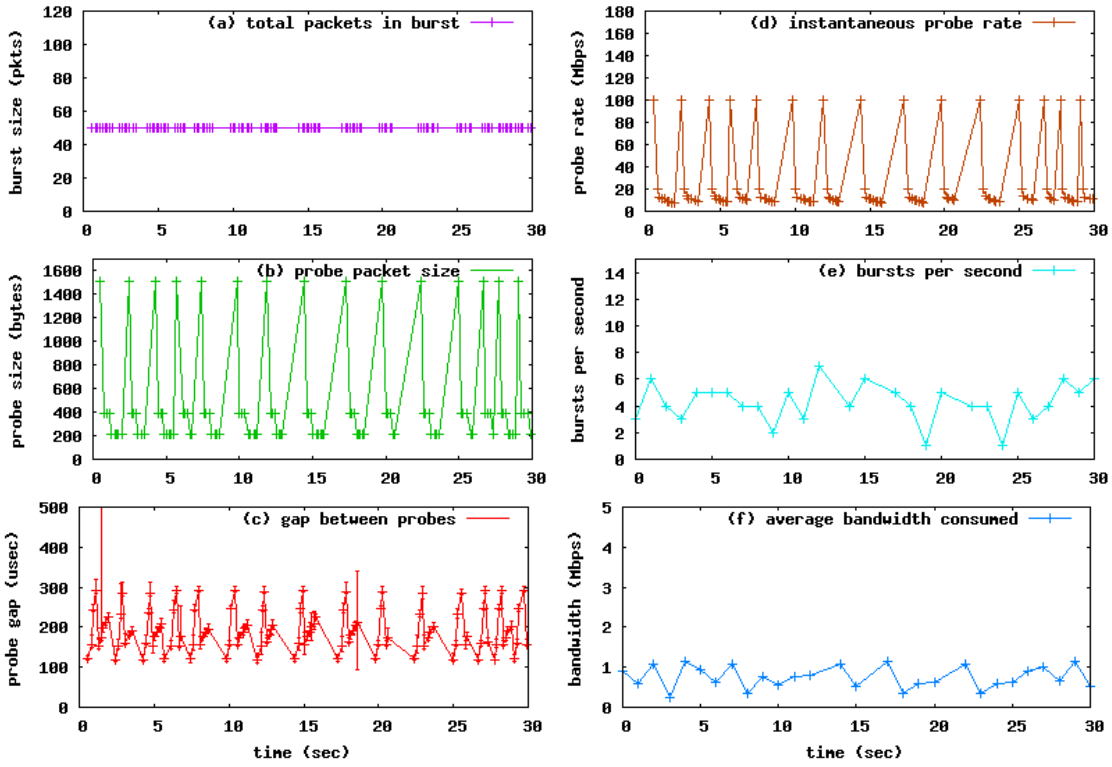


Figure 2.3: Burst Profile of Yaz session.

desired probe stream size is 50 probes. Yaz produces an estimate at the end of every measurement period, which results in more frequent measurements than pathload.

Yaz also differs from pathload in the estimation algorithm. While pathload only takes into account how the intervals between probes expand, yaz also considers how the intervals contract, and treats both events as indications of congestion. In addition yaz does not use the one way delays directly as pathload does, but uses averages to filter out error in individual measurements.

2.2.3 Pathchirp (available bandwidth)

Pathchirp [15] is an active measurement tool that measures end-to-end available bandwidth. Pathchirp, like pathload, relies on the principle of self-induced congestion to

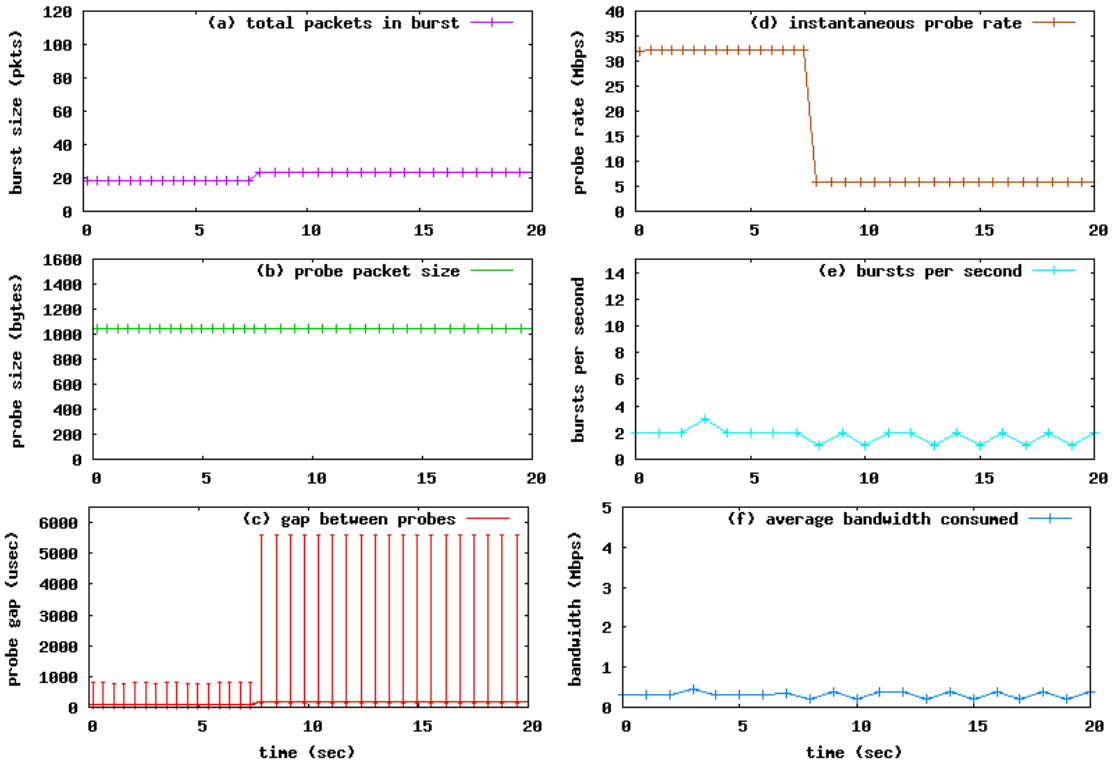


Figure 2.4: Burst Profile of Pathchirp. These plots have one point for every burst that pathchirp sends. Pathchirp produces estimates continuously. The plots show the first 20 seconds of a pathchirp sessions on a 10mbps bottleneck link with UDP cross traffic of 1Mbps We see that pathchirp (a) uses between 20-30 packets in every burst, (b) the packet size is fixed around 1000 bytes, (c) the gap between probes varies significantly within every burst; a few probes spaced up to 6 millisec but the average is around 200 microsec, (d) the instantaneous probe rate is of little importance here since every portion of each burst has different instantaneous rates due to the variable gaps, (e) the number of bursts per second is around 2, and (f) the number average bandwidth consumed is less than pathload (between 300kbps - 400kbps).

find the available bandwidth. Unlike pathload, pathchirp runs continuously and produces an estimate every second. Figure 2.4 shows the first 20 seconds of an example pathchirp session over a 10 mbps bottleneck link with 1 mbps CBR UDP cross traffic.

Pathchirp sends trains of probes from source to destination but, unlike pathload, pathchirp does not use the same spacing between probes of a particular stream. Instead it uses exponentially-decreasing spacing between probes of the same train. This has the advantage of embedding multiple probe rates within one probe stream. Pathchirp aims at being unobtrusive and more efficient than pathload since pathchirp does not have to send

a full packet train for every probing rate that it wishes to measure.

2.2.4 Pathrate (capacity)

Pathrate [7] is an end-to-end active measurement tool that estimates the path capacity, which is the minimum transmission rate along all links of the path. Pathrate uses the dispersion of long packet trains and packet pairs to arrive to its estimate after a variable amount of time.

An example pathrate session is shown in Figure 2.5 where the session takes a little less than 50 seconds to produce an estimate on a 10 mbps bottleneck link. Pathrate uses back-to-back probes with the maximum transmission unit (MTU) size. We can verify this in Figure 2.5 where plot (b) shows that the probe packet size is always around 1500 bytes and from plot (c) that shows that the gap between packets is a little over 100 microsec, which is approximately the transmission time of a 1500-byte packet on a 100 mbps link [1] (the bottleneck link is 10 mbps but the source node is on a LAN with a 100 mbps link).

Pathrate works as follows [2]. In the bootstrapping phase it sends multiple packet trains of back-to-back probes with increasing train length. The objective is to determine the maximum number of back-to-back probes that it can send per packet train without overloading the network buffers and causing loss. This step is clear in Figure 2.5(a) between 0 and 10 seconds where pathrate attempts to send trains with lengths up to 50

[1] The minimum transmission time for a packet is given by $gap * 10^{-6} = 8 * bytes/(linkrate * 10^{6})$, where gap is in microsec and linkrate is in mbps.

[2] The documentation that comes with the pathrate code version 2.4.1 has an excellent description of how the tool works.

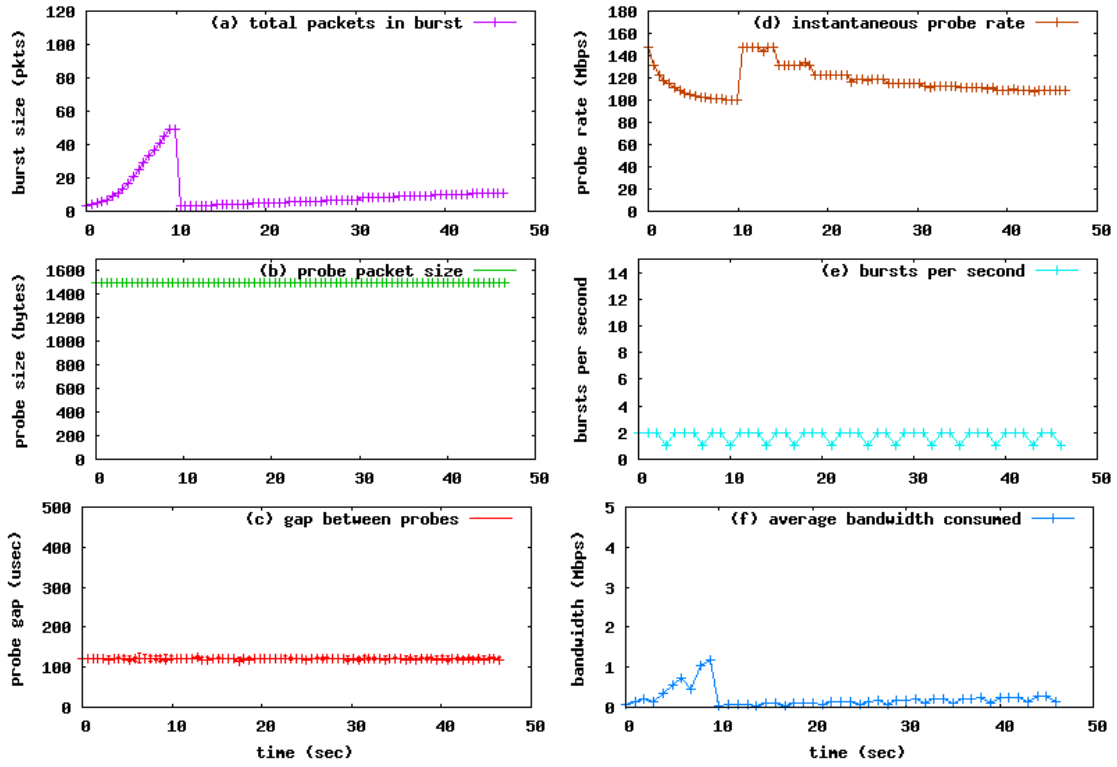


Figure 2.5: Burst Profile of Pathrate session.

probes.

Pathrate then enters a preliminary measurement phase that detects if the network path is shaped in any way. In this phase pathrate sends packet trains with lengths that increase slowly, which we can see in Figure 2.5(a) between time 10 sec and 48 sec. If the path is lightly loaded then pathrate is able to reach an estimate immediately and does not need to proceed to the main measurement phase. This is the case in our example where we use 1 mbps CBR UDP cross traffic on a 10 mbps bottleneck. If pathrate has not reached an estimate by this point, it enters a lengthy measurement phase where it first sends 1000 packet pairs of variable sizes, and then generates 500 packet trains.

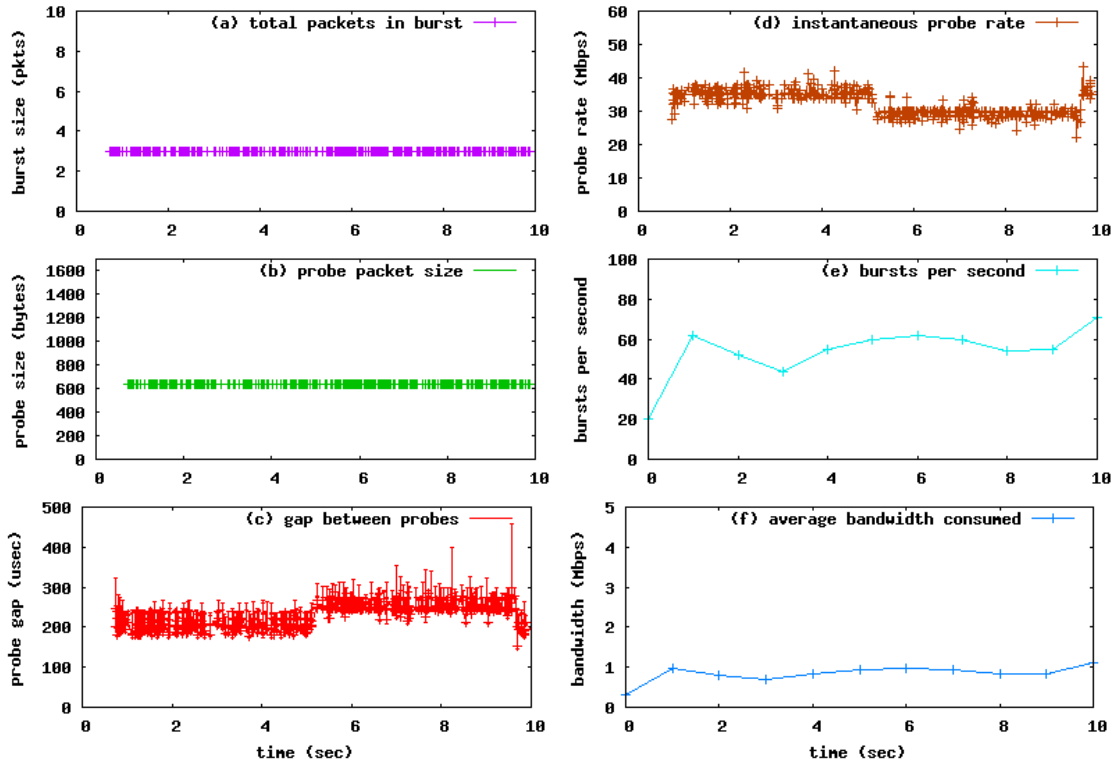


Figure 2.6: Burst Profile of Badabing session.

2.2.5 Badabing (loss)

Badabing [24] measures the loss rate of an end-to-end path. It sends a large number of short packet trains. The objective of the badabing packet trains is to capture loss due to congestion episodes. Badabing attempts to detect loss rates indirectly without incurring loss. For every packet train it receives, it checks the dispersion between probes and decides if the path is experiencing congestion while the train was en route. Then badabing uses this information (by analyzing consecutive intervals that experience congestion) to infer loss rate. Of course if a probe is lost, then badabing uses that information directly in its estimate.

The rationale behind using packet trains that contain a handful of probes each is that (a) single packet probes tend to miss congestion episodes, and (b) longer packet

trains distort the loss measurements because they tend to induce congestion themselves. Each packet train in badabing consists of three probes, each 600 bytes long. Badabing divides time in discrete intervals and in every interval it sends two packet trains back-to-back with probability 0.3. Figure 2.6 shows the first 10 seconds of a badabing operation. We verify that each packet train contains 3 packets (plot (a)), all the probes have the same size of 600 bytes (plot (b)). Plot (f) shows that badabing has probe overhead of 1 Mbps.

Chapter 3

The Measurement Manager

The Measurement Manager is an in-kernel service for sending measurement probes precisely with the least amount of overhead. The Measurement Manager Protocol (MGRP) schedules probes according to timing requirements provided by measurement tools while piggybacking application data inside the empty probe padding. MGRP can be viewed as a network protocol that fuses transport payload and probes at the sender and reconstitutes them at the receiver with the objective to minimize the bandwidth wasted on empty probe padding. We have implemented MGRP as a Layer 4 transport protocol in the Linux kernel and modified TCP in Linux to contribute packets to MGRP. This chapter presents the design and specification of MGRP, the core element of the Measurement Manager, while Chapter 4 presents the implementation details of MGRP.

In the Measurement Manager architecture, we have positioned MGRP in the network stack just before the IP layer and at the same layer as the transport protocols that contribute packets for piggybacking (Figure 3.1). MGRP works as follows. Rather than send probe packets directly (e.g., via UDP), an active measurement tool instead sends its probes through MGRP's Probe API by specifying an entire *train* of probes, which MGRP is responsible for scheduling. Most active measurement tools transmit packets with significant amounts of empty padding. MGRP treats such probes as *vessels* that potentially can contain useful payload. MGRP extracts this payload from data packets sharing the

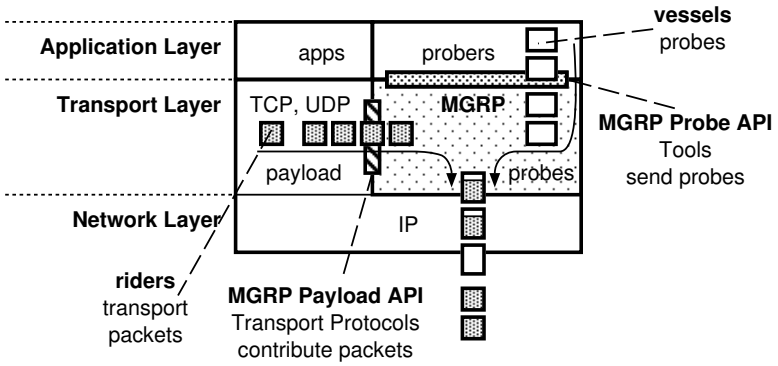


Figure 3.1: Position of MGRP in the Network Stack: MGRP is a transport protocol that is at Layer 4 of the network stack. It sits just above IP and is at the peer level of TCP and UDP. It exports, through the socket layer, a *Probe API* so that probers from userspace can schedule probes. MGRP also defines an in-kernel *Payload API*, so that transport protocols, like TCP, can contribute payload to be used for piggybacking.

same destination as the probes, which MGRP treats as *riders*. The piggybacked payload is extracted from the padding and reconstituted at the destination. Most of MGRP's complexity lies into matching riders with vessels.

To summarize, MGRP serves two roles: (a) MGRP sends probes on behalf of active measurement tools in the form of *probe transactions* through the Probe API, and (b) MGRP attempts to piggyback payload from transport packets on these probes by filling their empty padding.

This chapter begins by discussing the Probe API used by senders to specify probe transactions, which differs from the traditional approach of manually constructing and sending individual UDP packets. We next show the Probe API for probes at the receiver. We continue by showing how MGRP schedules probe transactions so that application data, when available, gets piggybacked on probes. We examine the effects of piggybacking on application and probe traffic, and discuss how information about MGRP piggybacking can help tools adjust their algorithms. We conclude by considering some variations to piggybacking policies that we have explored but did not use.

3.1 Probe API

MGRP sends probes on behalf of tools that wish to inject probes into the network. We saw examples of such applications in Chapter 2 where we surveyed a number of active measurement tools, which are applications that exclusively probe the network. But a client of MGRP (which we call a *prober*) can be any application that sends probes that contain empty padding.

In current practice, probers manually construct and schedule their probe packets, and send them via UDP. In contrast, an MGRP-enabled prober will specify packets to send using the MGRP Probe API, which provides means to define *probe transactions.* The kernel, upon receiving a transaction specification, will construct the completed probe packets and schedule their transmission over MGRP. Fortunately, it turns out that to convert a tool from using MGRP to using UDP is not very difficult, as we will soon show.

3.1.1 Probe Transactions

Unlike UDP, MGRP does not treat each probe independently. Instead, each probe is considered to be part of a *probe transaction*. This approach fits naturally with the way that probers send out probes (i.e.,in groups) as we demonstrated in Chapter 2. Each probe transaction contains the probe buffers to send, information about the empty padding in each probe and the transmission intervals between probes. Probe transactions are atomic and serialized in respect to each other; probes from different probe transactions never overlap. However packets other than probes can be interspersed between probes of a single probe transaction.

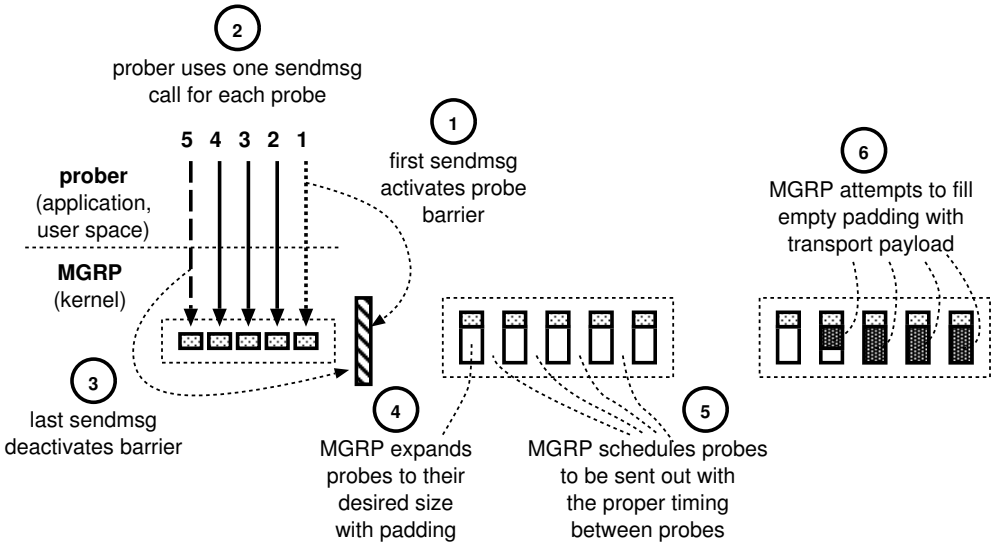


Figure 3.2: The life of a probe transaction in MGRP. In this example a prober is sending 5 probes that contain mostly padding and that are spaced equally apart. In the first stage (steps 1,2,3) the prober sends the probe transaction specification to MGRP with repeated `sendmsg` calls. In the second stage (steps 4,5) MGRP creates the desired probe sequence, and in the last stage (step 6) MGRP attempts to piggyback any transport payload and sends out the probes.

MGRP looks to probers like a transport protocol such as UDP. In fact the MGRP socket API is backwards compatible with UDP[1] and can be used as a drop-in replacement of UDP (i.e., the prober can substitute an MGRP socket for a UDP socket without changing any of the other socket calls). However, in UDP-compatibility mode the prober cannot use any of the advanced features of MGRP. To access these features a prober needs to always use `sendmsg` to send each probe (instead of `sendto` or `send`) since it needs to specify a probe transaction by passing MGRP-specific "ancillary data."

A probe transaction is constructed at the sender by calling `sendmsg` for each probe in the transaction. An example of a probe transaction is shown in Figure 3.2 and the corresponding pseudo code in Figure 3.3. The first `sendmsg` call schedules the first probe and

[1] Stevens provides a complete reference to the Unix Socket API in Volume 1 of Unix Network Programming [33]. Of particular relevance to our discussion is section 13.5 about `recvmsg` and `sendmsg` functions.

```
/* 5 packets of size 1000bytes each with 1ms gap */
char buf[1000];
int probe_size     = 1000;
int probe_data     = 20;        /* probe header */
int probe_gap_usec = 1000
int probe_pad      = probe_size - probe_data;

/* pass information using ancillary data */
struct msghdr msg = {...};
struct mgrp_probe *probe = (pointer to msg_control buffer);

int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_MGRP);

/* pass first probe to MGRP: activate barrier */
probe->barrier = 1;
probe->pad = probe_pad;
probe->gap = 0;
sendmsg(sock, &msg, 0);

probe->gap = probe_gap_usec;

/* pass probes 2..4 to MGRP */
for (i = 2; i <= 4) {
    sendmsg(sock, &msg, 0)
    /* no delay between calls */
}

/* pass last probe to MGRP: deactivate the barrier */
probe->barrier = 0;
sendmsg(sock, &msg, 0);
/* MGRP sends the packet train before it returns */
```

Figure 3.3: Pseudo code of a probe transaction in MGRP.

activates a *virtual barrier* inside MGRP that delimits the beginning of a probe transaction. As long as the virtual barrier is active, MGRP buffers all subsequent calls to `sendmsg` for that specific socket. The last call deactivates the virtual barrier and instructs MGRP to send all the probes. Before returning from the last `sendmsg` call, MGRP pads each probe to the desired length and sends all the probes in the transaction with the desired timing. In section 3.2 we will discuss how MGRP may use transport data in place of empty padding.

MGRP learns about the details of each probe transaction through ancillary data. Each call to `sendmsg(fd, msg)` corresponds to one probe packet and the function argu-

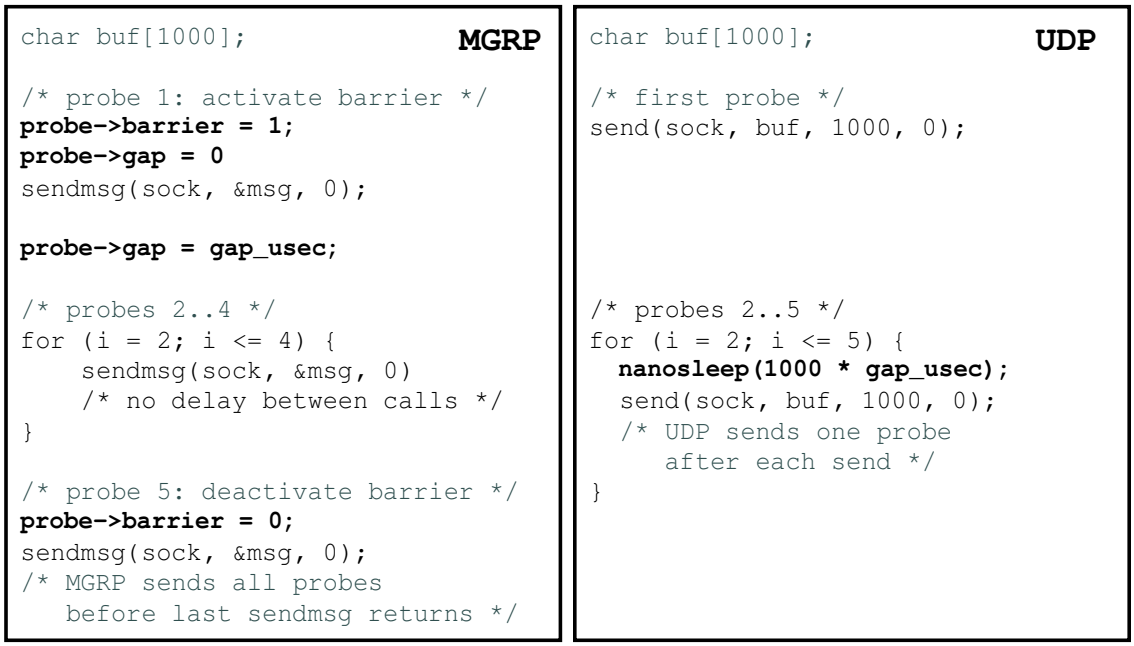


Figure 3.4: Key differences between MGRP and UDP socket APIs. In MGRP we need to send a specification of a probe transaction before the probes are sent out. All the probes are sent out during the last `sendmsg` call.

ment `msg`[2] contains a byte buffer and MGRP-specific ancillary data to indicate: (a) how much padding to include in the packet (in addition to the byte buffer), (b) the desired interval between the transmission of the previous probe, and the current probe, (c) whether the padding can be used for piggybacking data packets, and (d) whether the virtual barrier is active.

There are key differences between the MGRP and UDP socket APIs. as shown in Figure 3.4. When a prober sends UDP probes using `sendmsg` the probes are sent immediately after each call. So if a prober wishes to generate gaps between packets it just introduces delay between the `sendmsg` calls. Instead, in the MGRP case, a prober uses the `sendmsg` call to *schedule* a probe as part of a probe transaction The probes are buffered inside MGRP and are sent out only after the `sendmsg` call for the last probe. It is MGRP and

[2] The `msg` argument in `sendmsg(fd, msg)` is a `struct msghdr` from the Unix Socket API [33].

not the prober that generates the spacing between packets based on the MGRP-specific ancillary data. If the prober introduces a delay between MGRP `sendmsg` calls, it only delays the transmission of the first probe and does not affect the delays between probes. Another difference from the UDP case is that the probers pass to `sendmsg` only the byte portion of the probe that is not padding. MGRP pads the probe to the desired size using information in the ancillary data. This is key to the ability of MGRP to piggyback data on probes as we discuss in section 3.2

3.1.2 Receivers

At the destination, the tool's receiving component opens an MGRP socket and calls `recvmsg()` to retrieve each of the received probes. Each received probe consists of the header as provided at the sender, along with any extra padding that was specified. MGRP uses ancillary data to return the sender and receiver timestamps of the probe, which are indispensable to most active measurement tools.

The first timestamp is the system timestamp taken at the receiver, which is the same timestamp that can be retrieved using the `ioctl` option `SO_TIMESTAMP` in normal UDP sockets. This timestamp is typically taken by the kernel in the device driver code, just after the packet has been pulled from the device (the second layer from the bottom in Figure 4.1). The second timestamp is taken at the MGRP sender just before it sends the packet to the IP layer. The timestamp is stored in the MGRP header and extracted at the receiver (Section 4.2). Figure 3.5 shows pseudo code for receiving probes in MGRP and shows how the MGRP timestamps can be extracted from the `recvmsg` call.

```
struct mgrp_timestamps {
    struct timespec snd;
    struct timespec rcv;
};

struct probe_info {
    int size;
    char buffer[2000];
    struct mgrp_timestamps timestamps;
};

struct probe_info probe[5];

/* receive probes 1..5 from MGRP */
for (i = 0; i < 5) {
    msg.msg_iov.iov_base = probe[i].buffer;
    msg.msg_iov.iov_len = 2000;
    msg.control = &probe[i].timestamps;
    probe[i].size = recvmsg(sock, &msg, 0)
}
```

Figure 3.5: Pseudo code for receiving probes in MGRP

3.2 MGRP Piggybacking

After having discussed the first role of MGRP in sending probes through probe transactions (section 3.1), we discuss in this section its second role, where MGRP attempts to reuse the wasted padding inside the probes to piggyback transport payload. The Probe API that we outlined in section 3.1 provides MGRP with information about the padding inside every probe. This information gives the opportunity to MGRP to fill the padding with something more useful than placeholder bytes. We now present MGRP's basic piggybacking approach using an example.

3.2.1 Piggybacking by Example

Figure 3.6 illustrates how MGRP implements a probe transaction using piggybacking. At step (1), a measurement tool submits a probe transaction to specify a mea-

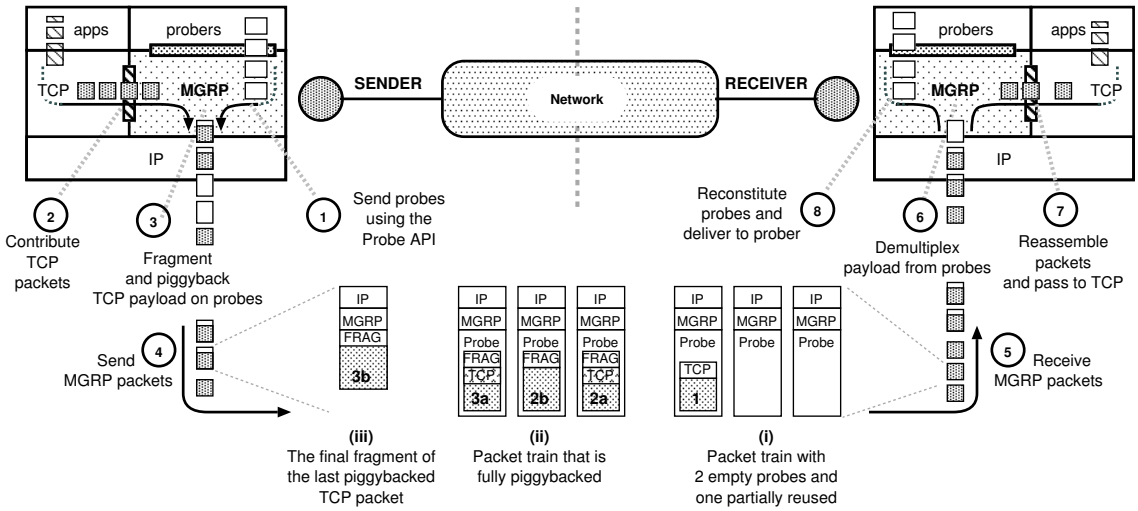


Figure 3.6: A step-by-step example to demonstrate MGRP operation. Measurement tools generate probes, which MGRP combines with TCP (or UDP) packets sent by applications. Depending on the timing, MGRP packets can contain (i) probes that are partially reused by full payload packets, (ii) probes that are fully reused by payload fragments, (iii) final payload fragments when MGRP has no more probes to piggyback on. The combined packets are reconstituted at the receiver.

surement. At the same time, applications may send data to the same destination as the probes (2). For example, in our video streaming scenario from the Introduction, an application may be streaming media data to some destination D when it invokes an active measurement tool (such as our MGRP-adapted pathload) that submits a transaction to probe the available bandwidth on the path to D.

Given a probe transaction, MGRP attempts to fill any empty padding in each probe with *riders*—the application packets bound for the same destination (3). Eventually it sends along the probes (4). In our implementation, we modified the TCP stack to pass its outbound traffic to MGRP for potential piggybacking; modifying UDP should be straightforward but is not yet implemented. MGRP will briefly buffer application frames (something that applications should presume TCP or UDP might do in any case) to increase the chances for successful piggybacking. If no riders are available, the probe is simply sent with empty padding (as in the first two probes of (i)). Conversely, if a potential rider

has been in the payload buffer long enough without being piggybacked, MGRP sends it normally.

Given a probe packet and a rider to piggyback, MGRP tries to fit the rider inside the probe's padding as follows. If the rider is smaller than the padding, piggybacking is straightforward (probe labeled 1. in (i)). More interestingly, if the rider is larger than the padding, MGRP fragments the rider into two chunks, piggybacking a padding-sized chunk in the probe and re-queuing the remainder for further piggybacking. We use a simple, custom fragmentation/reassembly protocol to manage fragmented chunks (section 4.6). In the figure we can see that probe packets in (ii) carry a single, fragmented TCP packet, whose chunks are labeled 2a and 2b. If MGRP cannot piggyback an entire rider before the queuing timeout, MGRP simply sends the chunk in its own MGRP packet, as shown in (iii) in the figure. Given enough space, a probe's padding could be filled with chunks from multiple riders, but for simplicity MGRP limits itself to a single rider.

Once a probe is ready, MGRP stores the current time in the MGRP packet header and hands it to the lower layer for transmission. MGRP then waits the specified gap time before constructing and sending the next probe (note that other application frames may be sent during these gaps).

When probe packets are received at the destination (5), MGRP extracts any piggybacked riders (6), reads the system packet timestamp, and queues the probe packet for delivery (8). When the receiver retrieves the probe packet, it may also retrieve the sender's and receiver's timestamps as ancillary data. If any rider piggybacked in the probe is a complete TCP or UDP frame, MGRP simply delivers it to the appropriate receive buffer.

If the rider is a chunk (fragment), MGRP stores it in a reassembly queue until all chunks are received, at which point the original data packet is reconstituted and delivered the appropriate receive buffer (7).

Piggybacking buffered transport packets on top of probe packets changes the way these packets interact with the network. We now consider the effects of piggybacking, good and bad, on application data and measurement tools, and discusses in particular how tool algorithms may need to be adjusted to account for piggybacking.

3.2.2 Piggybacking Effects on Application Data

Piggybacking data packets within probes can reduce the number of packets and bytes sent across the network, compared to sending probes and application data separately. In the best case, this savings is enough to eliminate induced congestion along the path, thus avoiding disruptive packet drops of source traffic, cross traffic, and measurement probes. Our experimental results presented in Chapter 5 show that this reduction in loss rates often leads to better application throughput, is more fair to cross traffic, and reduces the time required to complete measurements.

However, while piggybacking can reduce the total number of lost packets, it can increase the chances that a lost packet contains application data, and it can increase the negative consequences of such as loss.

To see why, observe that piggybacked application data is sent at the rate of the probes on which it piggybacks. Since measurement tools often send bursts of high instantaneous-bandwidth probes, these bursts are likely to induce loss. If the probe burst

rate is higher than the normal application data rate, then any lost packet is more likely to contain application data. For example, suppose a probe transaction sends 10 probes over 1 ms, while the application sends at roughly 2 padding-sized packets per ms. With a 5-ms buffering timeout, we will buffer 10 application packets and can completely fill the probes' payloads. If we did not use MGRP at all, we would send 12 total packets—two application packets, and ten probes. While the chances of *some* lost packet may be greater in the second case, the chances of a lost *application* packet may actually be greater in the first case. If the greater piggybacking is not sufficient to prevent loss events altogether, a loss is thus more likely to harm the application.

When a data packet loss occurs, MGRP can magnify the negative consequences. In particular, buffering increases application packets' round trip time (RTT). While this additional latency is not necessarily a problem in and of itself, an increased RTT delays TCP's (or applications') response to packet losses, making it slower to increase the congestion window. So while longer buffering timeouts can increase piggybacking and thus reduce the total number of packet drops, if an application's TCP packet *is* dropped then the increased RTT will slow the return to an application's preferred bandwidth.

We address both problems by choosing a relatively small buffering delay, to balance the positive effects of more piggybacking with the negative effects of increased application packet loss. We have found that 5 or 10 ms is sufficient to buffer substantial amounts of data without harming TCP's responsiveness when dealing with typical wide-area RTTs of 40 ms or more [34].

In addition, we have found that carefully choosing *which* probes on which to piggyback can positively impact performance. For example, pathload begins a measurement

round by sending a burst of 100 packets back-to-back. If there is any congestion, the packets toward the end of this burst (and any packets that might follow it) are likely to be dropped. If we selectively piggyback on only the first n packets of the burst, then if later probes in the burst are dropped, no data packets will be dropped with them. We could use external information to determine a suitable n automatically; e.g., if the TCP congestion window for a piggybacked data stream is small, that suggests the path is highly contended, and the likelihood of loss events is higher (we have not implemented this).

3.2.3 Piggybacking Effects on Measurement Tools

As mentioned above, by reducing the contention for a shared path, MGRP piggybacking may allow a measurement tool to converge more quickly. On the other hand, to provide accurate results, tools may need to account for locally-originating traffic that has been piggybacked.

Without MGRP, local traffic can affect a measurement tool's results by being interspersed with the tool's probes, thereby affecting probe latency, queuing behavior (whether it is dropped or not), etc. To probe tools that include a gap between packets, interspersed local traffic is equivalent to cross traffic. With MGRP, data traffic between the same hosts has much less influence: MGRP may piggyback data packets on the probes themselves, and also may delay local packets until after a probe transaction has completed, due to buffering.

While in a sense this gives the tool a clearer picture of the traffic that is competing with the local application, MGRP changes what a probe train measures. Instead of

Probe Statistic	Description
barrier_engaged_usec	How long the barrier was engaged. This represents the time it took for the application to send the probe transaction to MGRP
mgrp_transmission_usec	How long it took MGRP to send the whole probe transaction with the proper gaps.
pbk_pkts	The number of probes that were piggybacked.
pbk_bytes	The total number of bytes that were piggybacked.
pbk_compete_pkts	Of the probes that were piggybacked, the number of probes that carry payload that would have competed with the probes in the absence of piggybacking.
pbk_compete_bytes	The total number of bytes for pbk_compete_pkts.

Table 3.1: Details that MGRP returns about each probe transaction it sends.

measuring both local and cross traffic, with MGRP the spacings of a probe train take a snapshot of the network *as if the application were sending less traffic* (reduced by the traffic that was piggybacked). Since probe trains have relatively short durations and do not run continuously, without correction they may produce inaccurate estimates.

For that reason MGRP can provide details about what was piggybacked when a tool sends a probe train. In particular, a probe sender can query MGRP from userspace using the `MGRP_IOC_PROBE_RESULTS ioctl` to acquire the information shown in Table 3.1, which among other things indicates how much piggybacking took place.

3.2.3.1 Pathload

This section describes how we modified pathload to adjust its estimates to account for piggybacking when running on top of MGRP.

In pathload the estimation algorithm runs on the receiver. Since the piggybacking

information is available only at the sender, the process of adjusting the pathload estimates entails 3 steps: (1) the pathload sender retrieves from MGRP the piggybacking details after each stream is sent, (2) the pathload sender sends this information to the pathload receiver, and (3) the pathload receiver uses the piggybacking information to adjust its estimate.

The modifications to the original pathload are minimal. Retrieving the piggybacking information is accomplished at the sender through a simple `ioctl` after each stream of 100 probes is sent. Pathload needs two pieces of information from MGRP: (1) the number of piggybacked bytes that would have competed with the probes, and (2) the time window during which the competing traffic would have been sent (`pbk_compete_bytes` and `mgrp_transmission_usec` from Table 3.1). With these two numbers we can compute the rate of the piggybacked transport payload that would have otherwise competed (and been taken into account) by the probe stream.

This information is retrieved at the sender but is needed by the receiver. Since there exists already a control channel between sender and receiver, and the sender sends an acknowledgment after each stream, it is straightforward to simply send the piggybacking information to the receiver. Once pathload has sent enough streams, based on its algorithm, it computes an estimate based on the rate of the last stream. We modified pathload to subtract from the original estimate the rate of the competing traffic that was piggybacked, where this rate is calculated in one of two ways: (1) from the most recent stream only, or (2) from the average of all streams of the current estimate's fleet. We believe the former makes the most sense, since pathload's estimate is due only to the last stream. Examples of 3 pathload sessions and the way that the original estimates have been ad-

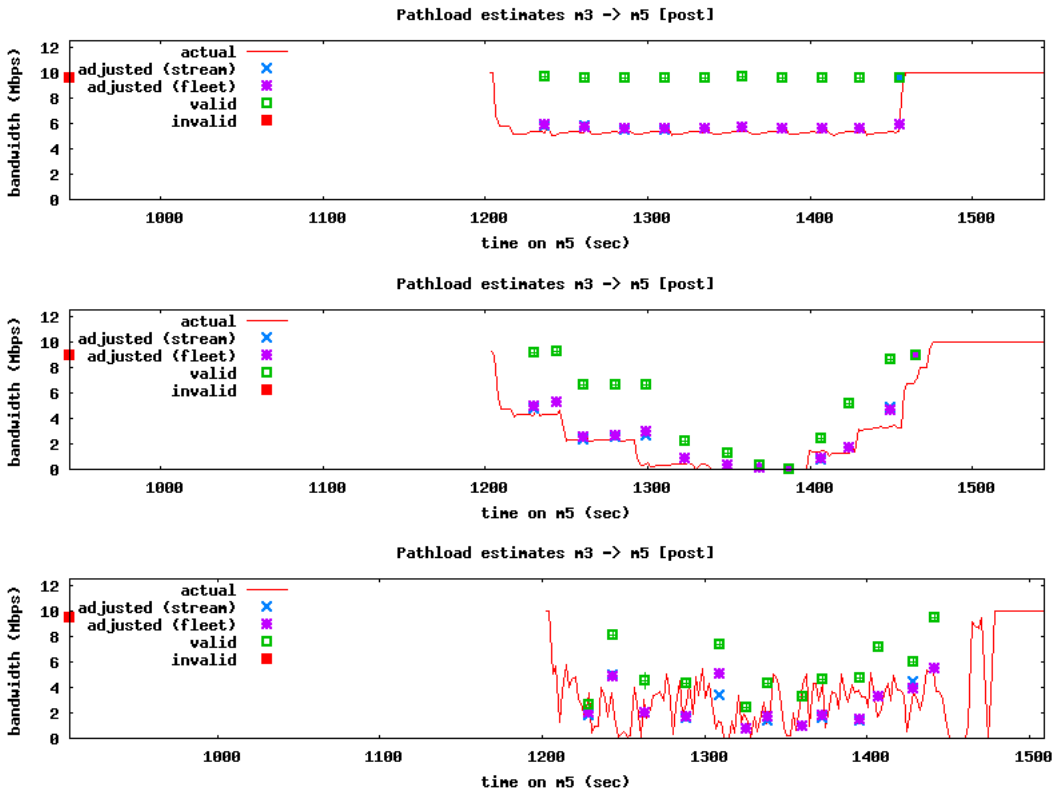


Figure 3.7: Examples of adjusting pathload estimates for piggybacking

justed according to these two methods appear in Figure 3.7. We present more detailed measurements of pathload accuracy in Chapter 5.

3.3 Piggybacking Policy Variations

The piggybacking approach described in Section 3.2 makes a couple of key design decisions. First, it is rider-agnostic; while we currently only piggyback TCP packets, MGRP in no way takes advantage of this fact when, e.g., fragmenting and reassembling large packets. Second, the design aims to reduce overall bandwidth consumed by piggybacking as much as it can. As alternatives to these two design points, we consider some other possible policies here, and discuss our experience with them.

3.3.1 Standalone Packets

Rather than fragmenting packets with a rider-agnostic protocol, we could instead take advantage of TCP's stream semantics to chop up a TCP packet directly into smaller chunks, each with a new TCP header. This makes each chunk a valid standalone "mini-packet," which can be included in the MGRP header with its protocol number. At the receiver, the mini-packet is extracted from the MGRP packet and delivered directly to the transport stack since it has a valid header.

Compared to the generic fragmentation approach, the mini-packet design has the benefit of being much simpler, and more backward-compatible with existing tools like `tcpdump` and `tcptrace`, which analyze packets. We implemented this approach in an earlier prototype of MGRP and found that it has two serious drawbacks. First, it multiplies the total number of acks fairly drastically for small probe packets but large TCP segments, which increases overhead on both the end hosts and the network. Second, it results in duplicate acks when a mini-packet that was toward the middle or end of a larger segment is lost. These duplicate acks may cause TCP to reduce its congestion window, penalizing the source traffic's throughput.

3.3.2 Duplicate Piggybacked Packets

As an alternative to piggybacking application packets, we might consider piggybacking them *and* sending the application packets as usual. Doing this has the downside that it will not save any bandwidth. On the up-side, it can alleviate the "shared fate" problem discussed above, giving each data packet two opportunities to reach the destina-

tion. We could go one step further and imagine duplicating chunks within a probe stream, similar to Skype's behavior [35], since any unused padding is wasted network bandwidth; after a certain threshold we could simply repeat the chunks within the probe stream and de-duplicate them at the MGRP receiver.

We have implemented this idea in MGRP but unfortunately have found in our experiments that the disadvantage of not reducing overall bandwidth is not ameliorated by the benefit of duplicating source packets. It is possible that further efforts and experimentation could yield a fruitful policy.

3.3.3 Partial Piggybacking

As discussed in Section 3.2.2, another way to address the "shared fate" problem is to piggyback selectively. We have implemented two policies for doing this: allowing measurement tools to selectively turn off piggybacking, and allowing them to specify a piggybacking ratio. We discuss these in greater detail in Section 4.7.

Chapter 4

The MGRP Implementation

In this chapter we describe the implementation of the MGRP protocol in Linux. We will describe in detail all the mechanisms that MGRP uses to piggybacking transport payload on probes.

4.1 The MGRP Linux Kernel Module

We have implemented MGRP as a loadable kernel module in the Linux 2.6.25 kernel building on kernel mechanisms introduced by DCCP [36] for registering protocols with IP and exporting a socket interface [37, 38]. Once the kernel module is loaded, the behavior of the Linux kernel changes in the following ways:

1. MGRP registers itself as a new Layer 4 protocol such that all incoming IP packets with the MGRP protocol field are passed to the module for processing. For experimental purposes [39, 40] we picked 254 as the MGRP protocol number. Figure 4.1 shows the position of MGRP in the Linux network stack.

2. Transport protocols can use MGRP hooks to contribute their outgoing packets to MGRP before they go over IP.

3. Applications can open an MGRP socket, a new type of datagram socket, that they can use to send probes over MGRP. The Probe API is implemented through this

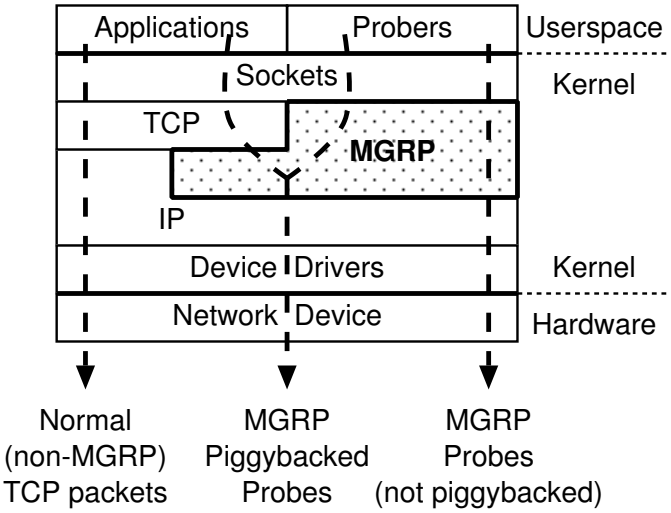


Figure 4.1: MGRP is positioned at the Layer 4 of the Linux network stack. Probers can access it through the socket layer to send probes, and TCP use MGRP hooks inside the kernel to contribute packets for piggybacking. MGRP uses its own IP protocol field, so that all incoming MGRP packets can be handed directly to MGRP from the IP layer.

socket interface.

4.2 MGRP Header

MGRP is a transport protocol that carries probes. These probes have padding that may be filled with payload from other transport protocols. So the MGRP header must contain enough information so that MGRP receiver can: (a) reconstruct the probe, restore its padding and acquire the sender's timestamp, and (b) extract the transport payload from the probe padding and reconstitute the original transport packet. The MGRP header fulfills both of those purposes with a fixed 16-byte header that is shown in Figure 4.2. The header contains:

1. **One 32-bit sequence number.** This is a number that, combined with the time stamp, uniquely identifies an MGRP packet. It is used to reassemble the fragments of transport payload embedded inside the probe padding into the original packets

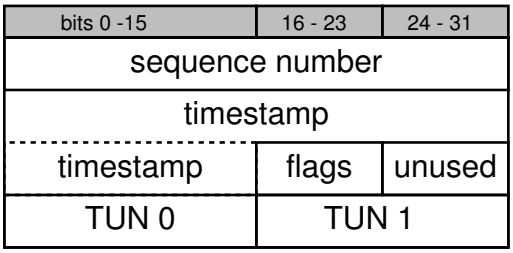


Figure 4.2: The MGRP Header contains a 32-bit sequence number, a 48-bit timestamp and two Transport Unit (TUN) sub-headers that are used to extract the different parts of the MGRP packet (probe and transport payload).

(details in section 4.6).

2. **One 48-bit time stamp.** This is the send timestamp of the MGRP packet. It has nanosecond resolution and rolls over about every 78 hours. With this time stamp probers can calculate one-way delays, which as we saw in Chapter 2 play an important role in the estimation process of active measurement tools. This timestamp is taken at the sender using the high resolution clock of the Linux kernel just before the MGRP packet is pushed into the IP layer (Figure 4.1).

3. **One 8-bit field for flags.** The first two bits are used to store the number of Transport Units (TUNs) or Transport Chunks that the MGRP packet is carrying. A TUN can represent a probe header or piggybacked transport payload. Each MGRP packet can carry at most two TUNs: one probe and one transport payload. We provide details about about how TUNs are decoded in the next header item (TUN sub-header). The third bit is the `HAS_UNUSED_PADDING` flag. This flag indicates that the MGRP packet contains at least one pad byte and is used to reconstruct the probe.

4. **Two 16-bit TUN sub-headers** Each TUN sub-header describes a portion or a *Transport Unit (TUN)* inside the MGRP packet. As we see from Figure 4.3, the

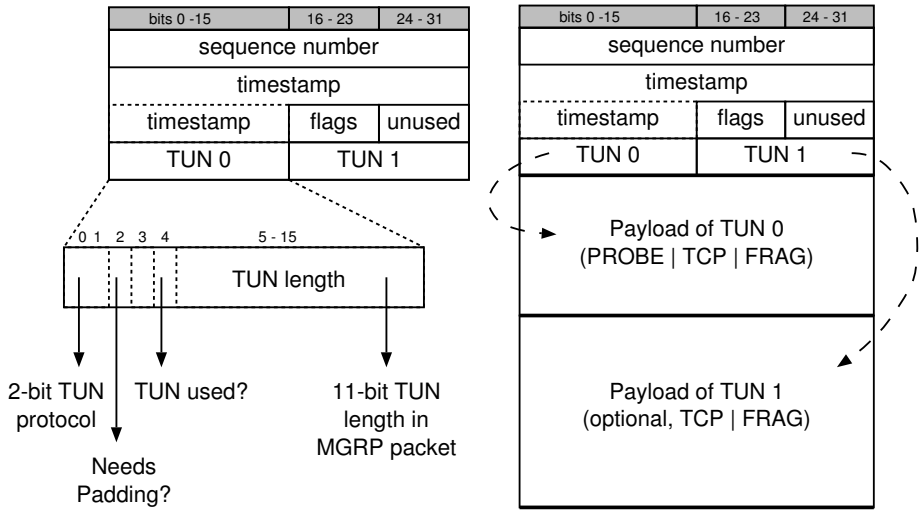


Figure 4.3: How MGRP extracts Transport Units

TUN sub-header has a 2-bit TUN protocol field, some flags and a length field. The protocol field tells MGRP where to deliver the TUN once it extracts it from the MGRP packet. Supported protocols are (a) TCP, (b) PROBE and (c) FRAG. The last protocol is a pseudo protocol that we use during fragmentation (section 4.6). The TUN sub-header does not contain any offsets to the data inside the packet because to derive them is straightforward. The first TUN (TUN-0) starts after the MGRP header, and the second TUN (TUN-1), if it exists, starts after TUN-0. Figure 4.3 shows how MGRP extracts Transport Units from MGRP packets using the TUN sub-header.

4.3 Probe Generation

The process of generating the probes of a probe transaction is straightforward. As we described in section 3.1 it is critical for the probes in a probe transaction to be transmitted at precise intervals. MGRP generates the gaps between probes at sub-microsecond

precision by using a a combination of high resolution timers (hrtimers) and busy waiting [1]. Using hrtimers is desirable, because they are not CPU-load intensive, but by themselves they are not sufficient for our precision requirements, because they can overshoot their expiration time. So we use a combination of methods: hrtimers for the initial portion of the gap up until a threshold, and after that we do busy waiting.

We have instrumented MGRP with the ability to monitor the accuracy of probe generation [2]. In Table 4.1 we show an example of our gap generation accuracy. The table shows how accurate MGRP is in generating 100 microsec intervals between the probes of a 20-probe transaction. Each table entry represents one interval between probes. Column 1 show the probe number, column 2 shows the number of nanoseconds that we went over the desired gap, column 3 shows the number of nanoseconds we went under the desired gap and column 4 shows the actual gap that we generated in nanoseconds. This example demonstrates that for even such small probe gaps the overrun for each probe is in the order of 200-300 nanosec for a gap of 100 microsec. That is an 0.3% error.

Our hybrid scheme for generating gaps needs two variables to be calibrated that we calibrate during MGRP initialization: (a) *The busy wait threshold.* This is the final portion of the gap that has to be generated using busy waiting. (b) *The busy wait adjustment ratio.* After we return from the hrtimer, we need to busy wait for the remaining part of the gap. This variable reduces the actual number of remaining nanoseconds (a value of 75 reduces it by 1/4). This is needed because of the overhead associated with retrieving wall clock time and executing ndelay().

[1] For busy waiting we use ndelay().

[2] To turn on gap monitoring in MGRP: `echo 1 > /sys/module/mgrp/parameters/probe_show`.

Probe Num	Overrun (ns)	Underrun (ns)	Actual Gap (ns)
0	**0**	0	0
1	**311**	0	100311
2	**148**	0	100148
3	**218**	0	100218
4	**225**	0	100225
5	**251**	0	100251
6	**253**	0	100253
7	**273**	0	100273
8	**297**	0	100297
9	**241**	0	100241
10	**240**	0	100240
11	**228**	0	100228
12	**246**	0	100246
13	**273**	0	100273
14	**246**	0	100246
15	**250**	0	100250
16	**261**	0	100261
17	**225**	0	100225
18	**344**	0	100344
19	**283**	0	100283

Table 4.1: Error in generating gaps between probes in MGRP for 20 probes with a desired gap of 100 microsec. MGRP goes over the desired gap (second column) for about 200-300 nanosec, which represents an error of 0.3%.

4.4 The Payload Buffer

Whenever MGRP receives packets from TCP, it buffers them in in its payload buffer so that it can perform piggybacking. Buffering is necessary because tools typically send probes in short intense bursts while applications, through their transport protocols, space out their packets to avoid congestion and loss. Without a payload buffer it would be practically impossible to match probes with payload. The payload buffer is only needed on the sender side.

TCP packets are buffered in the payload buffer so that they can piggyback on probes. But if no probes are available these packets need to be transmitted in a timely

manner.

MGRP guarantees that every payload packet it receives from TCP will be either piggybacked or transmitted Tms after it was received, where T is the payload buffer delay. To do this, MGRP stores each TCP packet in a packet queue and starts a virtual timer with an expiration time Tms in the future. If the timer expires and the packet has not been piggybacked, then MGRP sends it out immediately. For efficiency reasons, the MGRP implementation does not start one timer per packet. Since all timers expire in the same order that they are started, it is sufficient to keep a queue of *transmission time requests* and always have one timer active for the request at the head of the request queue.

This works very well due to the high resolution timers (hrtimers) [41] that were recently introduced in the Linux kernel (starting with version 2.6.16). Empirically, the resolution of the Linux hrtimers is on the order of $100ns$ which is at least an order of magnitude smaller than the typical intervals between consecutive packets sent out by TCP. That means that the MGRP payload buffer preserves the intervals between transmitted TCP packets and only adds a constant time offset.

We initially used low-resolution timers to implement the MGRP payload buffer. This resulted in poor performance for small MGRP delays because the resolution of the timers (10-20 ms) was too close to the desired delay. What we observed was that the fine-grained intervals between TCP packets were lost and TCP packets were clustered together more than the TCP congestion algorithm permitted, which led to increased losses.

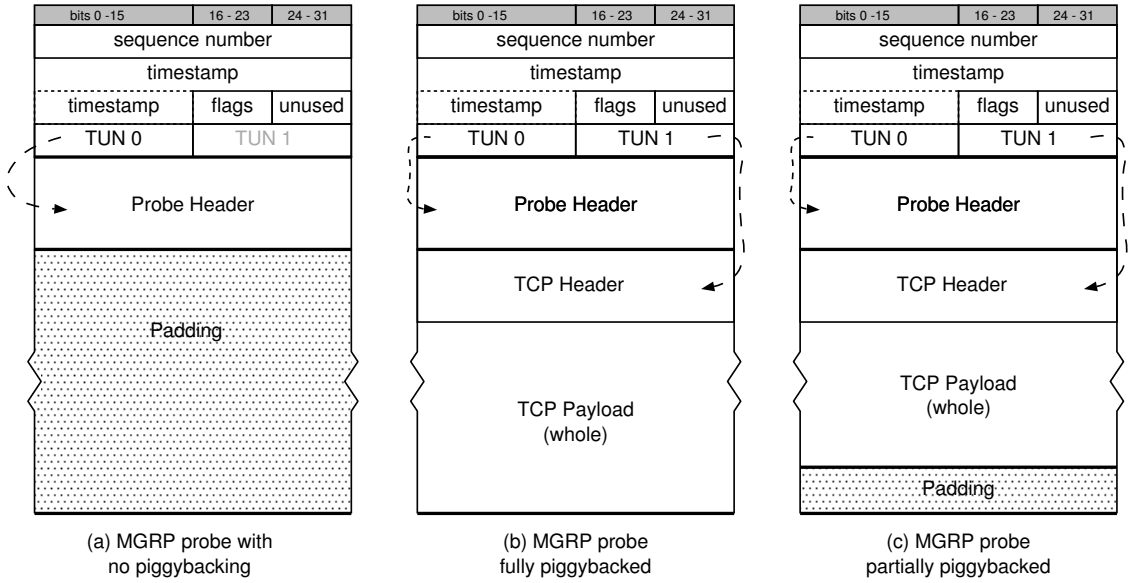


Figure 4.4: Examples of Piggybacked Packets

4.5 Piggybacking

Piggybacking in MGRP happens on a probe by probe basis. Whenever MGRP is ready to send a probe, it first looks into the head of its *payload buffer* (described in Section 4.4) for a candidate packet for piggybacking and then it expands the probe to the desired packet size using padding. The whole point of MGRP is to avoid the need for empty padding and instead extend the probe by using application payload. There are many cases depending on the state of the payload buffer:

1. The buffer is empty: MGRP adds the desired padding to the probe and sends out the packet with one MGRP chunk with proto=PROBE. (Figure 4.4(a)).

2. The head packet of the buffer is a TCP packet that does not fit fully inside the padding: MGRP fragments the TCP packet (details in section 4.6) and adds the TCP fragment as a second MGRP chunk with proto=FRAG. The probe is now fully

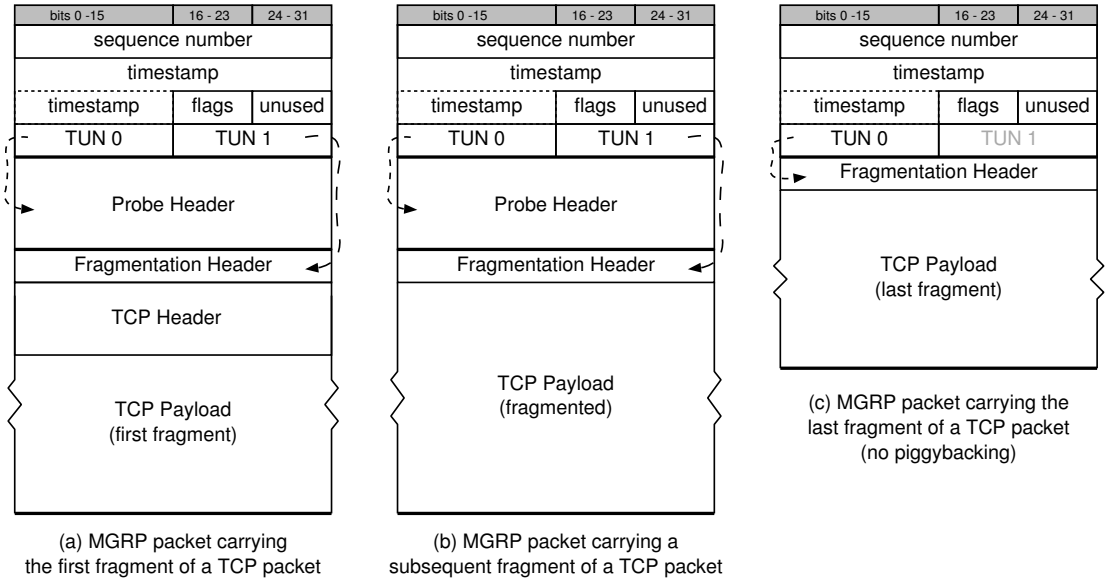


Figure 4.5: Examples of Fragmented Packets

expanded and requires no padding. Piggybacking for this probe is 100%. MGRP puts the remaining TCP fragment back to the payload buffer for the next probe We discuss fragmentation in section 4.6 and an example of this case is shown in Figure 4.5(a).

3. The head packet of the buffer is a TCP packet that has already been fragmented from prior piggybacking: MGRP repeats step 2 for every subsequent probe until it reaches the last fragment that fits fully inside the padding We discuss fragmentation in section 4.6 and an example of this case is shown in Figure 4.5(b).

4. The head packet of the buffer is an intact TCP packet (not already fragmented) and it fits fully inside the padding: MGRP adds a second MGRP chunk with protocol TCP and expands the packet, but with less padding than before (Figures 4.4(b) and 4.4(c)).

There are cases when TCP fragments need to go out by themselves inside MGRP packets and not piggybacked on probes. This may happen when there is a TCP fragment at the head of the payload buffer (as a result of case 2) and one of these two events occur: (a) MGRP finishes sending all the probes of a probe transaction, or (b) the virtual timer of the TCP packet where the fragment belongs expires. In either of these cases MGRP needs to send out the remaining TCP fragment immediately because of the MGRP timing guarantees. The generated MGRP packet contains one MGRP chunk that holds the TCP fragment with proto=FRAG. We discuss fragmentation in section 4.6 and an example of this case is shown in Figure 4.5(c).

4.5.1 TCP with MGRP Piggybacking

We have modified the Linux TCP implementation slightly to allow its packets to be piggybacked. The changes to the TCP code are so that TCP can pass its outbound traffic to MGRP, which will then pass it to the IP layer (whether or not it is piggybacked).

After the application writes its data on the TCP socket, TCP constructs packets and runs its algorithm as usual, and when it is ready to transmit a packet, TCP calls the MGRP transmit callback instead of the IP callback. Since the callback is stored in a function pointer, the changes to the TCP kernel code do not exceed 50 lines, with most of the lines devoted to servicing the new socket option that we need to introduce.

Applications opt-in to MGRP by using a new TCP socket option (Figure 4.6). Once turned on, all the application packets sent on that socket pass first from TCP and then they are passed to MGRP, where they may get piggybacked on probes or sent out standalone.

```
/* Turn on TCP over MGRP */

sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)

on = 1
setsockopt(sock, IPPROTO_TCP, TCP_MGRP, &on, sizeof(on))
```

Figure 4.6: Code sample for turning on the TCP socket option that applications use to opt-in to MGRP. Applications can use the same code prevent their packets to go through MGRP.

We chose the opt-in, rather than opt-out, approach because piggybacking on MGRP alters a packet's delivery characteristics. Thus, for now we imagine that applications must be aware that their traffic is being piggybacked. However, we can imagine that under some configurations (e.g., for short buffering timeouts), piggybacking by default is a reasonable option.

The changes to TCP are only needed on the sender side. On the receiver side, the MGRP packets are received from the network subsystem directly by the MGRP module. After MGRP extracts the piggybacked payload, it uses the TCP receive callback, exactly like IP does in the non-MGRP case (so no changes required to TCP).

4.5.2 MGRP Piggybacking Statistics

When MGRP sends a probe transaction it keeps track of statistics that a prober can query from userspace using an `ioctl`. We described in Section 3.1 and Table 3.1 the information that MGRP tracks. For every probe train MGRP records (i) the amount of total piggybacking in the train, and, most importantly (ii) the amount of piggybacking that is due to TCP packets that would have been interspersed with the probes if there was no piggybacking. The latter amount can allow probing tools to adjust their estimates to account for source traffic they would otherwise have competed with. Consider for

example the following MGRP output:

```
                <pkts>    <bytes>
total             30      12000
pbk               26       7696
pbk_compete       16       4752

duration actual        8729708 nsec
duration precomputed   8700000 nsec
```

In this example an application sent a probe train with 30 400-byte probes (30x400=12000 bytes) of which 26 probes were piggybacked. Out of the 26 piggybacked probes, 16 probes are piggybacked from TCP packets that would have been competed with the probe train (if there was no piggybacking). These 16 probes (4752 bytes) can be used to adjust the tool's results.

MGRP gathers this information by pre-computing the duration of the probe train before sending it. From the output above we can see that the precomputed duration is accurate enough. MGRP then sends the probe train and for every piggybacked payload it tests whether the source TCP packet would have been sent within that duration window. If yes, then it should be counted against the `pbk_compete counter`.

4.6 Fragmentation and Reassembly

In order to piggyback TCP payload on probes, MGRP may need to fragment TCP packets (as we described in section 4.5, case 2). MGRP uses a scheme similar to IP fragmentation [42, 43, 44]. In this section we describe in detail how MGRP performs this fragmentation and how the MGRP receiver reassembles the packet.

On the MGRP sender each TCP packet that needs to be fragmented is assigned a

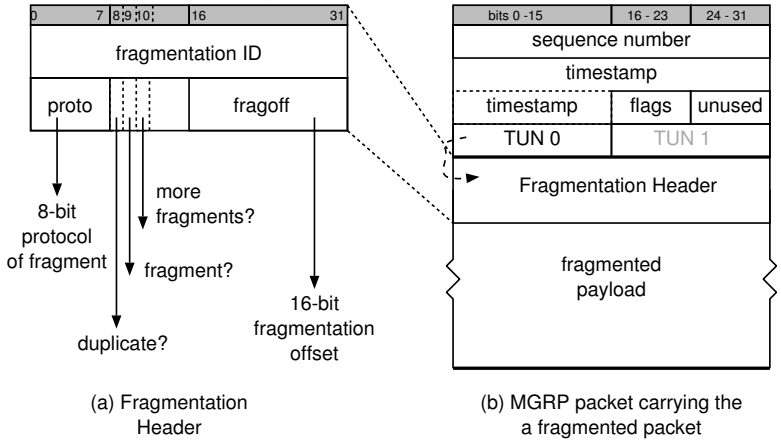


Figure 4.7: The MGRP Fragmentation Header

unique identifier. MGRP then extracts a portion of the TCP packet, calculates the fragmentation offset and fills a fragmentation header, whose fields are shown in Figure 4.7. MGRP then adds the TCP packet portion along with the fragmentation header inside an MGRP packet. Figure 4.5(a) shows an MGRP packet that contains the first fragment of a TCP packet. For subsequent fragments, MGRP repeats the same process but uses the same fragmentation identifier so that all the fragments can be reassembled at the receiver. Figure 4.5(b) shows an MGRP packet that contains a subsequent TCP fragment.

On the receiver, MGRP extracts the fragment, removes the fragmentation header and performs packet reassembly. MGRP stores the fragment in a hash table keyed on the unique fragmentation ID. For every fragment it adds, it checks whether it can reconstruct the whole packet; if yes, it reassembles the packet and pushes it to TCP. Exactly like IP, MGRP needs to have a reassembly timeout. RFC 1122 [44] indicates a reassembly timeout must be fixed and recommends a value between 60 and 120 seconds. Current Linux implementations use a value of 30 seconds [3] and MGRP uses the same value. If the

[3] In Linux use `/proc/sys/net/ipv4/ipfrag_time` to access/configure the reassembly timeout.

packet is not reassembled within 30 seconds of receiving the first fragment, then MGRP discards all the collected fragments.

MGRP uses a fragmentation and reassembly method that is agnostic to the kind of packet that is fragmented. We now only use TCP with MGRP, but the fragmentation scheme can be applied without changes to UDP. Also MGRP does not need to keep any special state for multiple TCP flows. The only state needed is the reassembly hash table which stores packet fragments and reassembles each packet independently of any other.

4.6.1 The Reassembly Bucket

The reassembly code stores the fragments in the hash table as they arrive. Actual reassembly is performed by a "reassembly bucket." The hash is used to locate the bucket that corresponds to each fragment. Each bucket is keyed by: (a) source address, (b) fragmented packet protocol (TCP, UDP, etc.), and (c) fragment ID. The reassembly process is conceptually simple: add the new fragment to the fragment queue for that fragment ID, and, if all fragments are present, reassemble the packet and push up to the next protocol.

Reassembly is straightforward if all the fragments arrive in order, but must consider the possibility that packets will be lost or reordered. MGRP deals with this situation by maintaining a list of *gaps* that need to be filled before the packet can be successfully reassembled. The method using gaps is an existing IP reassembly algorithm [43] (the existing algorithm uses the term *holes* instead of *gaps*). MGRP implements this algorithm with some adjustments to deal with the the peculiarities of MGRP.

The first gap in the list (representing the tail end of the packet) starts as open-ended

since MGRP does not know, until the last fragment is received, the size of the reassembled packet. As fragments come in MGRP adjusts the gap list by shortening, removing or splitting gaps, based on the offset of the received fragment. A fragment that is marked as “last” always matches an open-ended gap. That is how the original gap becomes close ended and eventually disappears. When the gap list becomes empty, MGRP is ready to reassemble the packet.

In each reassembly bucket, in addition to the list of gaps, MGRP maintains a list of fragment buffers. Each gap in the gap list maintains back-pointers to the fragments in the fragment list that fill the data around the gap. Figure 4.8 shows how the reassembly process works. With this approach, MGRP can keep the fragment list ordered as an indirect consequence of maintaining the gap list. When a fragment arrives, MGRP finds which gap it “plugs”. There are four cases:

1. **The new fragment fills the left side of the gap:** MGRP inserts the new fragment in the fragment list after the left backpointer and make the new fragment the left backpointer of the gap. The gap remains in the gap list but shrinks in size and its left offset increases (Figure 4.8(b)).

2. **The new fragment fills the middle portion of the gap:** This can happen when fragments are lost or delivered out of order. MGRP inserts the new fragment after the left backpointer and splits the gap into two new smaller gaps. The left gap retains the left backpointer of the original gap and points its right backpointer to the new fragment. Similarly for the right gap (Figure 4.8, cases (c) and (f)).

3. **The new fragment fills the gap completely:** MGRP uses the backpointers to insert

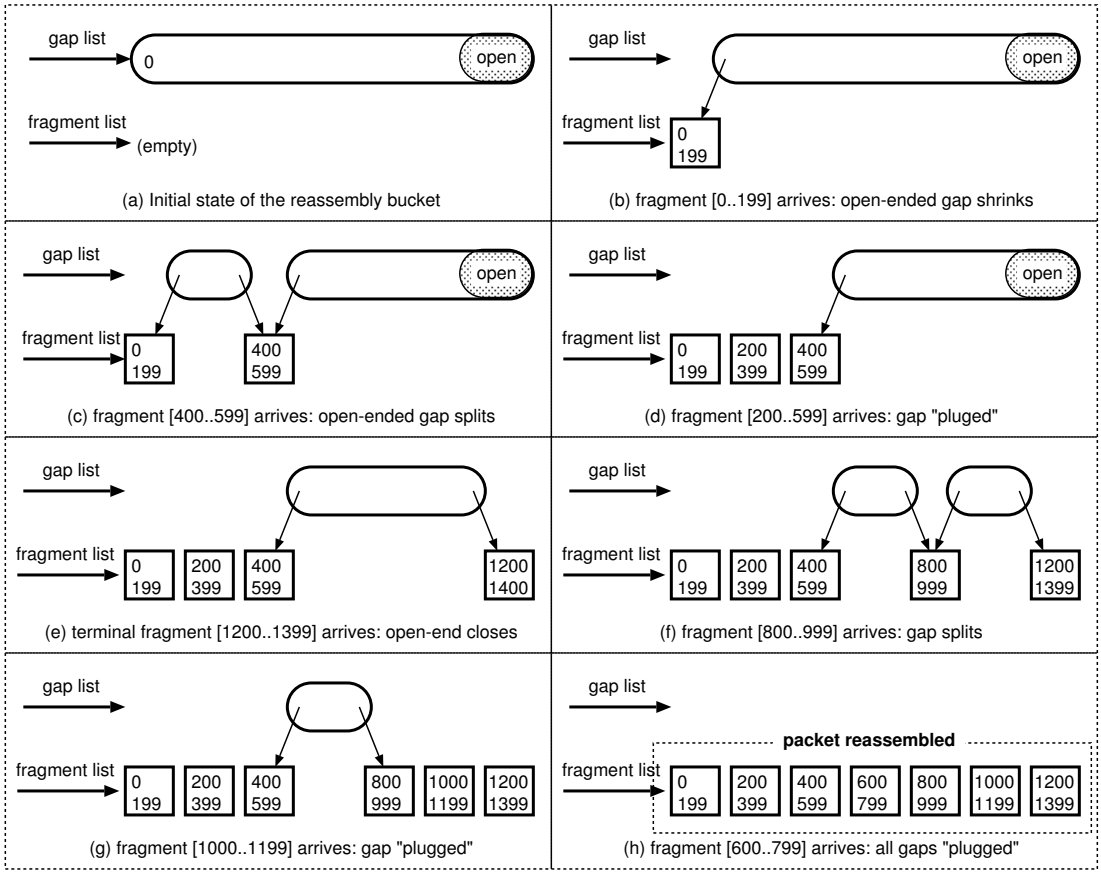


Figure 4.8: The Reassembly Process

the new fragment into the fragment list. Since the left fragment backpointer is for the fragment on the left of the gap, and the right backpointer for the fragment on the right of the gap, the new fragment is inserted in the proper sorted position in the fragment list. MGRP then removes the gap from the gap list. This is the only case where the gap list shrinks, which brings us closer to being able to reassemble the packet (Figure 4.8, cases (d) and (g)).

4. **The new fragment fills the right side of the gap:** The gap and fragment lists are updated similar to the left side case (case 1). A special case is when the right side of the gap is open ended, which is only matched by the last fragment. In that case,

ID	FLAGS	Fragments	Status
3984	R.SF.	7	OK: reassembled 7 fragments and sent to TCP
3985	R.SF.	3	OK: reassembled 3 fragments and sent to TCP
3986	R.SF.	4	OK
3987	R.SF.	7	OK
3988	R.SF.	6	OK
3989	.....	5	LOSS: cannot reassemble, 5 fragments received
3990	R.SF.	7	OK
3991	.....	3	LOSS: only 3 fragments received
3992	R.SF.	7	OK
3993	.....	2	LOSS: only 2 fragments received
3994	R.SF.	7	OK
3995	.....	2	LOSS: only 2 fragments received
3996	R.SF.	7	OK
3997	.....	2	LOSS: only 2 fragments received
3998	R.SF.	7	OK
3999	R.SF.	4	OK
4000	R.SF.	7	OK
4001	R.SF.	4	OK
4002	R.SF.	6	OK
4001	.....	1	reassembly in progress, 1 fragment received

Table 4.2: A view of MGRP's reassembly hash table in operation. Each line represents one reassembly bucket and corresponds to one packet being reassembled. The first column contains the Fragment ID, the second contains flags that indicate the status of the reassembly process and the third column shows the number of fragments that have already been received. Flag **R** means packet was reassembled, Flag **S** means packet was delivered to TCP, Flag **F** means packet was reassembled from fragments. So any entry that appears as `R.SF.` means that packet has been successfully reassembled (dots indicate missing flags). Any entry that appears as `.....` means that it may have received fragments but reassembly is not possible yet. After a certain timeout these packets are declared lost and the reassembly process is abandoned.

the gap becomes a normal close-ended gap (Figure 4.8(e)).

So the only complexity of the algorithm is to maintain the list of gaps according to these four cases. When MGRP is ready to reassemble the packet, it just needs to traverse the fragment list and combine all fragments into one large packet, since the fragment list is always sorted.

A View of the Reassembly Hash Internals We have instrumented MGRP so that we can, for debugging and diagnostic purposes, freeze a view of all the reassembly hash

buckets at any point of MGRP's operation. Table 4.2 shows such a view, simplified to remove some details that would otherwise clutter presentation.

4.7 MGRP Parameters

MGRP provides many ways to control its operation, as we have described. This section summarizes how tools and applications may fine tune the piggybacking operation and optimize the performance delivered by MGRP.

4.7.1 For tools that send probes

Selectively Turning Off Piggybacking When tools send probe transactions (section 3.1) they have fine-grained control over every probe. We already saw (Figure 3.3) how tools use the Probe API to control the padding and gap of each probe. We now discuss how tools can selectively turn piggybacking on/off for a single probe. This is accomplished through a flag in the ancillary data passed for each probe during send.

The rationale behind the decision to give tools such fine-grained control stems from the fact that tools are best equipped to decide if piggybacking makes sense for that particular probe. We discussed in Section 3.2.2 that while piggybacking decreases the bandwidth wasted with padding it may increase the chance of loss. If a tool knows it is sending *high risk* probes, i.e.,probes with a high probability of loss, then it is counterproductive to piggyback those probes. But MGRP cannot know this automatically so it has to rely on the tool to make that decision. For example, pathload bootstraps by sending a few packet trains at very high rate. Since many of these packets are going to be lost, pathload

turns off piggybacking for the those initial packet trains. Alternatively, a tool can turn off piggybacking for a portion of the packet train, usually the later probes, since those run the highest danger of overflowing the network buffers. We return to this issue in the next chapter (Section 5.3).

Auditing Probe Generation It is very important to tools to be sure that probes are sent with accurate gaps. MGRP provides an auditing mechanism that provides feedback about the relative error in generating probes; this audit was used to produce Table 4.1. Tools can use this information to invalidate a probe train that does not meet some accuracy threshold, or they can incorporate the error in their estimation algorithm (for example, using the actual rate that the probe train was transmitted and not the theoretical one).

4.7.2 For applications that contribute payload

Opting-in to Piggybacking Applications need to opt-in to MGRP. This mechanism works at the socket granularity. An application opens up an MGRP socket and then can turn piggybacking on/off on that socket any number of times during the socket lifetime. This provides control to an application to prevent any kind of alteration of its traffic for any period of time. When an application turns off piggybacking for a socket, it also turns off the payload buffer for that socket, which means that the application packets pass through MGRP without any delay. In Section 4.5.1 we saw how applications that use TCP sockets turn on piggybacking for the packets they send.

Bounding the Delay of Payload Buffering An application may find acceptable the MGRP piggybacking and buffering but may want to impose a lower delay than the global MGRP delay. This can be accomplished on a per-socket basis and can be changed during the socket lifetime.

4.7.3 For global MGRP behavior

Payload Buffer Timeout This is the default fixed delay that MGRP adds to all application packets in order to to increase the chances of piggybacking It can be overridden by the application on a per-socket basis. Setting the global delay to 0 amounts to turning off piggybacking.

Piggybacking Ratio This controls the initial portion of the probe transactions that can potentially get piggybacked. A ratio of 1.0 indicates that all probes are candidates for piggybacking. A ratio of 0.2 indicates that only the first fifth of the probe transaction is eligible for piggybacking. The remaining four fifths cannot be piggybacked even if there are available packets in the payload buffer. The reasoning is that losses tend to happen at the end of packet trains, so this parameter gives us the opportunity to test various piggybacking scenarios.

Chapter 5

Experimental Evaluation

To understand the costs and benefits of MGRP, we carried out several experiments. Our experimental setup aims to simulate the media streaming scenario motivated by the introduction. However, we imagine that this basic scenario could be extrapolated to more interesting settings, e.g., adaptive, streaming media overlay networks [21, 16] and even best-effort services like BitTorrent [45].

We begin in the next section by describing our experimental setup, and the next section reports the results.

5.1 Experimental Setup

The network topology we use is shown in Figure 5.1. In all experiments we transmit a steady 4 Mbps TCP source traffic stream, representing the media stream, from $m3$ to $m5$. 4 Mbps is the rate of high-definition streams for both Apple TV and the Vudu Internet video service [46, 47]. We periodically send probe transactions along the same path, representing (or actually implementing) a measurement tool probing for available bandwidth, while varying certain parameters of the probe transactions (e.g., their length, bandwidth, and spacing). We also measure the effects of using different kinds of cross traffic, which is sent between $c1$ and $c2$, and shares a bottleneck link $x1x2$ with the source and probe traffic. We elaborate on the experimental parameters of the source traffic, cross

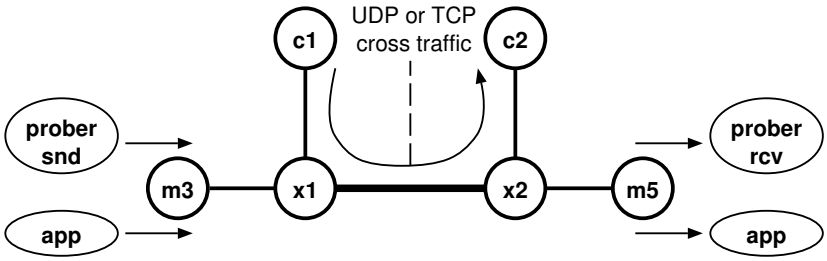


Figure 5.1: Experiment topology. Source and probe traffic goes from $m3$ to $m5$ and cross traffic from $c1$ and $c2$ (UDP or TCP).

traffic, and probe transactions in the remainder of this section.

All experiments were run with and without MGRP enabled, and we measured the performance of the source traffic, cross traffic, and measurement tool. When MGRP was enabled, we used a buffering timeout of 10 ms. We conducted the experiments on Emulab [48], using $pc3000$ hosts for all nodes, which are connected via 100 Mbps Ethernet. We shape the $x1x2$ bottleneck link using the FreeBSD Dummynet [49], limiting the link speed to 10Mbps (typical for a fiber-at-home user). Dabek et al. [34] report that the median RTT between PlanetLab nodes is 76 ms, and between DNS servers worldwide the median RTT is 159 ms. Since higher RTTs lessen the penalty of MGRP's buffering delay, we present all results using a relatively low RTT of 40 ms, roughly the RTT between Washington, D.C. and Kansas City, MO.

5.1.1 Source traffic

For our source traffic, we use $nuttcp$ [50] (a variant of iperf [51]) to generate a constant 4 Mbps stream at the application, which runs over TCP with CUBIC congestion control [52] (NewReno [53] gave similar results). Since our goal is to explain the effects of piggybacking data and measurement traffic, the application writes to TCP at a constant

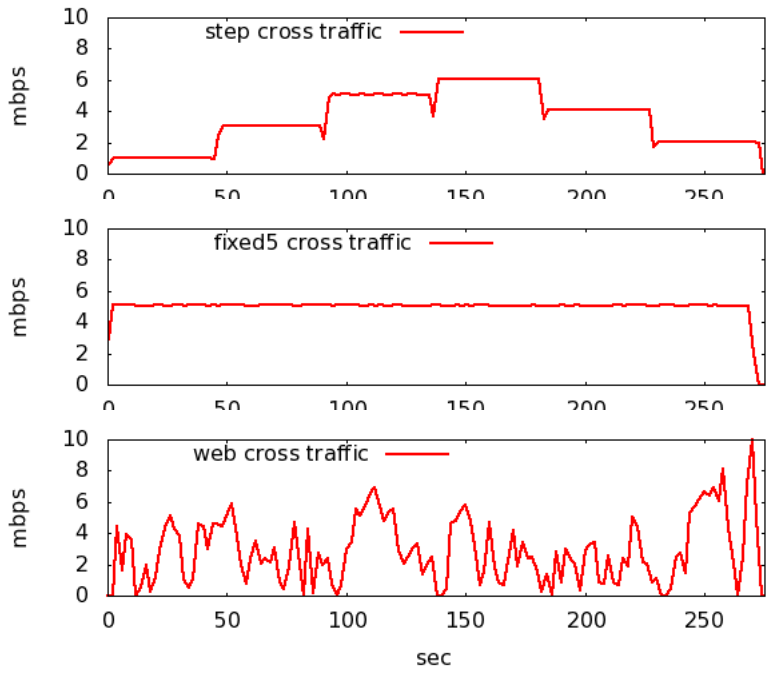


Figure 5.2: The three different types of cross traffic

rate. An actual video streaming implementation might choose to scale down the video and audio rates to avoid passively-detected congestion at some points during the experiments.

5.1.2 Cross traffic

In addition to the congestion-aware source traffic, we generate three types of cross traffic between nodes $c1$ and $c2$, shown in Figure 5.2: (a.) STEP, which transmits in fixed-rate intervals lasting 45 seconds each (1, 3, 5, 6, 4, 2 Mbps), (b.) FIXED-5, which transmits at a constant 5 Mbps for the duration of the experiment, and (c.) WEB, which models web traffic with Poisson-distributed inter-arrival times. UDP-based cross traffic (a. and b.) permits consideration of the effects of piggybacking on source traffic without the added variable the cross traffic's response to induced congestion. STEP provides more of a challenge for measurement tools, while FIXED-5 provides relatively simple

conditions. TCP-based cross traffic (c.) allows us to consider the effects of measurement on the cross traffic's performance.

Both STEP and FIXED-5 use tcpreplay [54] to replay a snapshot of a 150 Mbps trans-Pacific Internet link [55]. Replaying a trace permits us to test against actual Internet packet size distributions and interleavings, but without cross-traffic congestion avoidance, and at rates that are stationary over a reasonable period of time [56]. The WEB cross-traffic uses the NTools [57] suite to generate up to ten concurrent TCP connections, with Poisson-distributed inter-arrival rates over 5-second intervals. Like the source, each TCP flow performs normal CUBIC congestion control. The flow size varies between 64 KB and 5 MB, and is weighted toward flows in the 64-200 KB range. The initial seed to the generator is fixed to provide consistent (though not identical) behavior between runs.

5.1.3 Probe Transactions

We use pathload [6], modified to use the MGRP Prove API, to send probe transactions. Pathload is the gold standard of available bandwidth tools, as it is both robust and fairly accurate [58, 59]. Pathload produces simple packet trains with uniform gaps, and uses more bandwidth than other tools (approximately 5–10% of the residual bandwidth in our experiments), which gave us an opportunity to test probe reuse in a non-trivial way.

As we show in Section 5.2, the original pathload is sometimes slow to return results. We found that between each stream, the pathload implementation waits $RTT + 9 * TX_{stream}$, where TX_{stream} is the amount of time it takes to transmit one stream. In an effort to increase the number of measurement rounds performed by pathload, we slightly

modified pathload to reduce this pause between streams to one RTT. We call this more aggressive version pFAST, and the original pSLOW in our experimental results.[1]

5.1.3.1 Packet Trains

In addition to pathload, we ran some experiments with fixed-sized packet trains. We did this for two main reasons. First, the basic primitive of many active measurement tools is the packet train, so considering it separately from a particular tool's implementation seems useful, e.g., to consider higher-rate measurements that would not be feasible without piggybacking afforded by MGRP. Second, pathload responds to improvements in network conditions by sending more traffic. This can make it somewhat more difficult to evaluate the effects of MGRP vs. the effects of pathload's implementation; considering non-adaptive packet trains makes it possible to separate these effects.

As discussed in Chapter 2, tools that use packet trains often send N probe packets of equal length L within a short interval. The gap G between sending successive packets can be fixed (pathload, yaz, pathrate) or variable (pathchirp). When the gap is zero we have trains of back-to-back packets (pathrate). There are also cases of tools where packet trains are surrounded by special TTL-limited packets (pathneck).

We built a utility we call $pktrain$ to generate packet trains at various configurations using either traditional UDP probes or MGRP probes. The size, timing, and transmission rates of the packet trains in our experiments are shown in Table 5.1. Note that all

[1] Both pSLOW and pFAST perform user-level timing and send over UDP when MGRP is OFF, and use kernel timers and the probe API when MGRP is ON; the only difference is the delay between streams within a round.

	pk1	pk2	pk3
packet length	300	500	300
packets/train	10	20	10
packet gap (usec)	200	1000	0
train gap (msec)	20	20	10
inst. throughput (Mbps)	12.0	4.0	line rate
avg. throughput (Mbps)	1.1	2.0	2.3

Table 5.1: Parameters for packet train experiments

of the packet trains are relatively short (10-20 packets). Each train type demonstrates different likely probing scenarios: *pk1* has a relatively high burst rate (12.0 Mbps) but low average rate (1.1 Mbps), and its ten-packet bursts approximate the sub-trains over which pathload calculates changes in one-way delay; *pk2* has longer 20-packet trains at lower rates (4.0 Mbps), but a higher overall rate (2.0 Mbps) and demonstrates what is feasible with MGRP; *pk3* is similar to capacity estimators such as pathrate, and transmits 10 packets back-to-back with an average probing rate of 2.3 Mbps.

5.2 Results

This section presents the results of our experiments. We present our results by the cross traffic used: first STEP, then FIXED-5, and finally, WEB. For each kind of cross traffic, we use the same 4 Mbps CUBIC TCP source and run several types of measurement algorithms (pSLOW, pFAST, pk1, pk2, pk3) to test the effectiveness of MGRP against measurement without MGRP. The MGRP buffering delay is set to 10 ms, and the network RTT delay is 40 ms.

For each type of cross traffic and measurement algorithm, we perform four runs, discarding any runs that did not complete due to experiment error. We measure the throughput of the source, cross, and probe traffic at the bottleneck link—i.e., after any losses have occurred—over one second intervals. We also measure how long it takes pathload to complete a measurement, when using pFAST and pSLOW, and compare the accuracy of these measurements with and without piggybacking enabled.

In general, the results of our experiments demonstrate that MGRP leads to better application throughput and reduced jitter and is fair to cross traffic. Piggybacking on MGRP reduces the time required for pathload to complete a measurement, allows it to successfully complete measurements more often, and can be properly accounted for so as not to impact measurement accuracy. MGRP also makes it possible to measure at higher rates in conjunction with data traffic, an approach that would not be feasible without MGRP. On the other hand, highly-variable cross traffic (i.e., for WEB) in combination with bursty probe trains can exacerbate source packet losses in some cases (as discussed in Section 3.2.2); in the next section we discuss ways to address this problem by piggybacking more selectively.

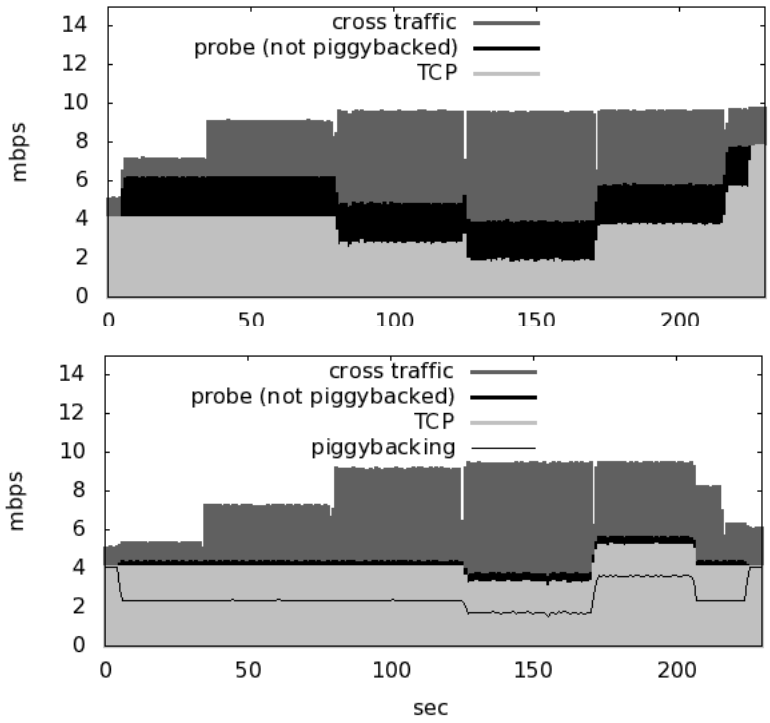


Figure 5.3: STEP: Timeseries plot, with pk2 probes Duration of a single experiment run. The bandwidths of each type of traffic as measured at the 10 Mbps bottleneck link are stacked on top of each other. The top plot is without MGRP, and with MGRP on the bottom. The black line shows the bandwidth of piggybacked TCP segments.

5.2.1 STEP Experiment

The STEP experiment demonstrates all the potential benefits of MGRP.

Improvements to normal traffic

First, MGRP is able piggyback significant amounts application data, nearly eliminating probing overhead. We can see this clearly in Figure 5.3, which plots a single run using pk2 traffic—notice that the bottom plot has almost no black band—and in Figure 5.4, which is the same kind of plot, but for pFAST. Figure 5.5 shows the average throughput of each kind of traffic for the duration of the experiment for all runs. Probe-only traffic appears as a black bar, and riders (“source/probe (pbk)”) contribute to the source throughput. The overhead of piggybacked probe headers is included with “probe

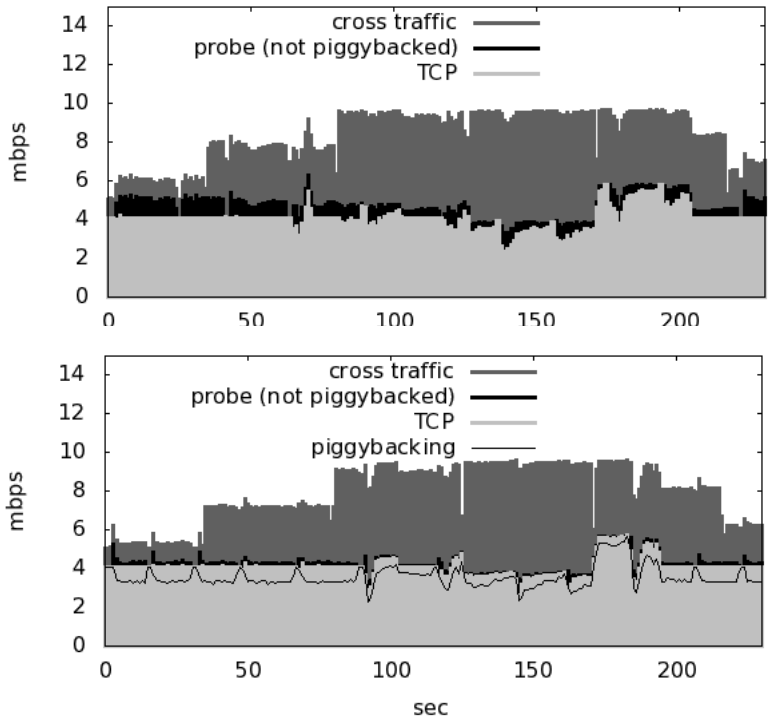


Figure 5.4: STEP: Timeseries plot, with pFAST probes

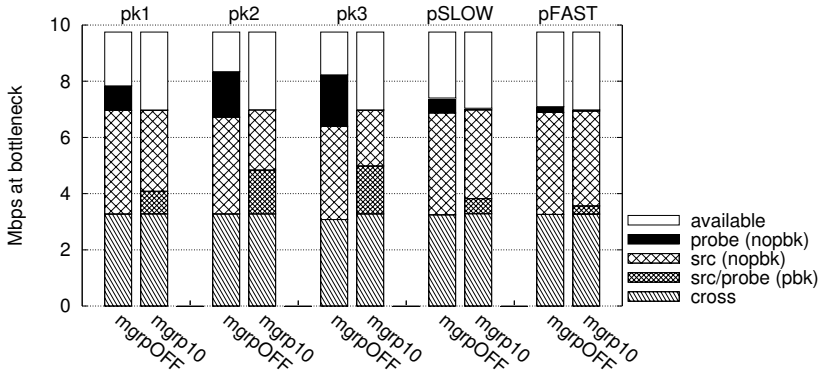


Figure 5.5: STEP: Average per-second throughputs. Each pair of bars represents a different type of probe traffic.

(nopbk)," and is minuscule for most experiments with MGRP enabled.

Second, MGRP "smooths out" source traffic because it competes less with probe traffic. This effect can be seen in the time series plots mentioned above, and also in Figures 5.6 and 5.7. These figures show the CDF of the source traffic throughput measured in

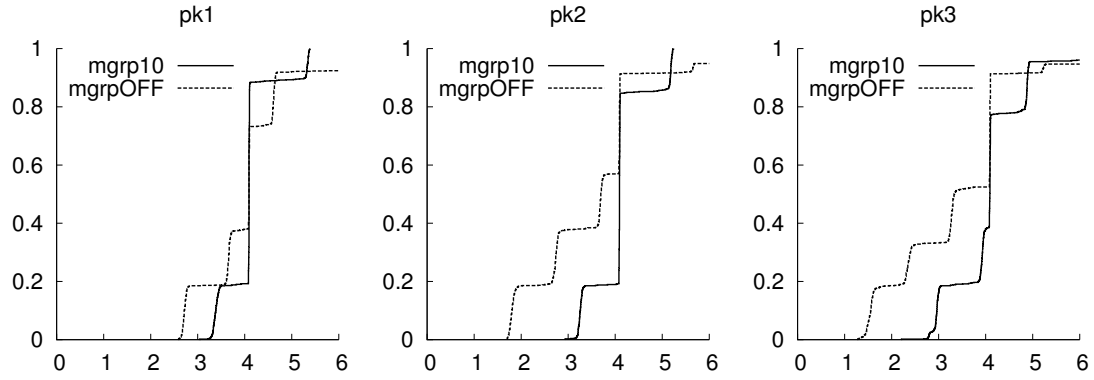


Figure 5.6: STEP: Source throughputs while running packet train probes pk1, pk2, pk3

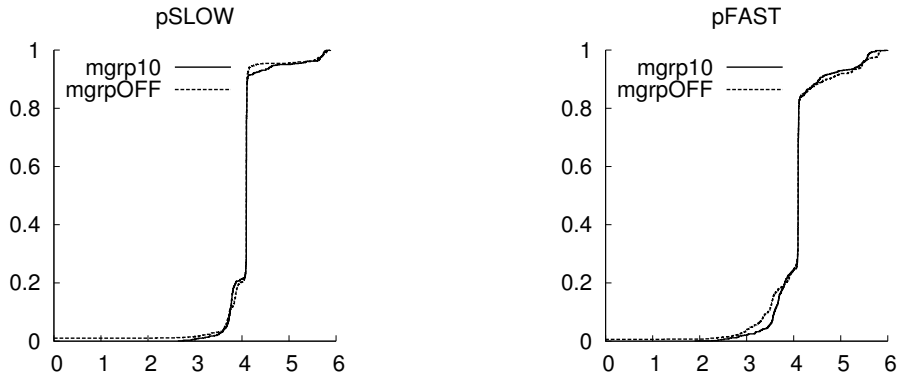


Figure 5.7: STEP: Source throughputs while running Pathload measurements

one-second intervals, for different kinds of measurement traffic. Ideally, the CDF of our 4 Mbps source traffic would be a vertical line at 4 Mbps, indicating that the application was able to sustain a constant 4 Mbps during the entire experiment. Any fluctuation below that rate (to the left) is compensated for by a later return to higher throughputs in slow-start (to the right). These higher throughputs are the result of queuing the application traffic, and are not desirable in a streaming media scenario. Without MGRP, the source rates for pSLOW and pFAST are comparable, but at higher probing rates (as in packet trains pk1, pk2, and pk3), MGRP exhibits a significant improvement.

Finally, MGRP imposes no adverse effect on cross traffic, which sustains the re-

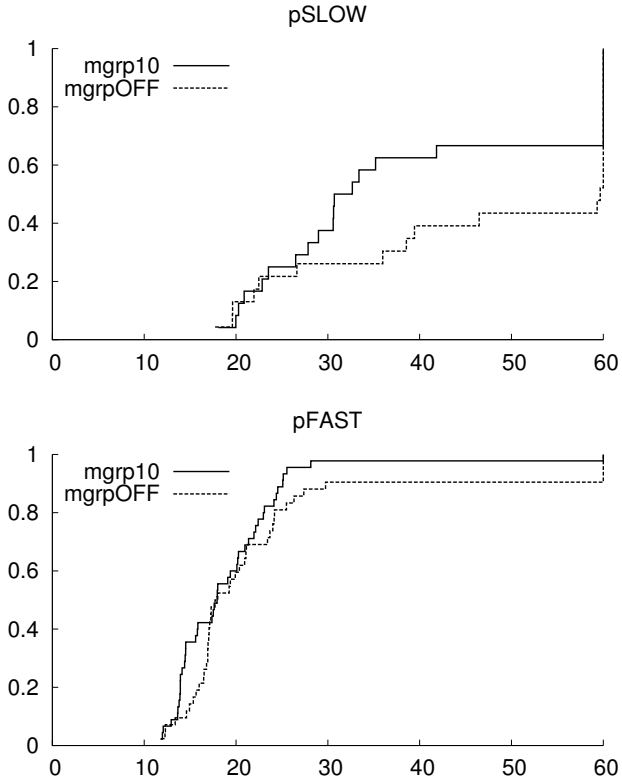
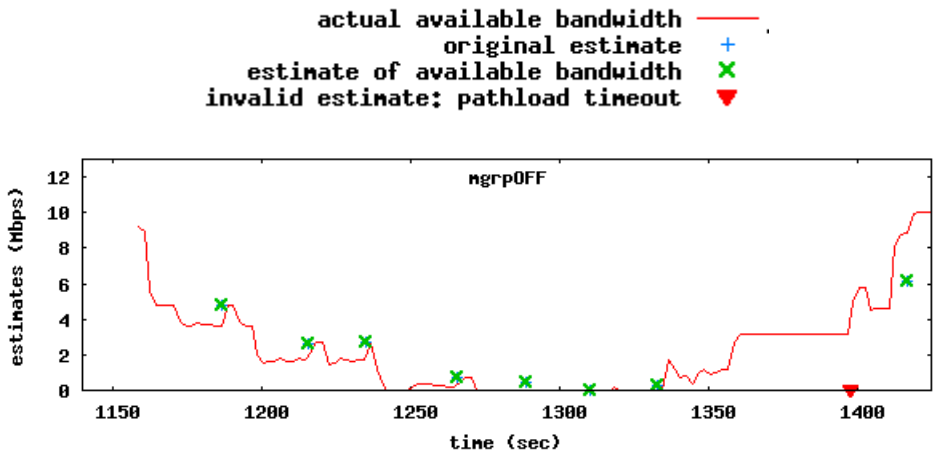


Figure 5.8: STEP: Completion times for pathload measurements.

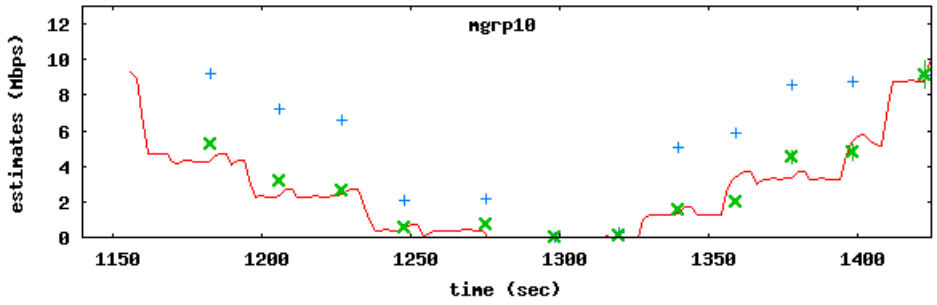
quested rates, with minimal losses. This is clear from Figure 5.5—the cross traffic parts of the bar are fixed for all runs. Any effect on throughput is paid by the congestion-aware TCP source application (as shown in Figure 5.6, just discussed).

Improvements to measurement traffic

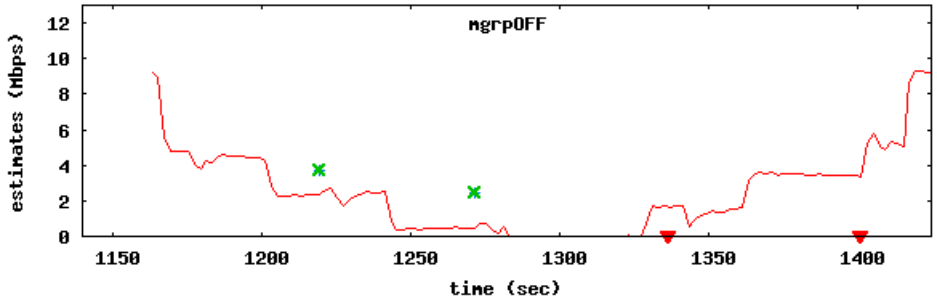
MGRP provides several benefits to measurement traffic as well. First, pathload is able to complete its measurements more quickly and more often, because there is less contention for the link. This can be seen in the CDF plots of pSLOW and pFAST completion times, shown in Figure 5.8. With MGRP, 50% of pSLOW measurements complete within 30.7 seconds, but without it, only 26% complete within that time and 48% fail to complete at all (we time out pathload rounds after 60 seconds).



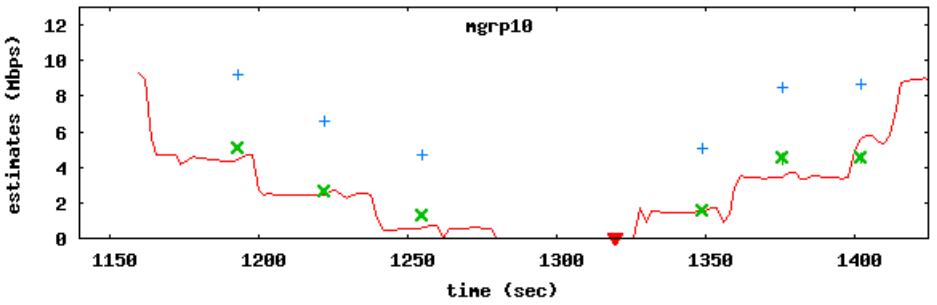
(a) mgrpOFF/pFAST



(b) mgrp10/pFAST



(c) mgrpOFF/pSLOW



(d) mgrp10/pSLOW

Figure 5.9: Pathload accuracy plots for STEP experiment

Second, these improvements do not adversely impact the accuracy of pathload's estimates, once piggybacking is properly accounted for. Figure 5.9 presents for time-series plots that illustrate the available bandwidth on the shared link during an experimental run. The red line shows the actual available bandwidth tabulated by the receiving host on the bottleneck link, based on the traffic it receives. The remaining dots illustrate pathload estimates. We show estimates for pFAST and pSLOW, with and without MGRP piggybacking enabled. We can see here again that pathload is able to complete successfully more often when piggybacking is enabled, and moreover that pFAST completes more often than pSLOW. We can also see that when piggybacking is enabled, pathload will overestimate available bandwidth when not adjusted for piggybacking, but is otherwise as accurate as normal pathload when piggybacking is accounted for.

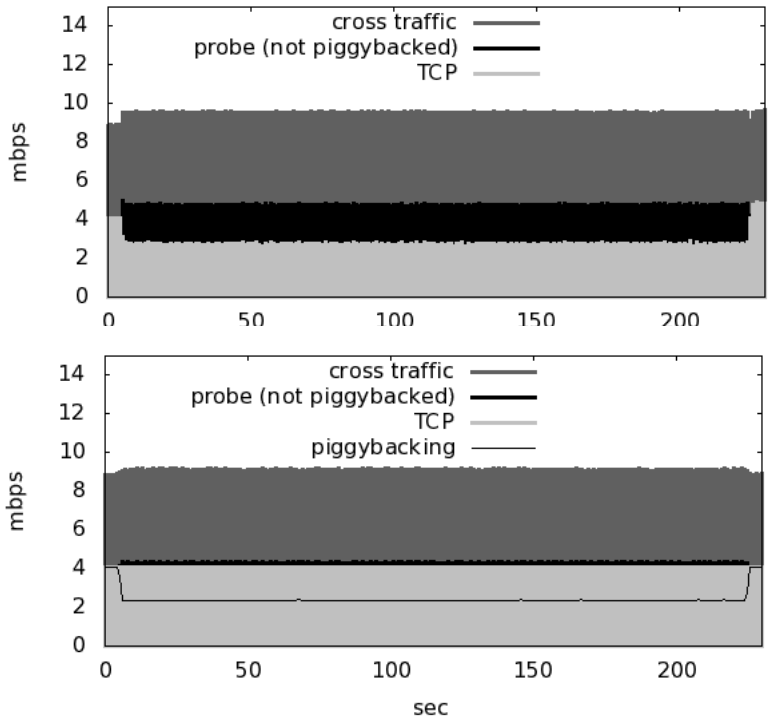


Figure 5.10: FIXED-5: Timeseries plot: pk2 probes

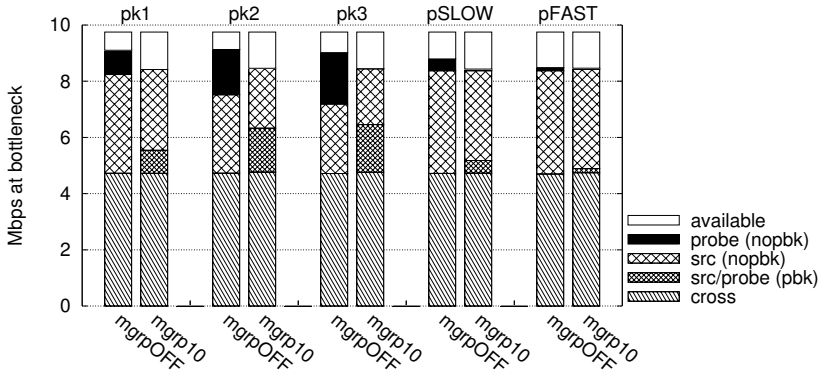


Figure 5.11: FIXED-5: Average per-second throughputs.

5.2.2 FIXED-5 Experiment

As can be seen in Figures 5.10 and 5.11, MGRP again absorbs the measurement traffic into the application traffic, imposing only a slight probing overhead due to probe headers and occasional empty probes.

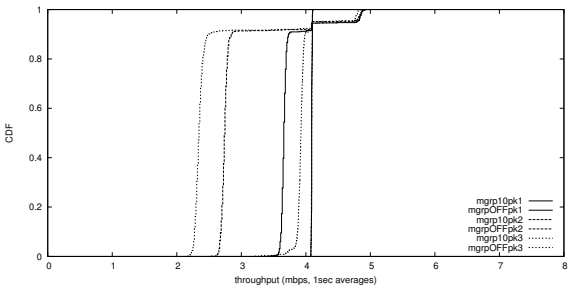


Figure 5.12: FIXED-5: Combined source throughputs for pk1, pk2, pk3

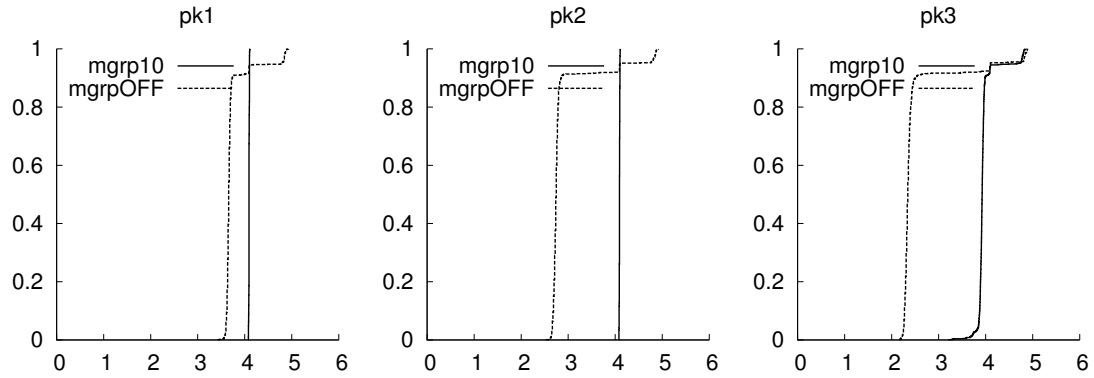


Figure 5.13: FIXED-5: Source throughputs while running packet train probes pk1, pk2, pk3

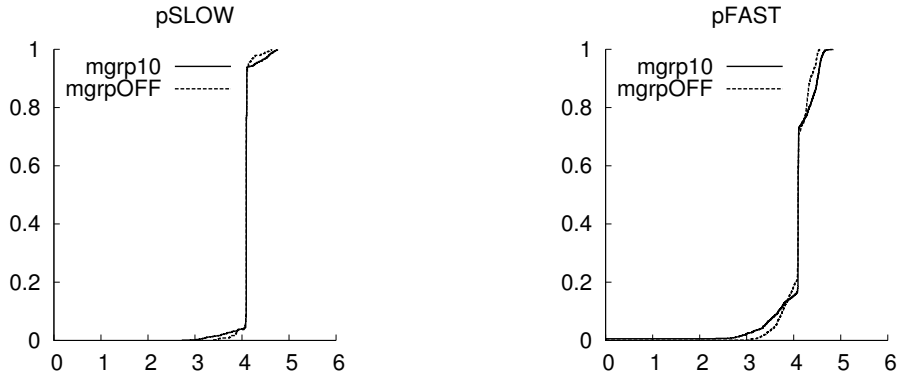


Figure 5.14: FIXED-5: Source throughputs while running Pathload measurements

As shown for pFAST and pSLOW in Figure 5.14, source traffic throughputs are essentially the same with and without MGRP, with a slight advantage to MGRP OFF while running pFAST. With the (pk1, pk2, pk3) packet train runs, however, MGRP shows a clear benefit (Figures 5.10 and 5.13). In fact, as can be seen in Figure 5.12, even with

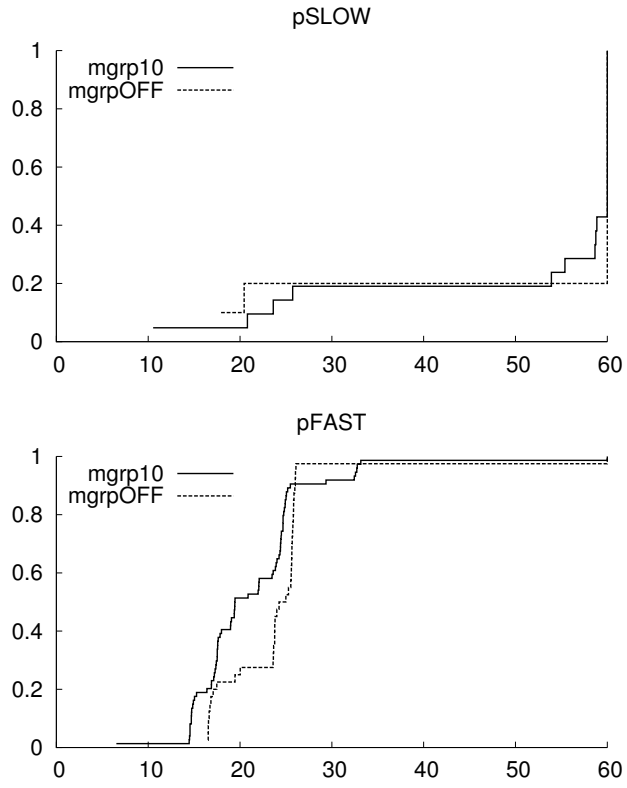
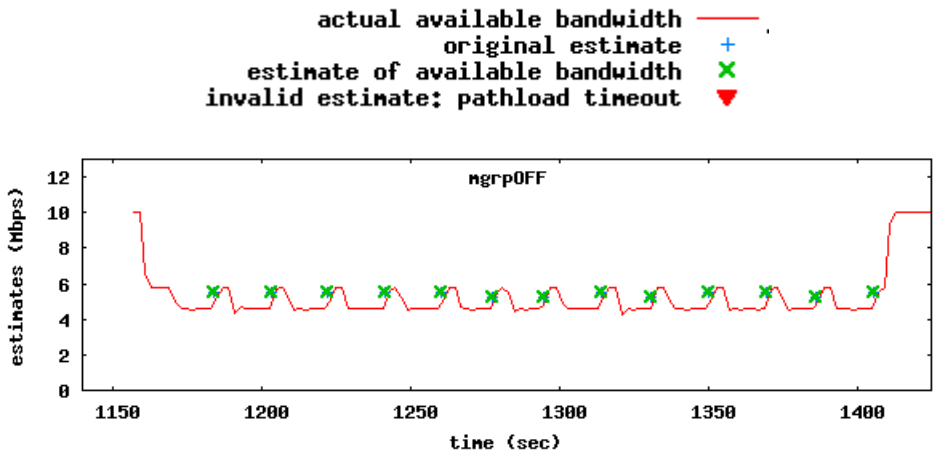


Figure 5.15: FIXED-5: Completion times for pathload measurements.

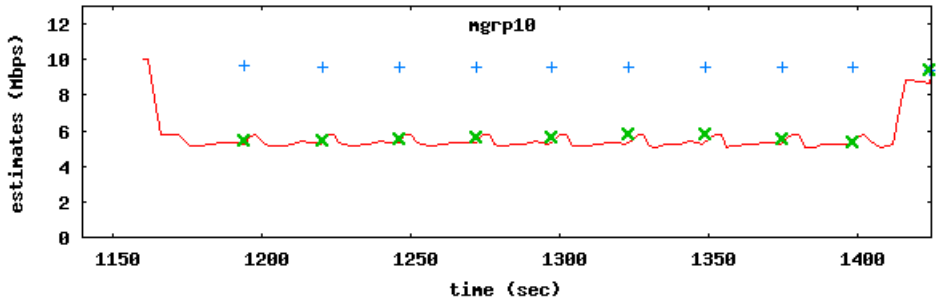
drastically increased probing traffic, MGRP continues to sustain the source traffic, while MGRP OFF penalizes the source traffic heavily. This fact is also shown in Figure 5.11, which shows that without MGRP, as the probe bandwidth increases the TCP source is slowed significantly.

As in the STEP experiments, cross traffic back-off is non-existent with minimal losses for all experiments.

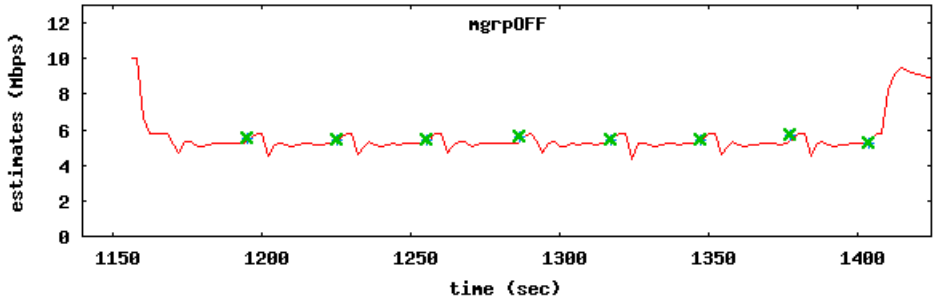
Pathload completion times are similar both with and without MGRP. This is largely due to the ease with which pathload can detect a constant amount of cross traffic. Note, however, in Figure 5.15 that without MGRP there fewer complete pathload rounds. Finally, we note that Figure 5.16 again illustrates that MGRP-enabled pathload produces



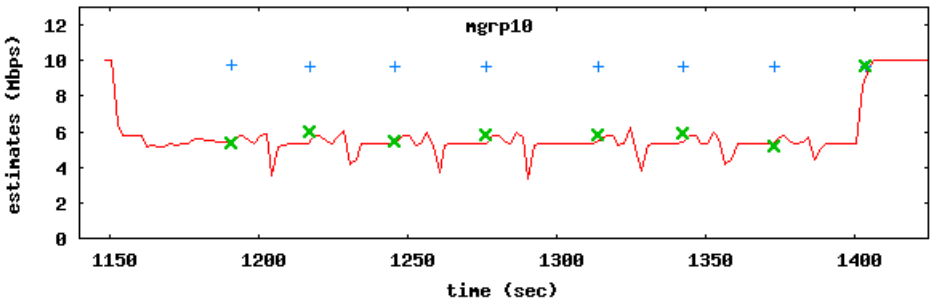
(a) mgrpOFF/pFAST



(b) mgrp10/pFAST



(c) mgrpOFF/pSLOW



(d) mgrp10/pSLOW

Figure 5.16: Pathload accuracy plots for FIXED experiment

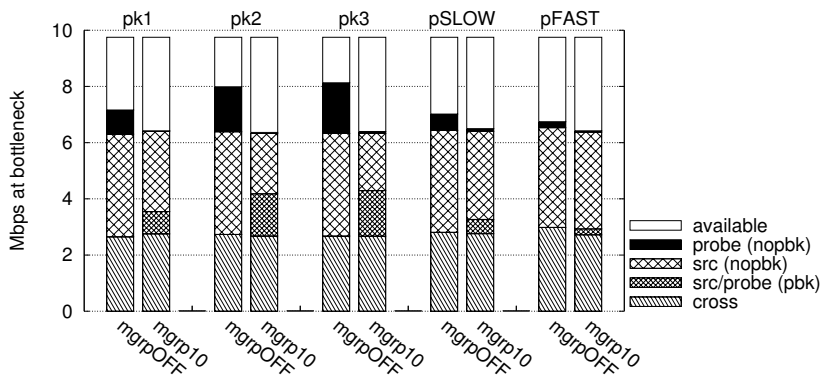


Figure 5.17: WEB Average per-second throughputs. Each pair of bars represents a different type of probe traffic.

accurate estimates when piggybacking is properly accounted for.

5.2.3 WEB Experiment

Unlike the STEP and FIXED-5 experiments, the WEB cross traffic uses TCP sources. We may expect that in the presence of probing traffic with high burst rates, the cross traffic suffers. Somewhat unexpectedly, the cross traffic suffers little with or without MGRP, and the overall rates are comparable. Overall, the cross traffic throughput does not vary greatly between experiments.

As can be seen in Figure 5.17, average throughput across the whole experiment is the same with both MGRP on and off. Figure 5.18 illustrates that MGRP exhibits slightly more consistent source throughput than MGRP OFF for the packet train experiments, thus exhibiting less jitter. However, as can be seen by the CDF plots, the source does not fare as well with MGRP and pFAST: over 40% of the time, the application is unable to sustain its target rate of 4 Mbps, but without MGRP the application pays a penalty only 20% of the time (Figure 5.19). With pSLOW, the MGRP penalty is less severe but still exists. We

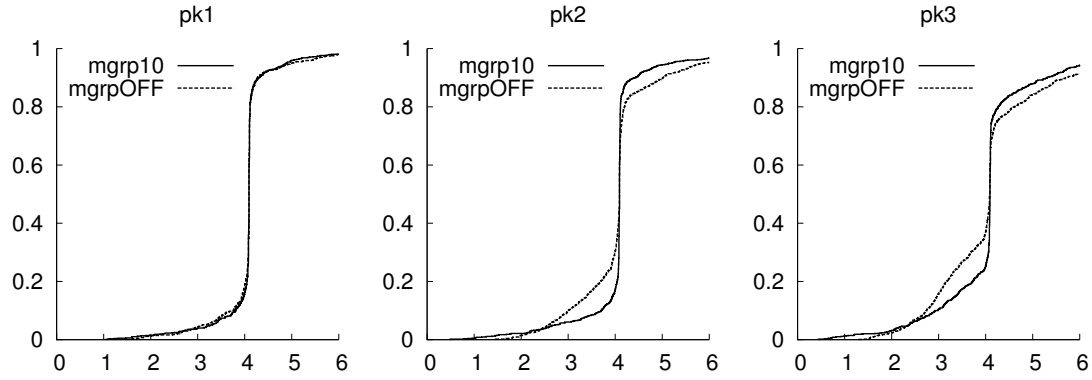


Figure 5.18: WEB: Source throughputs while running packet train probes pk1, pk2, pk3

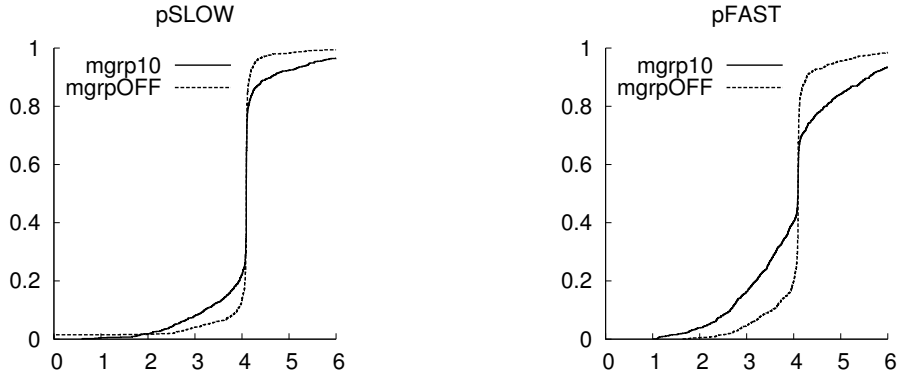


Figure 5.19: WEB: Source throughputs while running Pathload measurements

discuss ideas for improving performance in this situation in the next section.

As in other experiments, MGRP greatly improves measurement completion times (Figure 5.20). With MGRP, pFAST measurements complete 20-25% faster at each of the 25th, 50th, and 75th percentiles, with even more significant improvements for pSLOW. Average pathload probing rates are also 20-25% higher with MGRP. Finally, we can see from Figure 5.21 that once again adjusted MGRP-enabled pathload estimates are still quite accurate, despite the highly-variable cross traffic. We can also see that for these runs MGRP-enabled pathload is able to complete more often.

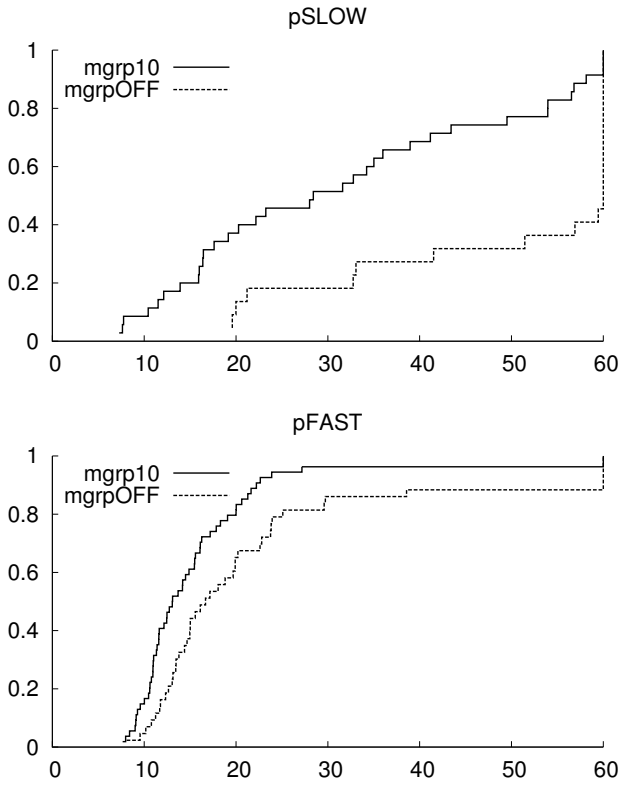
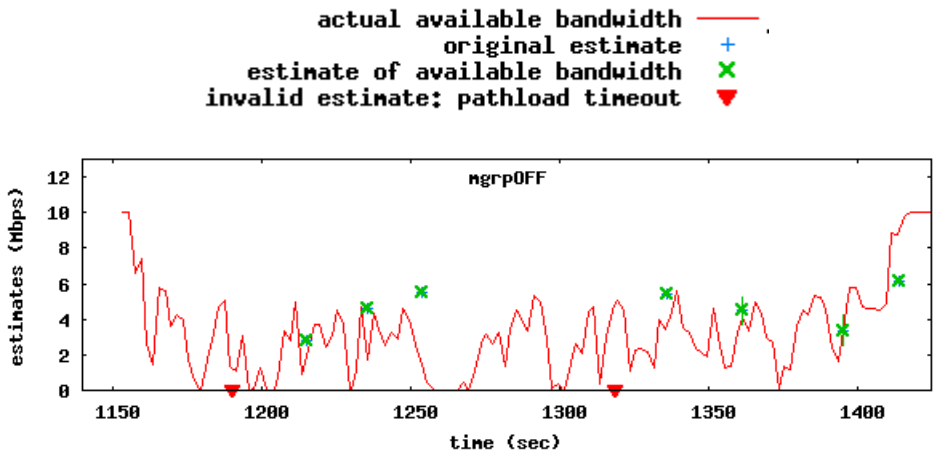
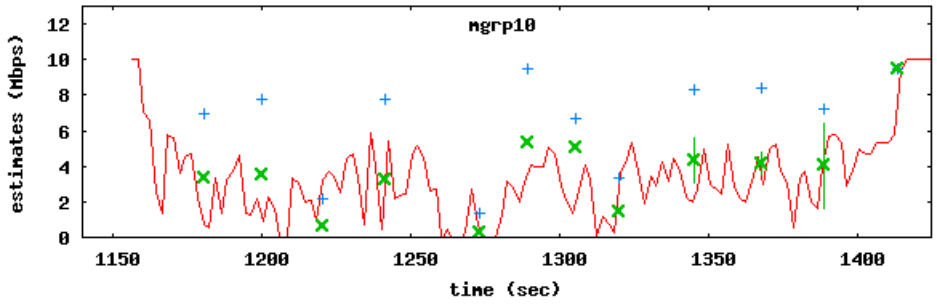


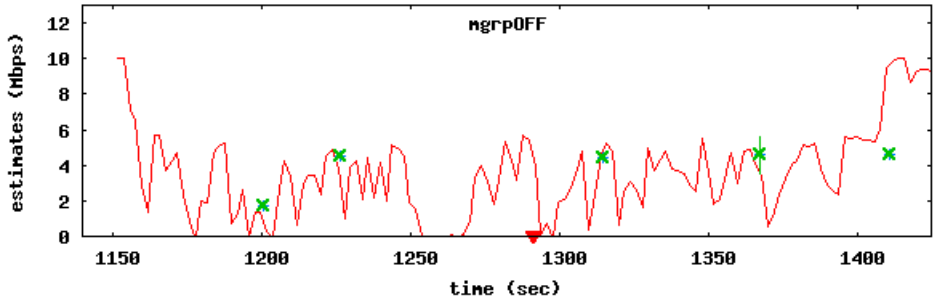
Figure 5.20: WEB: Completion times for pathload measurements.



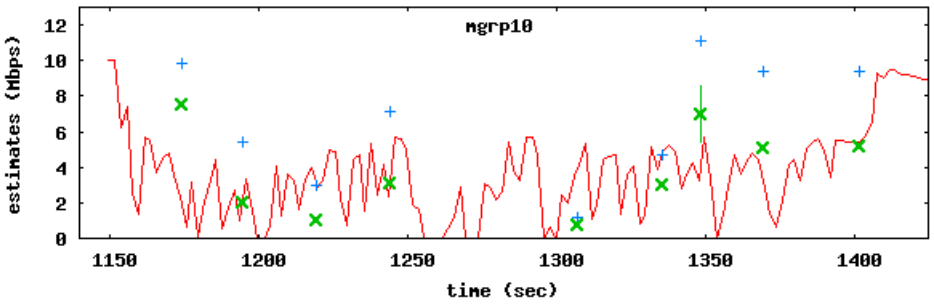
(a) mgrpOFF/pFAST



(b) mgrp10/pFAST



(c) mgrpOFF/pSLOW



(d) mgrp10/pSLOW

Figure 5.21: Pathload accuracy plots for WEB experiment

5.3 Discussion

As shown in the previous section, in most situations MGRP enables higher rates of probing without impacting source or cross traffic, and greatly improves measurement completion times overall. However, in the presence of the WEB cross traffic, using MGRP penalizes the source traffic heavily.

What our discussion so far has not taken into account is the pattern and timing of loss events. Since the WEB cross traffic is highly variable (see Figure 5.2), pathload is often too aggressive in increasing its rates; information gleaned from earlier trains quickly becomes outdated, and higher-rate trains are transmitted into a more congested network.

As we mentioned in Section 3.2.2, piggybacking can eliminate measurement-induced losses when it reduces overhead below the available bandwidth (Figure 5.10), or drastically reduce the occurrence of losses (Figure 5.3), thus improving source throughput and latency. However, piggybacking creates a shared fate between application and measurement traffic, and can actually make source losses worse when the probing burst rate is too high (Figure 5.19).

We can see this effect in Figure 5.22, which displays a dot for each pathload probe. Each large grey rectangle shows a complete pathload measurement, which consists of many probe rounds. Successful probes are light grey, dropped probes without riders (nopbk) are medium grey, and dropped probes with riders (pbk) are black. These black dots are application data losses, and cause TCP congestion avoidance to kick in, lowering throughput and increasing latency. As is apparent from the plot, source losses are highly correlated in particular probe trains (those at higher rates). They also tend to fall toward

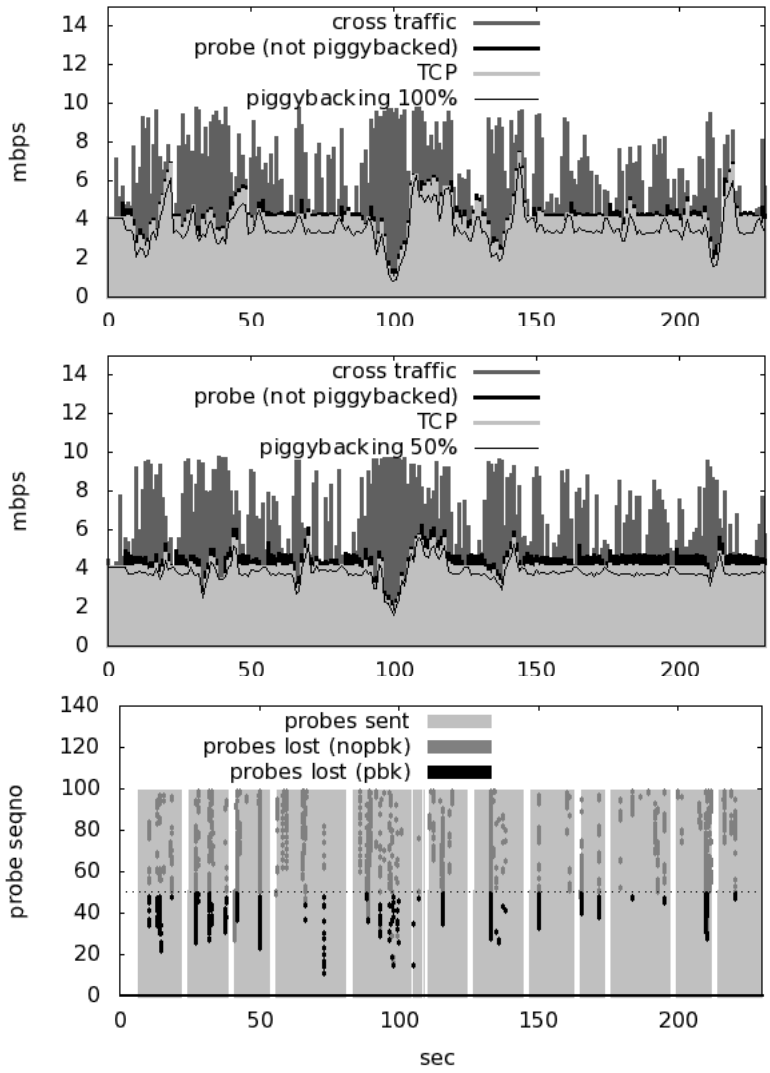


Figure 5.22: Effect of the piggybacking ratio on losses of the source traffic. The top figure shows 100% piggybacking and the middle shows 50% piggybacking. The bottom figure shows the losses inside the packet trains for the 50% case. Losses only to probes is shown in gray, losses to both source and probes (the piggybacked probes) are shown in black.

the end of probe trains (higher sequence numbers), when the bottleneck queue is already full of earlier probes and cross traffic.

We experimented with two approaches to avoid this problem: making the measurement tool piggybacking-aware, and an MGRP regulator which limits the number of probes we piggyback on.

In the first case, we modified pathload to selectively enable piggybacking on probes (via the Probe API) on rather than always setting the piggybacking flag in the call to `sendmsg()` (Section 3.1). Our modified version (run in all experiments with MGRP on) disabled piggybacking during the initial "testing the waters phase" when pathload sends several high-rate bursts. We considered having pathload avoid piggybacking on trains that are on the high side of its binary search [6], or selectively piggybacking only the first portion of those higher-rate trains, but implemented a more general solution instead.

In the second case, we added some policy tuning knobs inside of MGRP itself. One knob controls the maximum percentage of a train that MGRP will add riders to. By reducing the amount of the probe train that carries riders, we trade off data loss in probe trains against the added congestion of competing traffic. Piggybacking on the first 50% of the train seems to improve measurement times significantly while have a minor additional impact on source traffic. However, a piggybacking-aware measurement tool would likely be able to optimize further.

Our coarse piggybacking percentage implemented inside of MGRP is probably not the optimal approach, since shorter probe trains (10-20 packets) will probably pay too high a penalty. The MGRP regulator could instead fix a number of probes it is willing to permit riders on (say, 50 consecutive probes in a transaction, rather than 50%), or calculate the number based on a TCP source's current congestion window.

Chapter 6

Case Study: MediaNet Overlay

Application layer overlays form a network of nodes on top of the physical network so that they can provide a service that is either not available at the network layer or is not provided with an adequate quality of service. For example, NICE [60] is an overlay network for implementing scalable multicast, and RON [8] provides resilient packet delivery services by routing around network-level route failures it discovers through monitoring the paths between end hosts. To provide their service, overlays need to measure the network during both their construction and their maintenance phases. Since they also continuously send data traffic, overlays are ideal candidates for optimization through the inline measurement that the Measurement Manager enables.

In this chapter we will demonstrate the utility of the measurement manager by using it to enhance the functionality of MediaNet [21], an overlay that aims to provide adaptive, user-specified quality-of-service guarantees for media streams. MediaNet's original design incorporates only passive measurement so it suffers from the problem mentioned in the introduction: It fails to assess directly whether more bandwidth has become available along a path, and so its schedulers may make poor routing decisions. To address this problem, we augment MediaNet's schedulers with available bandwidth estimates measured by pathload running over MGRP, where pathload instances are organized into a *estimation overlay* network. By configuring this network alongside MediaNet's overlay, measure-

ment sessions are able to seamlessly reuse MediaNet traffic when it is available. We show that this approach leads to better overall performance than the original MediaNet and MediaNet augmented with a measurement overlay that does not use MGRP.

The section begins with an overview of MediaNet as it currently works and its shortcomings in detecting newly-available bandwidth. It then describes our Estimation Overlay network for gathering available bandwidth estimates and how MediaNet was modified to take advantage of them. Finally we present a series of experimental results demonstrating the benefits of our approach.

6.1 MediaNet

MediaNet [21] is a forwarding overlay that delivers media streams. Each user submits to MediaNet's *global scheduler* (GS) a request that specifies the source of a media stream along with several alternative means of delivery, each tagged with a decreasing level of *utility*. A configuration's utility level indicates the user's relative preference between configurations, from the most to least desired. A typical configuration will specify at utility level 1.0 (the most desired) that a media stream be sent using the highest possible bandwidth, while increasingly lower levels of utility will reduce bandwidth requirements. In the original MediaNet paper and in the experiments conducted here, a full-bandwidth MPEG-2 stream is specified at utility 1.0, while utility 0.5 allows its B frames to be dropped if needed, nearly halving its bandwidth requirements, while utility 0.3 allows both P and B frames to be dropped, more than halving its bandwidth requirements yet

further.[1] MediaNet is agnostic about what adaptations are used, so long as lower utility levels reduce bandwidth needs. For example, the user could specify the stream be transcoded to a smaller frame size, or sent using a different, lower-bandwidth CODEC.

The MediaNet overlay network itself consists of a series of nodes, each of which runs a single *local scheduler* process (LS). The global scheduler combines all user requests together and schedules them along the overlay network in a manner that attempts to maximize each user's utility. In the best case, the GS will be able to deliver all streams with utility 1.0. Usually it will achieve this by combining streams into an application-layer multicast tree, when possible, or by routing streams around congestion spots. Otherwise, some streams may need to be reduced to utility < 1.0, by applying the specified adaptations. Both of these approaches are shown in Figure 6.1.

Once the GS has produced a global schedule, it reconfigures the local schedulers to deliver, and as necessary transform, packets along the chosen overlay paths. In addition to forwarding or adapting the traffic they receive, the LSes also report some statistics to the GS. The GS uses these statistics to determine if a LS reconfiguration is needed to maximize utility under current network conditions.

MediaNet uses TCP for its transport and maintains one TCP connection between each two nodes that form an overlay link. MediaNet operates by selecting how the packets

[1]For an MPEG-2 video stream, I frames are essentially JPEG pictures, while P frames and B frames exploit temporal locality, including "deltas" from adjacent frames. P frames rely on the most temporally-recent P or I frame, and B frames rely on the prior I or P frame, and the next-appearing I or P frame. Therefore, I frames are more important than P frames, while B frames are the least important. Most MPEG decoders are implemented so that they can tolerate lost frames.

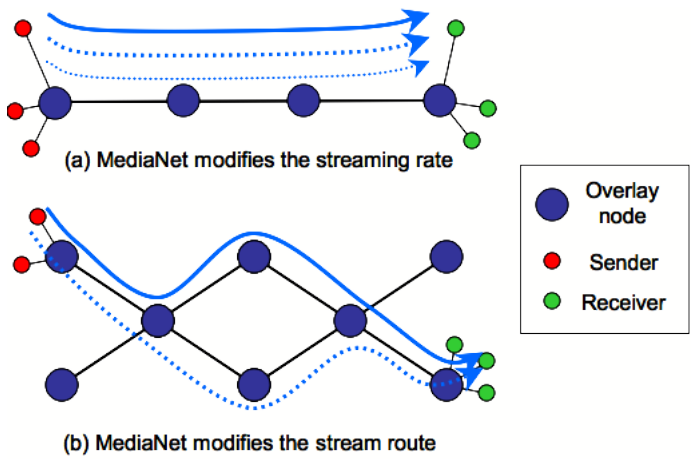


Figure 6.1: The basic operation of MediaNet: MediaNet forwards media streams from senders to receivers. When it detects that the network path cannot handle the current rate of its streams, it adapts by (a) changing the streaming rate between HIGH, MEDIUM and LOW, and (b) switching the routes that streams take, if alternate routes are available.

of each media stream are switched between overlay links (or equivalently, between TCP sessions). For example, in Figure 6.1(a) we see a *line configuration* where MediaNet has three overlay links with only one possible route between sender and receiver. If network conditions deteriorate and the rate of a stream cannot be sustained, then the only option to MediaNet is to downgrade the stream rate. In Figure 6.1(b) we see a *diamond configuration* with 8 overlay links and two possible routes.

MediaNet needs to know the available bandwidth on all the links of the overlay, whether they are used or not. Currently, MediaNet uses passive measurement to get approximate estimates of the available bandwidth on each link that it sends traffic, but gets no estimates for unused links. Each overlay node keeps track of the TCP rate it can sustain on each overlay link. If no loss occurs, this is the lower bound of the available bandwidth. MediaNet knows that it can send up to that rate on that link, but does not know if it can send more. If an application-level loss occurs (the LS will buffer packets it cannot send at the desired rate, and then drop them selectively when the buffer becomes full) this implies

the current TCP rate cannot be sustained, so MediaNet needs to reconfigure that node to send at a lower rate.

By using passive measurement, MediaNet can react to losses which indicate reduced bandwidth but it cannot easily reclaim that bandwidth once it becomes available again or switch to inactive paths that have more bandwidth than the existing path. In its original implementation, the GS treats the current, average sustained bandwidth along a virtual link as a lower bound, and then it slowly "creeps" up, at a configurable rate, its view of the upper bound of the available bandwidth. Eventually this upper bound will reach a level that the GS will attempt to reconfigure the delivery schedule to try the link again. If its guess was wrong then packets will be dropped, causing the GS's estimate of the available bandwidth to be readjusted and its schedule to be reconfigured. If the creep rate is too high, the packet drops and reconfigurations can be disruptive to the user experience. If the creep rate is too low, users will receive lower utility levels for longer than necessary.

Note that Skype [17] faces a similar problem at the beginning of each voice/video session: without any estimate about the bandwidth that the path can sustain, it cannot determine what quality to start with. To avoid starting a call with quality that is too low, Skype resorts to a simple duplication scheme where it sends each packet twice until it can measure the conditions on the path and make an informed decision of the best stream rate [13] to use.

6.1.1 Active Measurement with MGRP

It is clear that passive measurement techniques are not enough for MediaNet to fulfill its goal. What MediaNet needs is to actively probe the network and get periodic estimates of the actual bandwidth on all overlay links. One way to do this is for MediaNet to use custom measurement techniques, like VDN [16], and reuse its own traffic to measure the paths. But as we discussed in section 1.1, this has the main drawback that the estimation algorithm has to be tightly integrated with the packet delivery implementation. This makes such an approach difficult to reuse and difficult to change.

Active measurement with MGRP alleviates the problems associated with coupling estimation algorithms with application implementations, and gives you the best of both worlds: the available bandwidth algorithm is written and run as a different process, which seamlessly makes use of MediaNet traffic if it is available. Since MediaNet forwards traffic using TCP and continuously sends application traffic on many of the links that it needs to measure, this makes it a perfect client of the Measurement Manager and MGRP. We use pathload to add active measurements of the available bandwidth to MediaNet.

Given the current Measurement Manager architecture, we still need to manage the multiple instances of pathload that need to run between MediaNet nodes. In MediaNet the local schedulers are responsible for passively measuring the overlay links adjacent to their nodes and reporting the results to the global scheduler. This makes them natural candidates for running the pathload processes continually, collecting the results every time a pathload session ends and reporting them back to the GS. However, this approach still does not address our modularity concerns since we need to change the local schedulers

to use a *specific* active measurement tool. It is still difficult to reuse (e.g., the mechanism for running pathload and collecting results) and requires additional work if we decided to use another tool.

The modularity of the Measurement Manager architecture and the fact that MGRP works with any application and any active measurement tool creates a better opportunity which will allow MediaNet to start using active measurement without any changes to the local schedulers. This achieved by employing the notion of an *Estimation Overlay*.

6.2 Estimation Overlay

The *Estimation Overlay* is a separate measurement overlay service that performs active measurements between pairs of nodes using MGRP. The Estimator Overlay currently measures, upon request, the available bandwidth between any two of its overlay nodes. By setting this network up to mirror the structure of the MediaNet overlay, the measurements of the Estimator Overlay will take advantage of existing MediaNet traffic automatically, when it is available. MediaNet's GS can then query this measurement overlay to acquire per-link statistics for every link it needs to monitor; the local overlay nodes do not have to be changed at all. These per-link measurements are in addition to any reports the GS receives from the LSes about their own traffic. The combination of the two provides MediaNet with a fuller picture of network conditions on its paths. It can react quickly to reports of loss from the LSes and then ramp back up the streaming rates when the estimation overlay reports high available bandwidth.

The Estimation Overlay provides measurement as a service. When applications

such as MediaNet need a particular measurement, they query the overlay which in turn measures the network and responds with an estimate. This separation between *what to measure* and *how to measure* allows the estimator overlay to use a variety of estimation algorithms and pick the best one suited for the task at hand. A typical trade-off is the one between measurement accuracy and overhead. For example MediaNet needs estimates for all of its links; both active and inactive. For links where MediaNet sends traffic, MGRP piggybacking to sharply reduce the probe overhead, and the estimator can afford to use a high-overhead algorithm, if it provides quick and accurate results. But for links that have no traffic, the estimator may choose a less accurate algorithm that uses fewer probes. The Estimation Overlay currently implements the pathload algorithm. A plan for future work is to extend the estimator to combine together a number of existing tools (Chapter 8).

6.3 MediaNet Experiments

We conducted a series of experiments to demonstrate how MediaNet operates and how it can benefit from integration with MGRP.

6.3.1 Experimental Setup

Network Topology We use the network configuration shown in Figure 6.2. One local scheduler runs on each of the forwarder nodes (m3, m5) and the global scheduler runs on a separate node. In our enhanced version the Estimation Overlay also runs between m3 and m5, and the GS queries it for available bandwidth between those nodes. Since there is but one path between m3 and m5, if network conditions determine the GS can only

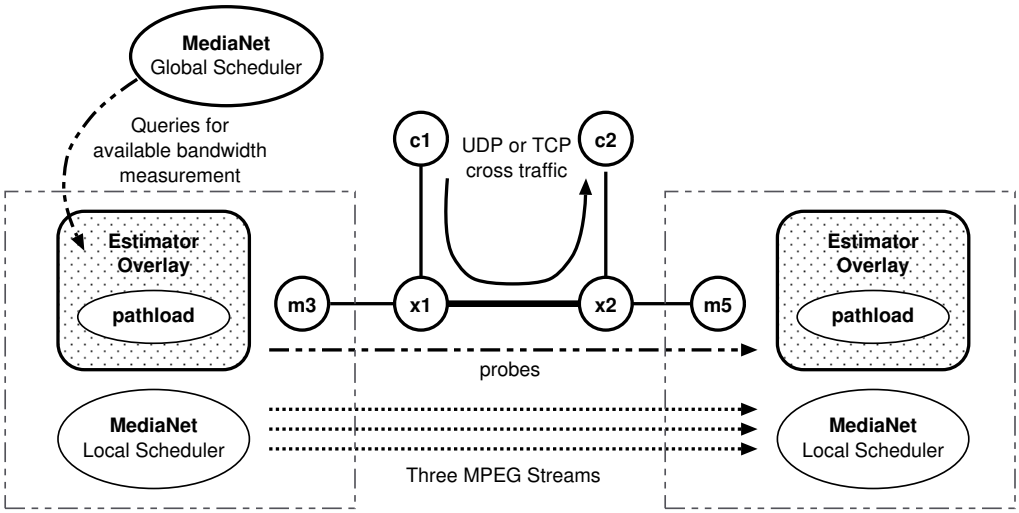


Figure 6.2: MediaNet experiment: The local schedulers schedule three MPEG streams between m3 and m5. The global scheduler asks the estimator overlay for periodic estimates of the available bandwidth between m3 and m5. The MediaNet streams share the bottleneck link (x1x2) with cross traffic from c1 to c2.

control the sending rate of source traffic by applying adaptations from user specifications, described next.

Source traffic For the source traffic, we loop three MPEG video streams. The I, P and B frames in the MPEG stream are distributed according to Table 6.1. For each stream we provide a specification to the MediaNet GS that prefers the full stream at utility 1.0, dropped B frames at utility 0.5, and dropped P and B frames at utility 0.3. Each video stream requires about 145 KB/s to send at its full rate, about 88 KB/s to send only I and P frames, and about 27 KB/s to send only I frames. Since the experimental network permits only one path between the source and destination, all adaptation will take place by dropping frames, rather than rerouting. In MediaNet, frames are dropped as early as possible (in this case at the sender), which reduces the bandwidth sent across the network overall.

Frame Type	Average Size (B)	Frequency (Hz)
I	13500	2
P	7625	8
B	2850	20

Table 6.1: MPEG frame distribution: full rate sends all three frame types, medium rate drops the B frames and sends only the I and P frames, and the lowest rate uses only the I frames.

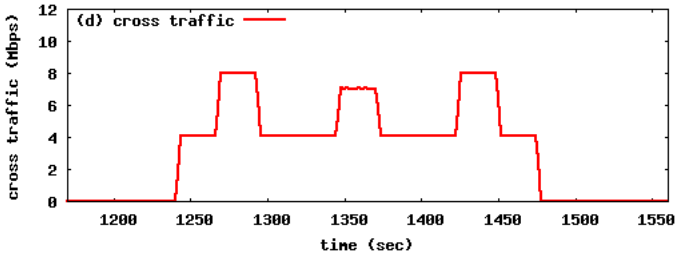


Figure 6.3: Cross traffic for the MediaNet experiment: UDP cross traffic that uses fixed-rate intervals of variable duration. We use the following rates: 4, 8, 4, 7, 4, 8, 4 Mbps.

Cross traffic The MediaNet streams share the bottleneck link (x1x2) with cross traffic from c1 to c2. We generate UDP cross traffic following the STEP pattern (also used for experiments in Section 5.1), which adjusts the bandwidth in a stepwise fashion for 240 seconds using fixed-rate variable-duration intervals (4, 8, 4, 7, 4, 8, 4 Mbps). Figure 6.3 shows the cross traffic that we use.

6.3.2 Original MediaNet

Our baseline experiment considers the original MediaNet implementation, which uses only passive measurement to adapt the media streams. Figure 6.4 shows how MediaNet adapts its streams in the presence of STEP traffic. All the plots have time on the x-axis and depict 400 seconds of an experiment run that we have found to be representative of the many runs we considered. We now proceed to explain the plots in the figure.

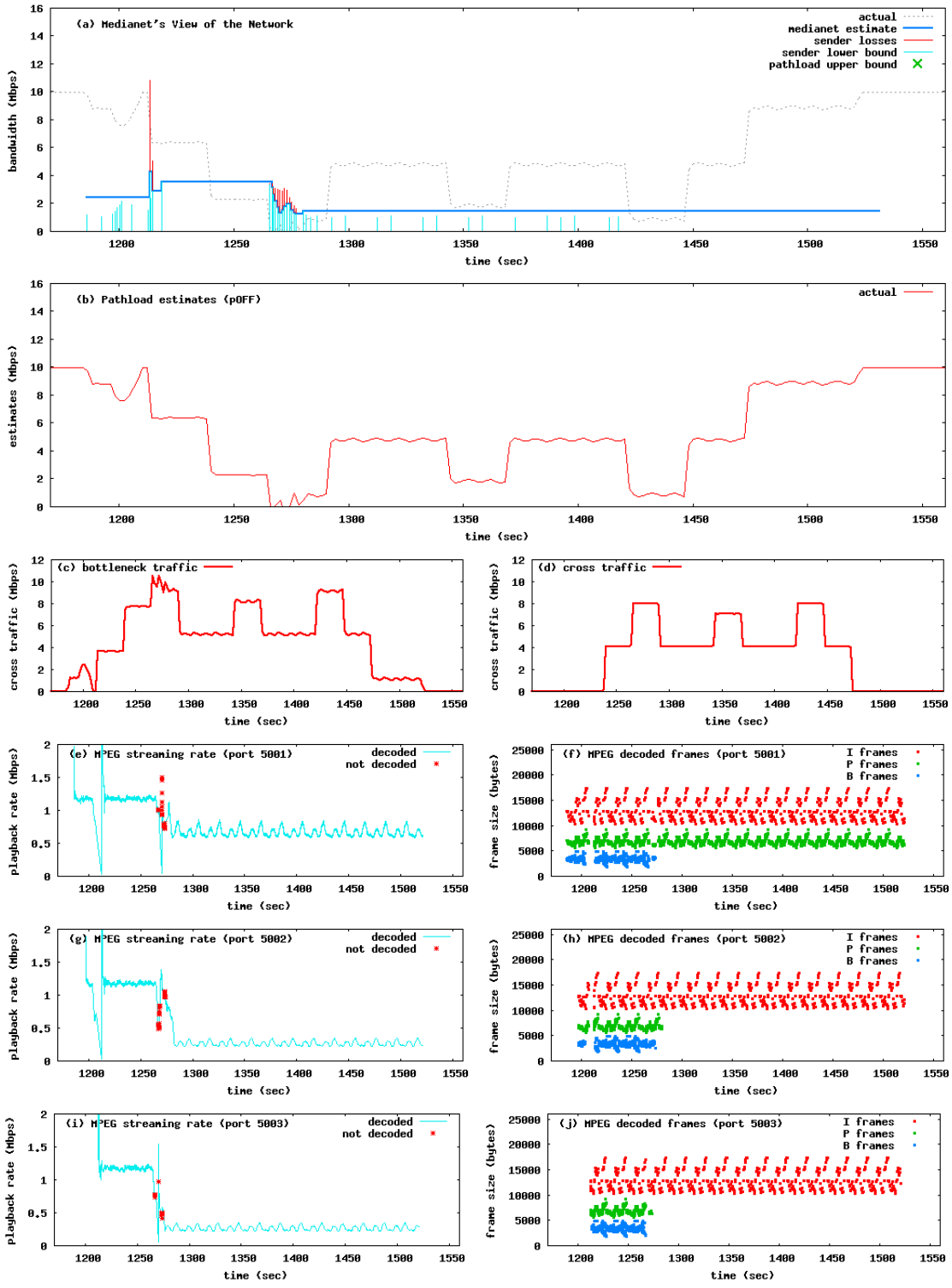


Figure 6.4: Original MediaNet Experiment: We see that all the MPEG sessions start at high streaming rate (e), (g) and (i) but around time 1270 MediaNet downgrades all streams due to high loss and no available bandwidth since cross traffic (d) spikes. From the MPEG frame plot (f) we see that the first stream is downgraded to medium quality (loses only the B frames) but the other two streams (h) and (j) lose both P and B frames and are downgraded to low quality.

Cross Traffic Plots We already saw the cross traffic plot (d) that shows the UDP cross traffic that is sent from c1, the source of the cross traffic. We can see clearly how the UDP traffic is sent in fixed-rate intervals. The data for the traffic plots is extracted from the Ethernet counters of the network devices and represent actual traffic sent. Plot (c) uses the Ethernet counters on x1 (the ingress router to the bottleneck link) to plot the actual traffic sent on the bottleneck link. The traffic on this plot combines the cross traffic, and the traffic of the MPEG streams.

MPEG Stream Plots These plots are the bottom six plots in Figure 6.4, subplots (e) through (j). Each horizontal pair represents one MPEG stream and we will concentrate on the first stream, subplots (e) and (f). Plot (e), the one on the left, shows the rate that frames of the MPEG stream arrive at the receiver. This rate is not necessarily the playback rate. For example, due to TCP slow start effects, the incoming MPEG rate dips for a few seconds around time 1210 but does not lose any frames. So the stream can be decoded correctly but requires the use of playback buffers, which are outside the scope of our experiment.

Plot (f), the one on the right, shows the types of MPEG frames that are decoded at any point of time. The top red dots represent I frames, the middle green dots are P frames and the bottom blue dots represent B frames. Since the more frame types we have, the better the quality of the decoded stream, it is desirable to have a plot that is as full as possible with all three types of frames. For example, at the beginning of the session we start with high quality but then we drop to medium quality after time 1275. The gap across all three types at time 1210 is due to the TCP rate dip that we discussed earlier; no

frames were lost, but they were delayed (handled by the playback buffer).

Medianet Plots Plot (a) in Figure 6.4 shows MediaNet's view of the available bandwidth on the network path between m3 and m5. At any point of time, the long horizontal blue line represents the amount of bandwidth that the MediaNet GS thinks is available. During reconfigurations, MediaNet ensures that the media streaming rates are adapted so that when combined together they do not go above that known available bandwidth. The dotted line indicates the actual available bandwidth, which is computed by subtracting the bottleneck traffic (plot (c)) from the capacity of the x1x2 link (10 Mbps). This line is shown in a more pronounced way in plot (b).

MediaNet computes a new value for the available bandwidth every time it receives a report. Since this is the original MediaNet, we do not get any estimates from active measurement tools and MediaNet has to rely solely on reports from the LSes. We see these reports in the form of vertical impulses. A light blue impulse represents the rate that the local scheduler could send successfully (representing a lower bound). A red impulse on top of the blue impulse indicates that the LS tried to send traffic at a higher rate than it could (representing an upper bound). The light blue "could send" impulses push MediaNet's estimate up, while the red "tried but failed to send" impulses push the estimate down.

6.3.2.1 How Medianet Adaptation Works

Using the example run in Figure 6.4 we can see how MediaNet adapts MPEG streams as a result of loss. Around time 1200, before the cross traffic starts, all MPEG

sessions start using the highest streaming rate. The local schedulers succeed in sending the desired rate and send reports to MediaNet, which result in the vertical light blue impulses in (a). MediaNet starts forming an estimate for the available bandwidth (blue horizontal line in (a)). Around time 1270 the second step of the UDP cross traffic kicks in and overwhelms the bottleneck link. The local scheduler encounters loss which we see as red dots in (e), (g), and (h). The LS sends a report to the GS; these appear in (a) as the red impulses. As a result MediaNet pushes down its estimate of the available bandwidth and initiates a reconfiguration. MediaNet then instructs the LS to downgrade the first MPEG stream to medium level, and the other two to low level. This is reflected in the change of streaming rate in (e), (g) and (i), and from the types of frames that disappear from plots (f), (h) and (j).

From this example, we see the main drawback of the original operation of MediaNet: it correctly downgrades the streams to adapt to reduced bandwidth but fails to upgrade the streams when bandwidth becomes available again. This is apparent in (a) where MediaNet's view of the available bandwidth does not change after time 1270 even though the dotted line shows that bandwidth becomes again available at time 1300, 1370 and 1450.

6.3.3 MediaNet with Active Measurement

In this set of experiments we demonstrate how the performance of MediaNet improves when we supplement the reports that the global scheduler receives with active network measurements from the estimator overlay, as we described in section 6.1.1. The

main difference with the original MediaNet (Section 6.3.2) is that now the estimator overlay runs pathload between m3 and m5 and sends periodic reports to the estimator. We will present two sets of experiments. One where the estimator overlay uses the original version of pathload (pSLOW), and one with the more aggressive version (pFAST) (Section 5.1.3).

6.3.3.1 Summary of Results

In the following sections (6.3.3.2, 6.3.3.3 and 6.3.3.4) we present plots of representative experiment runs that show that we can improve the quality of service of the original MediaNet (section 6.3.2) by (1) incorporating active measurements using pathload, and (2) using MGRP. We have repeated our experiments multiple times and the MGRP benefit remains consistent across runs.

We will use three metrics to assess the quality of the service provided by MediaNet. The first metric is the *average number of decoded frames per second (fps)*. The higher the frame rate, the better the quality of the MPEG stream. The second metric is the *average streaming rate (mbps)*, which is related to the frames per second, with higher rates more desirable. Even if we increase the quality of the MPEG stream on average, we need to make sure that playback is smooth and not full of glitches or freezes. To determine this we use the third metric, the *ratio of frames that failed to decode*. These failed frames are usually a result of MediaNet reconfigurations and we need to keep their number and ratio as low as possible. It is conceivable that we can improve on average the MPEG streaming rate and the number of frames we decode per second, but at the same time have a high ratio of failed frames.

experiment	runs	sec	fps	% over non-MGRP	% over original
mgrpOFF.pOFF	14	337	30.11		
mgrpOFF.pSLOW	22	336	39.58		31.44%
mgrp10.pSLOW	32	336	43.42	**9.69%**	44.19%
mgrpOFF.pFAST	10	335	39.10		29.87%
mgrp10.pFAST	22	336	52.08	**33.19%**	72.96%

Table 6.2: Increase of decoded MPEG frames/second with MGRP: The number of successfully decoded MPEG frames appears in column 4 as a ratio of frames per second (fps). We see that using active measurement improves MediaNet in both cases (column 6). But with MGRP, MediaNet can adapt the streaming rates better so that the number of MPEG frames it sends and decodes per second (column 5) increases by 9-30%.

We collected results from multiple runs of all MediaNet modes of operation and we present our results for the three quality metrics. We will show that MGRP improves MediaNet's performance across all metrics: it increases both the frame and streaming rates, and either keeps the same or lowers the ratio of failed frames.

Table 6.2 shows the average number of decoded frames per second. Each row represents a different MediaNet mode of operation. We collected results from multiple runs with the same duration. Column 2 shows the number of runs and column 3 shows the average duration of each run. For each row, we added all the frames that were successfully decoded across runs, and divided by the total number of seconds. This produced the frames-per-second (fps) metric on column 4. We then computed percentages to quantify how much the frame rate increased in MGRP, compared to: (a) the active measurement case, but without MGRP, (b) the original case. As we can see from the results (Table 6.2), it is very clear that using active measurement with MediaNet is a clear win, since we get a 31.44% increase (column 6) even without MGRP. With MGRP, we get an even further increase (column 5) where frame rate increases by 9.69% for pSLOW and 33.19% for

experiment	runs	sec	mbps	% over non-MGRP	% over original
mgrpOFF.pOFF	14	337	1.84		
mgrpOFF.pSLOW	22	336	1.96		6.29%
mgrp10.pSLOW	32	336	2.05	**4.40%**	11.21%
mgrpOFF.pFAST	10	335	1.86		0.94%
mgrp10.pFAST	22	336	2.28	**22.52%**	23.86%

Table 6.3: Increase of MPEG streaming rate with MGRP: The MPEG streaming rate appears in column 4 as Megabits per Second (mbps). MGRP provides a clear improvement over the non-MGRP case in both cases of active measurement (pSLOW and pFAST) where it increases the streaming rate by 4-22%. The improvement on streaming rate is less than the improvement on the number of frames decoded (Table 6.2) since MediaNet uses three different frame sizes, and most of the increase in frames was due to small-sized frames.

pFAST, compared with the non-MGRP case.

Table 6.3 shows the average streaming rate. For each row we added all the sizes of frames that were decoded correctly and divided them by the total number of seconds. This produced the average MPEG streaming rate (mbps). With MGRP we get an increase of 4.40% for pSLOW and 22.52% for pFAST, compared to the non-MGRP case. The difference between the percentage increase of the streaming rate metric (mbps) and the increase of the frame rate is expected, since we have three different kinds of frames with different sizes. With MGRP, the number of the much smaller frames increase, which explains how the large increase in frame rate, translates to a more moderate increase of streaming rate.

Table 6.4 shows the ratio of the frames that failed to decode. These failed frames are usually the result of frame losses during MediaNet reconfigurations and result in glitches or freezes during playback. It is important to keep this ratio low, but there is a trade-off between using reconfigurations to change back to higher streaming rates, and disrupting

experiment	runs	sec	bad frames	% over non-MGRP	% over original
mgrpOFF.pOFF	14	337	0.0009		
mgrpOFF.pSLOW	22	336	0.0080		785.70%
mgrp10.pSLOW	32	336	0.0079	**-0.76%**	782.09%
mgrpOFF.pFAST	10	335	0.0114		1162.93%
mgrp10.pFAST	22	336	0.0100	**-12.07%**	1013.73%

Table 6.4: Occurence of decoding failures in MediaNET with MGRP: The ratio of the frames that failed to be decoded over the total frames appears in column 4. We see that MGRP (column 5) lowers the failure ratio slightly (less than 1%) for pSLOW, and more substantially for pFAST (around 12%). This is a good result because it shows that the gains of MGRP (in frame and stream rate) are not at the expense of the frame decoding process.

the video playback by introducing failed frames as a result of reconfigurations. The results in tables 6.2 and 6.3 shows that MGRP improves the operation of MediaNet. Our goal is to show that these MGRP gains (which are average gains) do not increase the occurence of failed frames (which momentarily affect quality).

From Table 6.4 we see that MGRP not only does not increase the failed frames, but also reduces them. Column 4 shows the ratio of frames that failed to be decoded over the total number of frames. In column 5 we see the percentage that MGRP increases that ratio, compared to the non-MGRP case. For both pSLOW and pFAST the percentages are negative. In pSLOW, MGRP achieves a 0.76% reduction, and in pFAST a 12% reduction. In any case, the ratio of bad frames is quite low. For pSLOW, about 0.8% of frames fail to decode, and for pFAST about 1% of frames fail to decode (with or without MGRP).

The very large increase in the failure rate of active measurement compared to the original MediaNet (column 6), is expected, and is due to the baseline ratio being too close to zero. In the original mode, MediaNet hardly makes any reconfigurations so the number of failed frames is negligible. We can see from the table that only 0.09% of frames fail to

decode in the original case. The failed frames is the price for using active measurement and reconfigurations to maximize the MPEG streaming rates. This table shows that the price for that approach is well worth it.

The results in tables 6.2, 6.3 and 6.4 make clear that MGRP improves the operation of MediaNet, by increasing the MPEG streaming rates without increasing the frames that cannot be decoded. So the experiment runs that we describe in detail in the following sections are representative of the improvements that MediaNet gets by using MGRP.

6.3.3.2 Using Original Pathload (pSLOW)

Figure 6.5 shows how MediaNet operates with the addition of the new pathload estimates. Plots (a) and (b) now show the pathload estimates as green X points. In plot (a) we see how the pathload estimates refine MediaNet's knowledge about the available bandwidth. MediaNet's estimate (horizontal blue line) now follows closer the actual bandwidth (dotted line). Compare this to the original MediaNet (Figure 6.4(a)). In plot (b), we show one point for every available bandwidth estimate that MediaNet receives from the active measurement overlay, which can be compared to the actual available bandwidth to visually asses the accuracy of the estimates.

The benefit to the MGRP streams is evident. In plots (e), (g) and (i), we see how all three streams are upgraded to a higher streaming rate around time 1350, after the downgrade at time 1270. This upgrade was not possible in the original case, and is due to MediaNet now receiving a pathload estimate. As we see in plot (a), the estimate at time 1350 prompts MediaNet to raise its bandwidth ceiling from about 2 Mbps to about

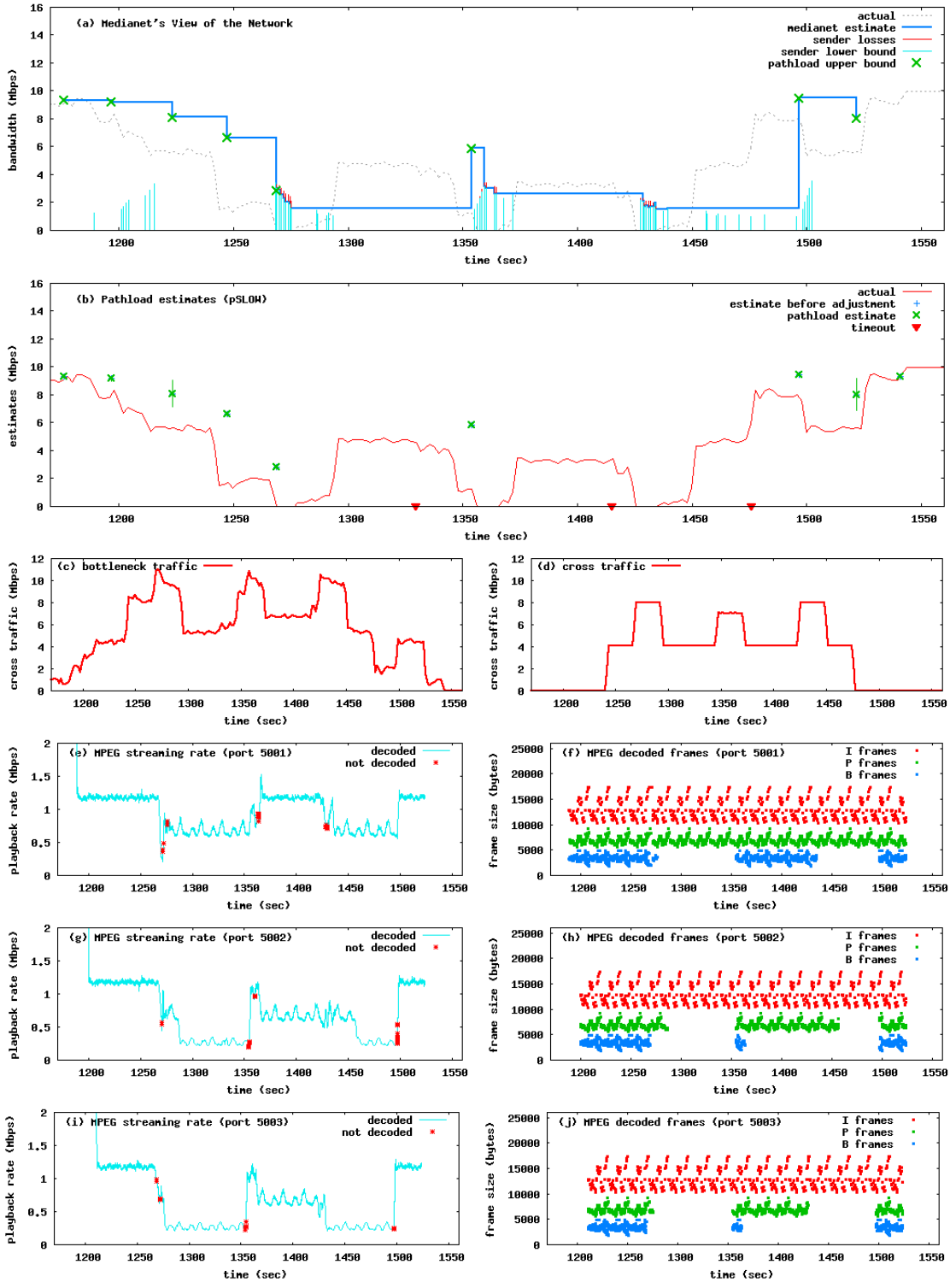


Figure 6.5: Medianet with Active Measurement but without MGRP (mgrpOFF/pSLOW)

6 Mbps. This triggers a reconfiguration since all the MPEG streams are using rates that are lower than the maximum quality rate. After the reconfiguration, MediaNet instructs the local scheduler to raise the rate of all of the MPEG streams to the highest level, since 6 Mbps appears to MediaNet to be more than enough to accommodate the three MPEG streams of 1.16 Mbps each.

However, pathload overestimated the available bandwidth at time 1350. Compare in plot (b) the difference between the pathload estimate (green dot) and the actual bandwidth (red line). This leads MediaNet to raise the available bandwidth ceiling too high. Once the local scheduler attempts to send at the higher rate, it incurs loss and reports back to the GS. This prompts another reconfiguration and finally the stream rates stabilize at HIGH, MEDIUM and MEDIUM for the time period [1370,1420]. We can see from plots (h) and (j) that streams 2 and 3 briefly used the higher rate (all frames are present).

This experiment demonstrates that the interplay between active and passive measurement is useful in guiding MediaNet to adapt the stream rates better than the original case. Without the active measurements of pathload, MediaNet would have never upgraded to the higher rates when bandwidth became available again, and without the passive measurements of the local scheduler, MediaNet would not have adjusted the stream rates when the estimate is too high.

It is also clear however, that the accuracy and timeliness of the estimates provided by the estimator overlay have a direct effect on the operation of MediaNet. In this experiment, pathload demonstrates some shortcomings: (1) it tends to overestimate the available bandwidth, and (2) estimates take on the order of 20 seconds but can be much longer (we have set a timeout of 60 seconds). These are limitations of pathload that are especially

apparent during periods of lower actual bandwidth (a fact we also showed in our experiments in Chapter 5).

6.3.3.3 Using Original Pathload (pSLOW) with MGRP

Figure 6.6 shows the same MediaNet session we presented in the previous section but now going over MGRP. Plot (b), in addition to the pathload estimates (green x), also shows the pathload estimates prior to being adjusted for piggybacking (as we discussed throughout section 5.2), shown as a blue cross. Otherwise the plots are the same we presented in the previous section.

We notice in plot (a) that pathload over MGRP provides estimates more often than the non-MGRP case and with similar accuracy. This agrees with our results in Chapter 5. The higher frequency of pathload estimates improves the adaptation of MediaNet so that the the three streams, taken together, achieve higher streaming rates in the MGRP case. The plots (e, g, i) in Figure 6.6 show that most of the time in the MGRP case all three streams are using the MEDIUM rate, with brief excursions to the HIGH and LOW rates. In the non-MGRP case (Figure 6.5), the first stream performs better but the other two spend half of the time in the LOW rate. Also, towards the end of the experiment, around time 1500, in the MGRP case MediaNet reverts sooner to the HIGH rate than the non-MGRP case, because pathload returns an estimate sooner.

However, MGRP does not eliminate the limitations of pathload. Pathload still times out twice for 100 seconds during the time period [1375,1475] where the available bandwidth bounces back from zero. This happened exactly in the same way for the non-MGRP

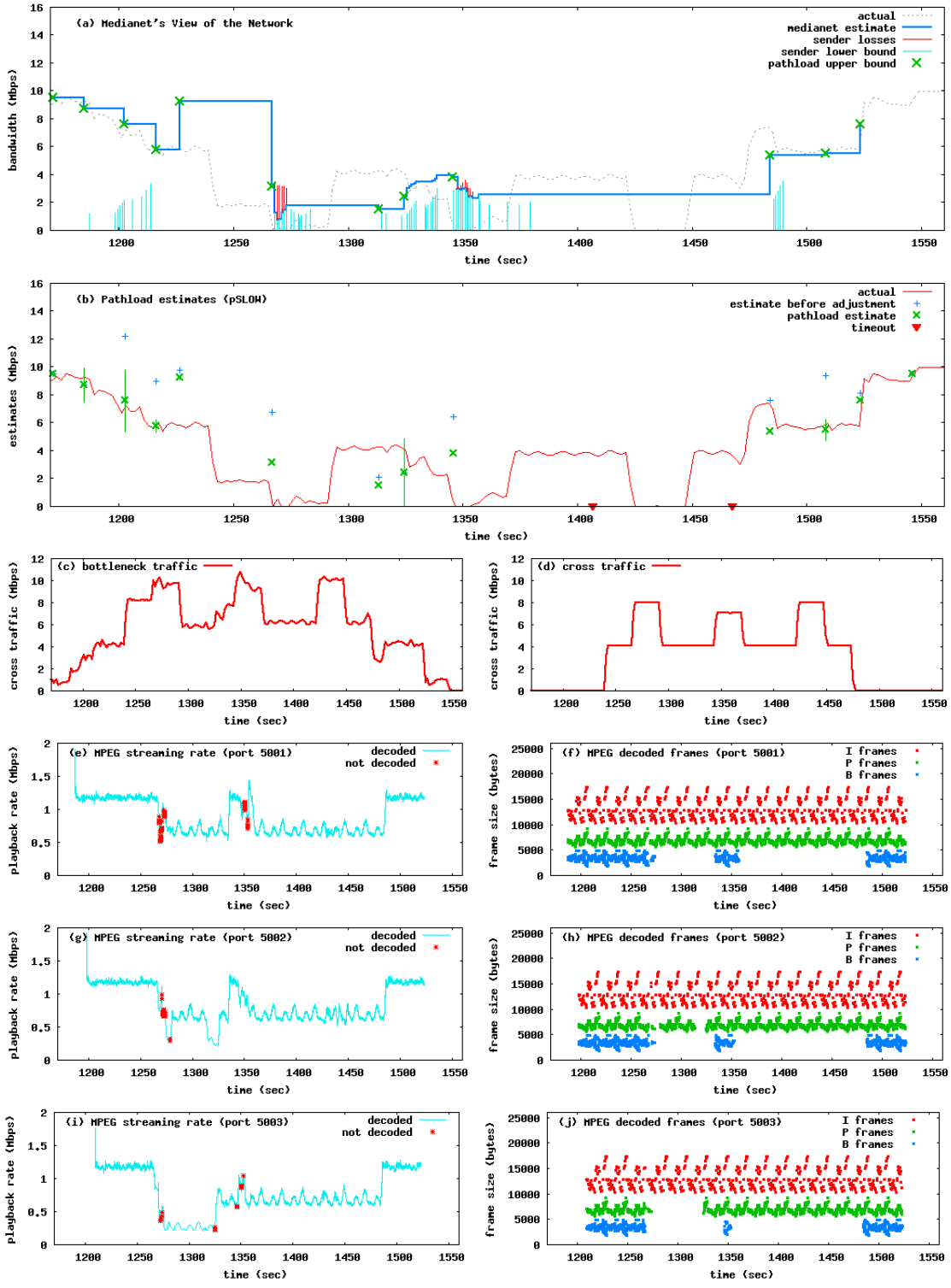


Figure 6.6: MediaNet experiment with three MPEG streams using mgrp10/pSLOW

case. By not producing an estimate during that period, MediaNet misses in both cases the opportunity to bump up the MPEG streaming rates.

6.3.3.4 Using Aggressive Pathload (pFAST)

In Chapter 5 we showed that with MGRP we can run a more aggressive version of pathload (pFAST as compared to pSLOW) so that we can shorten the time it takes to produce an estimate. We have repeated the above experiment for both the non-MGRP case (Figure 6.7) and with MGRP turned on (Figure 6.8).

We see from Figure 6.8(b) that Medianet with MGRP receives estimates from pathload more often. Plot (a) shows that MediaNet follows the actual available bandwidth more closely and in a more timely manner. This enables MediaNet to take advantage of periods of increased bandwidth that are between periods of low bandwidth, and bump up the streaming rates to HIGH (plots (e) through (j)). There are two such 50 second periods in our experiment: [1300,1350] and [1375,1425]. For the same time periods in the non-MGRP case (Figure 6.7) the streaming rates remain at MEDIUM. On the other hand, pathload timeouts occur with MGRP too, which is the reason why MediaNet did not upgrade the streams earlier in the period [1300,1350]. But it still outperforms the non-MGRP case.

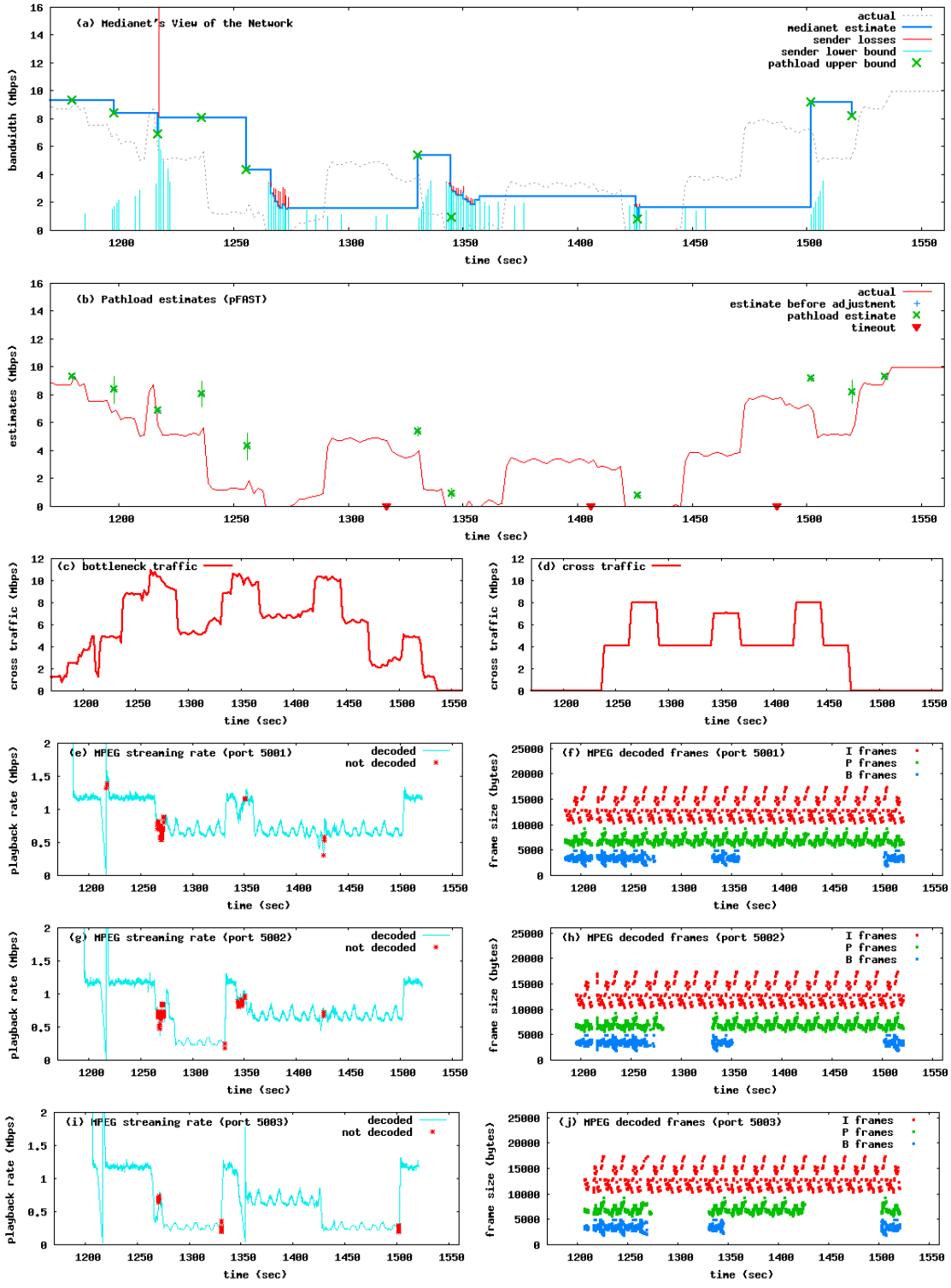


Figure 6.7: MediaNet experiment with three MPEG streams using mgrpOFF/pFAST

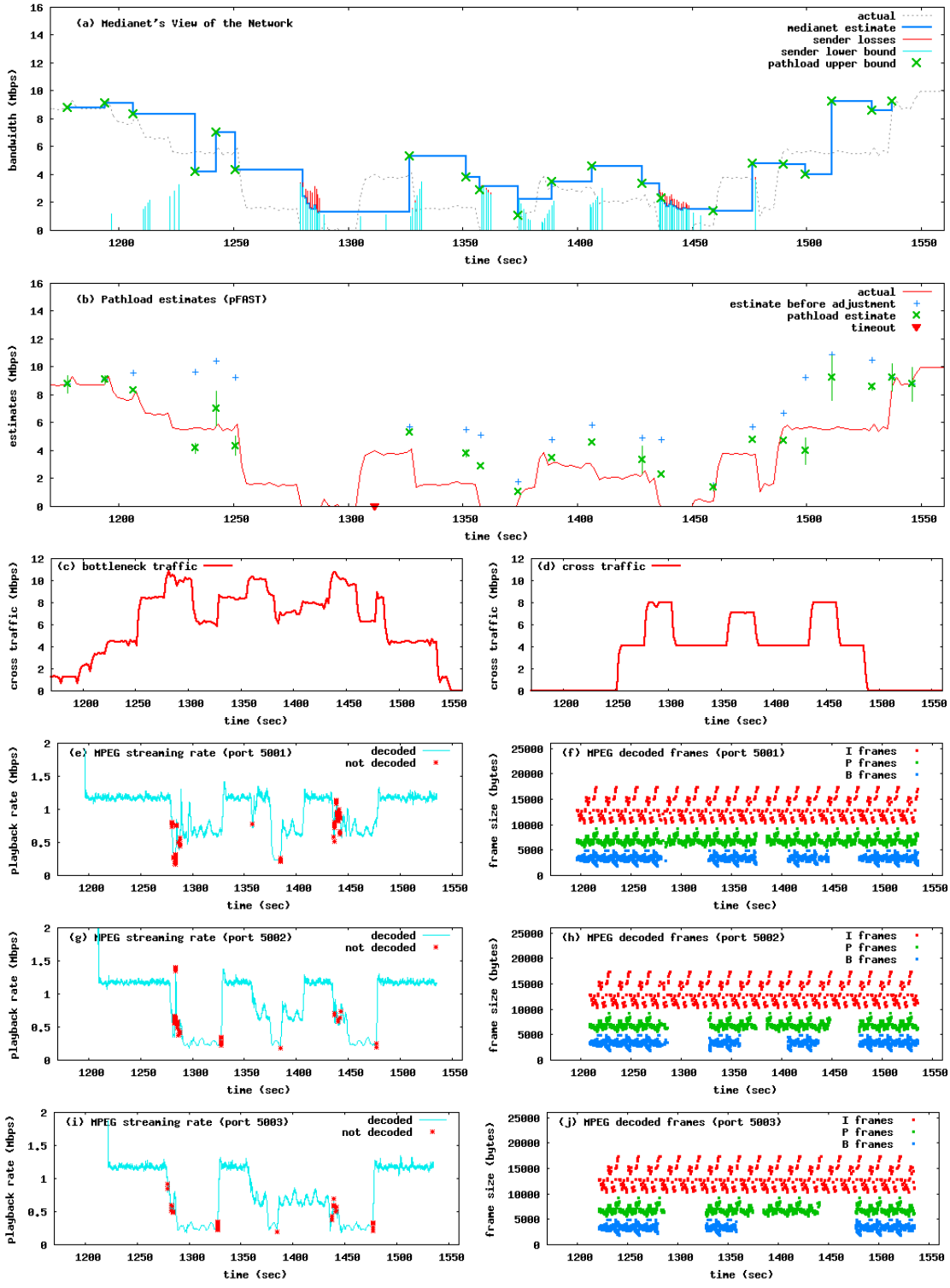


Figure 6.8: MediaNet experiment with three MPEG streams using mgrp10/pFAST

6.4 MediaNet lessons for the Estimation Overlay

In this chapter we integrated MediaNet with the Measurement Manager. Instead of using pathload directly, we introduced the notion of the Estimation Overlay (section 6.2). While experimenting with MediaNet we found the estimator overlay to be a valuable abstraction that concealed the complexities of running active measurement tools and simplified the process of collecting results.

However, because of limitations of pathload, MediaNet cannot achieve optimal adaptation. The time it takes for pathload to produce an estimate is too long. Sometimes the estimates are too high which leads MediaNet to upgrade rates when no bandwidth is available. Also the tendency of pathload to time out during transitions between low and high bandwidth prevents MediaNet from upgrading the streaming rates early. MediaNet would settle for a less accurate measurement, as long as it is reported quickly and predictably. In the simplest case, it is worth more to MediaNet to get just an *indication* that the available bandwidth has increased but to get it very soon after the change, rather than get a very accurate numerical estimate that is delayed.

In short, we can imagine improving the estimation overlay in the following ways:

1. Implement many algorithms for each network property, and use the simplest and fastest that fits the application requirements.

2. Know the limitations of each algorithm. Applications may find them unacceptable and may want to use a different algorithm. Or, there may be certain environments that some algorithms perform better than others.

3. Provide some sort of quality metric and error margin for the estimates provided.

We discuss more ideas about the estimation overlay in Chapter 8.

Chapter 7

Related Work

The need for end-to-end measurement has generated a large body of research. In this section we present a sample of the work most related to the Measurement Manager. Our work is differentiated by the body of literature in the following ways:

- The Measurement Manager is a new architecture for end-to-end network measurement that is integrated at the Layer 4 of the IP protocol stack. Probing and measurement are separate components of the architecture and each can evolve independently.

- The Measurement Manager provides generic inline measurements. Transport payload is reused on-demand to fill empty probes; if no payload is available probes are sent out empty. To our knowledge, the Measurement Manager is the first kernel service to attempt to fill empty probe padding with useful payload, to reduce measurement overhead.

- The Measurement Manager architecture was designed from the start to piggyback transport payload in probes. Most existing approaches to inline measurement instead employ trickery to embed their probes into TCP streams.

- The Measurement Manger is independent of applications, transport protocols and measurement algorithms, providing a small set of probing primitives for building

efficient estimation algorithms.

7.1 Network measurement tools

Network measurement tools infer network characteristics by observing the effects of injected or normally-sent packets. We used existing tools to shape the Probing API primitives. The number of available network measurement tools is quite large [61, 62]. We list here a sample of the tools that we studied in detail while designing the Measurement Manager architecture; they are described in more detail in Chapter 2.

- **Pathchar [22, 63]** uses variable packet size probing to estimate path characteristics.
- **Pathload [6]** uses packet trains with uniform spacing to estimate available bandwidth and has been covered in detail in Section 2.2.1.
- **Pathchirp [15]** uses packet trains with variable packet spacing to estimate available bandwidth. It uses fewer probes than pathload but is not as accurate [58] (also verified by our experiments).
- **Pathrate [7]** uses packet pair probing to estimate path capacity.
- **Pathneck [25]** uses recursive packet trains (TTL-limited probes surrounded by normal packets) to locate bottleneck links along a network path.

MGRP does not provide support for TTL-limited packets, or packets sent over ICMP, currently precluding us from implementing Pathneck and some other tools. We believe we could add this missing support without difficulty.

7.2 End-to-end Measurement Services

There are a number of systems that export end-to-end measurement services to endhosts and use a similar API to ours. The Measurement Manager extends and complements those systems by providing generic inline probing and by using transport payload on-demand to mitigate the intrusive nature of active probing without sacrificing measurement flexibility.

- **MAD [26]** is a system for Multi-user Active Measurement that provides measurement as a system service in the Linux kernel. Like the Measurement Manager, MAD generates probes on behalf of probers, but does not do any piggybacking and has a more restrictive Probe API. Probers specify probes with an assembly-like language, which schedules probes according to a discrete clock. MAD aims at being accurate and provides an interface for system self-measurement, so that probe tools can measure the error in the probing process.

- **Periscope [27]** is a programmable probing framework implemented in the Linux kernel. Periscope follows the active probing approach and exports an API that active measurement tools can use to define arbitrary probing structures (such as packet pairs and packet trains) which are then sent by the system. Periscope sends the probes as `ICMP ECHO REQUESTS` and can collect remote timestamps using the `ICMP` response packets. Like the Measurement Manager, Periscope generates the probes inside the kernel for greater accuracy.

- **Scriptroute [28]** is a system that allows ordinary Internet users to conduct network

measurements from multiple vantage points. Users submit measurement scripts in which they construct their probe traffic (using a *Send-train API*) and the Scriptroute server generates the probes using raw sockets. The primary goal of the system is to perform measurements on behalf of unauthenticated users in a flexible but secure manner. The Measurement Manager operates at a layer lower than Scriptroute and can complement its operation (for example by integrating them at the last stage of the Scriptroute's *Network Guardian* module where the system sends out the probes).

- Pásztor and Veitch **[29]** present a precision infrastructure for active probing that uses Real-Time Linux to send probe streams with high accuracy. Like the Measurement Manager, this system has two separate components, one for measurement and one for probing. The focus of this work is to send the probes as accurately as possible with commodity equipment and software.

- **pktd [30]** is a packet capture and injection agent that performs passive and active measurement on behalf of client measurement tools. The focus of this work is to provide controlled and fine-grained access to the resources of the network device. As in the case of Scriptroute, pktd operates above the Measurement Manager, which can be integrated with pktd's *Injection Manager* to send out precise probe patterns with payload piggybacking.

- **NIMI [64]** is a large-scale measurement infrastructure that consists of diversely-administered hosts. A user of NIMI submits measurement requests that are scheduled for some future time. The focus of this work is to scale the infrastructure to a large number of measurement servers that can execute a broad range of mea-

surement tools in a safe manner. NIMI requires authenticated access for use of its measurement services (in contrast, for example, to Scriptroute's unauthenticated access).

- **iPlane [65]** is a scalable overlay service that provides predictions of Internet end-to-end path properties. iPlane uses structural information (router-level and autonomous system topology), systematic active measurements (sending probes) and opportunistic measurements (monitoring actual data transfers) to gather its measurement data. iPlane participates in BitTorrent swarms and creates connections to existing peers for the explicit purpose of observing network behavior. It then uses the TCP packet traces of the BitTorrent sessions to infer network properties. Using BitTorrent this way does not save any bandwidth; all transfers may consist of actual data (i.e.,not probes) but the data is wasted (it is not stored or uploaded). iPlane can benefit from the Measurement Manager by collecting measurements from existing data transfers without the need to initiate data transfers solely for the purpose of measurement.

- **Sophia [66]** is and information plane for networked systems that is deployed on PlanetLab [67]. Sophia is a distributed system that collects, stores, propagates, aggregates and reacts to observations about the network's current conditions. It provides a distributed logic programming environment for constructing queries about "sensor" measurements of the network. Queries can retrieve cached measurements or induce new ones.

- The **Routing Underlay [68]** is a shared measurement infrastructure that sits be-

tween overlay networks and the underlying Internet. Overlay services can query the routing underlay whenever they need measurements instead of independently probing the Internet on their own.

- The **Congestion Manager [20]**, while not being strictly a measurement service, has been the inspiration for the Measurement Manager. The Congestion Manager (CM) integrates congestion management across all applications and transport protocols between two Internet hosts. The CM maintains congestion parameters and enables *shared state learning* by exposing an API for applications to learn about the network's congestion state. Similar to the Measurement Manager, the CM defines a protocol that resides on top of the IP layer.

7.3 Piggybacking measurements

It is generally accepted that active measurement techniques yield more accurate results and are faster than passive techniques. However, since they inject probes into the network these techniques can be very intrusive and do not scale. To reduce the overhead of active measurement techniques some research has explored the option of piggybacking probes on transport payload. Most of the research in this area has concentrated on manipulating TCP to send inline probes for specific active algorithms. The Measurement Manager on the other hand is a measurement architecture that provides generic inline measurement as an integral part of the architecture and is not dependent on a specific transport protocol or measurement algorithm.

- **TCP Probe [69]** is a sender-side only extension to TCP that estimates the bot-

tleneck capacity of a network path using a portion of the TCP packets as probes. TCP Probe uses the CapProbe [70] algorithm that was originally designed as a standalone active measurement tool. Since CapProbe relies on packet-pair probes to estimate capacity, TCP Probe needs to send a portion of the TCP packets back-to-back and retrieve the arrival (remote) timestamps. Due to the bursty nature of TCP, TCP Probe can easily find TCP packets to send back-to-back [71]. However, retrieving both timestamps from the remote (unaware) TCP side is not as straightforward due to the delayed acknowledgment mechanism of TCP (one ACK generated for every two packets). To circumvent TCP's mechanism, TCP Probe inverts the TCP packets inside a packet-pair, which forces the remote side to generate one ACK for each packet of the packet-pair.

- **TFRC Probe [72]** is another example of a piggybacking technique similar to TCP Probe. In this case, the CapProbe [70] algorithm has been integrated with the TFRC protocol [73] to perform inline capacity measurement using TFRC data packets. TFRC is an equation-based congestion control protocol for unreliable, rate-adaptive applications. Both TCP Probe and TFRC Probe are part of the OverProbe [74] project, which provides a toolkit for the management of overlay networks with mobile users.

- **ImTCP [75]** is an inline measurement mechanism for actively measuring available bandwidth using TCP packets. ImTCP combines a measurement algorithm for available bandwidth [76] with the TCP Reno algorithm. Like the Measurement Manager, ImTCP uses a buffer to temporarily store the TCP packets and then spaces the packets to conform with the sending pattern of a particular probe stream.

The Measurement Manager is more flexible and generic as it can combine any probe stream with TCP traffic and is not tied to a specific measurement algorithm. When the TCP packets are not enough to cover the entire probe stream ImTCP does not send the probe; the Measurement Manager on the other hand sends additional probes on-demand using only partially the TCP packets. ImTCP has been extended to also include inline capacity estimation [77].

- **ICIM [78]** is a variation of ImTCP that does not modify the spacing of TCP packets but adjusts the lengths of TCP bursts. ICIM measures available bandwidth for large bandwidth environments in the presence of interrupt coalescence [79], which has a negative effect on network measurements.

- **TCP Sidecar [80]** is a technique for transparently injecting measurement probes into non-measurement TCP streams. TCP Sidecar injects probes by replaying packets inside a TCP stream and making them look like retransmissions. It does not piggyback on existing TCP payload; all probes occupy additional space but this method has the advantage of passing through firewalls and avoiding detection by intrusion detection systems. The replayed packets can be any size and can be TTL-limited so that they can be used for traceroute-like probing. TCP Sidecar is another example of a system that separates measurement and probing and is not tied to a specific measurement algorithm, much like the Measurement Manager. Passenger [81] is a measurement algorithm based on TCP Sidecar that discovers router-level Internet topology by piggybacking on TCP streams.

- **Sting [82]** uses TCP as an implicit measurement service to measure the end-to-end

packet loss rate between two hosts (in both directions). Sting does not use real data (no piggybacking) but manipulates the TCP connection to get loss measurements. For an example of TCP manipulation, Sting sends packets with overlapping sequence numbers so that each packet contributes only one byte to the the receive buffer. This trick allows Sting to send multiple TCP packets back-to-back at the beginning of a TCP session.

- **Sprobe [83]** uses the packet pair technique to measure the bottleneck bandwidth. Sprobe is inspired by Sting and it manipulates TCP to get its measurements in both downstream and upstream directions.

- Hoe **[84]** uses passive measurements of TCP packets to infer available bandwidth. The estimate is used to set the slow start threshold `ssthresh` (instead of using the default value) thus improving the startup behavior of TCP.

- **TAgent** [18] was our initial attempt at the Measurement Manager architecture, proposing the idea of seamless combination of active probing with application data. TAgent was implemented as a user-space process that proxied both tools and applications, combining their traffic before handing it to the kernel, and likewise demultiplexed the combined traffic at the receiver. Implementing TAgent in user space was the source of many problems. In particular it created a severe performance bottleneck, and it was hard-pressed to properly simulate the kernel under abnormal (e.g., high congestion) conditions.

Jain and Dovrolis [16] propose to integrate probing into application traffic, rather than TCP traffic, to create a custom measurement service. They present a streaming media

overlay network called VDN (for *video distribution network*) that shapes the transmission of streaming media packets to resemble a pathload train, which it can use to infer available bandwidth. MGRP similarly allows application and measurement traffic to be scheduled together.

Skype [17] also probes the network with its own packets to measure available bandwidth and loss rate together [13]. This enables Skype to determine if periods of high loss are due to congestion. If not, then Skype employs a redundancy scheme where it duplicates voice packets that it has already sent to maintain high voice quality.

Compared to these approaches, using MGRP permits applications and measurement tools to be kept separate. MGRP can piggyback *any* application traffic onto measurement probes headed for the same destination, and likewise, an application can easily take advantage of a range of tools. A newly developed tool simply uses the MGRP API to immediately benefit from the lower overhead that comes from piggybacking.

7.4 Measurement in TCP

The Additive Increase Multiplicative Decrease (AIMD) algorithm of TCP Reno is in effect an end-to-end measurement method to roughly estimate the portion of the bandwidth that a TCP flow can use. Since TCP was never intended to be optimal in all environments, a number of TCP variants have been proposed that use different estimators to enhance TCP congestion control in a variety of network environments and traffic conditions:

- **TCP Westwood [85]** uses end-to-end bandwidth estimation to set the values of the

congestion window *cwin* and the slow-start threshold *sshthres*.

- **TCP Vegas [86]** uses round-trip time to measure the current throughput rate and compares it with an expected throughput rate. It then infers the bottleneck backlog and adjusts its congestion window to prevent congestion from occurring.

- **TCP Veno [87]** targets the wireless domain and combines TCP Vegas and TCP Reno. It measures the network congestion level so that it can distinguish between losses due to congestion and losses due to error.

- **Adaptive TCP** A common characteristic of most TCP variants is that they use the same algorithm for every network environment. This makes some TCP variants better suited than others for a specific environment (for example, wireless or high speed paths). While not common, another approach is to make TCP more adaptive, by using multiple estimators depending on the conditions. As an example, TCP Westwood has been extended [88] to select *adaptively* between two estimators based on the predominant cause of packet loss.

7.5 Passive Measurement

Passive measurement techniques, which examine existing traffic to infer characteristics of the network, have seen substantial research. MGRP is an optimization technique for *active* measurement algorithms—by piggybacking application data on measurement probes, the performance cost of active algorithms approaches that of passive algorithms.

Chapter 8

Future Work

In this chapter we briefly sketch plans for further work, including (1) the development of a more general estimation service that could run over MGRP and be used by applications and transport protocols, (2) ways in which the Measurement Manager and the estimation service can optimize file downloads and make operation of TCP more adaptive; and (3) the development of new probing paradigms that account in their design for the fact they may be piggybacked.

8.1 An Estimation Service

In our Measurement Manager architecture, MGRP is the core component. MGRP deals directly with probes and exports a probing service to applications above it. The actual estimation algorithms have remained outside of MGRP. This has been deliberate and has many benefits, since the *estimation process* is separate from the *probing process*, allowing us to optimize each separately. However our work has demonstrated that an *estimation service* could be a natural addition to our architecture. In section 6.2 we introduced the Estimation Overlay and discussed how it addressed the needs of MediaNet. In section 6.4 we used lessons learned from the MediaNet experiments to hint at extensions that would make such an estimation service even more useful.

One way to introduce estimation services to the Measurement Manager is to intro-

duce a new separate component, the *Estimator*, with the task of performing measurements on behalf of applications. When clients, such as applications or transport protocols, need a particular measurement, they can query the *Estimator* which in turn measures the network and responds with an estimate. To come up with an estimate, the *Estimator* uses MGRP to send out probes and collect observations. All the details about the measurement process are abstracted so that the *Estimator* is free to use the algorithm best suited for every request. The challenge is to make the estimator service abstract enough, so that the *Estimator* has latitude in its implementation, but at the same time not too abstract, so that clients find the service useful.

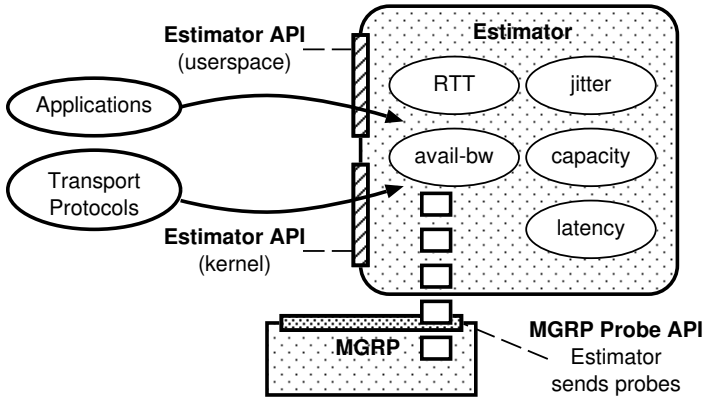


Figure 8.1: We can add an Estimator Service to the Measurement Manager on top of MGRP that implements estimation algorithms and provides measurement services to clients. The *Estimator* can be implemented as a service in user-space (for applications) and in kernel-space (for transport protocols).

Figure 8.1 shows how clients can query the *Estimator* for measurement estimates. We envision the *Estimator* producing simple estimates such as RTT, delay, jitter and loss rate and more involved estimates such as available bandwidth, path capacity and the detection of congestion and bottleneck links. Estimates are associated with a *quality tuple* that indicates how *good* an estimate is in terms of timeliness (age of sample), confidence (how accurate the algorithm is) and overhead (how expensive it was to acquire). Clients

use this information to reason about estimates in their algorithms. Measurement queries can be on-demand queries *(what is the current RTT?)*, periodic queries *(give me available bandwidth estimates every minute)* or event-driven *(notify me whenever jitter exceeds a threshold)*. The challenge is to identify how applications use measurements and thus provide an API that fits their needs.

While applications or network services (such as MediaNet, as shown in Chapter 6) are natural clients for an estimation service, we imagine that transport protocols can also benefit. Consider the case of TCP. Its operation depends on the calculation of the Retransmission Timeout (RTO) that is based on the Round-Trip Time (RTT). The more accurate the RTT, the better the operation of TCP. Instead of calculating the RTT itself, TCP has the option to ask the *Estimator* for periodic RTT estimates, based not only on the packets that the specific TCP flow contributes but based on all packets (TCP or not) that travel along the relevant path. It is clear that with such an approach the estimator can evolve in time to include better RTT estimators without the need for any changes to TCP. These estimators can take into account special path properties like wireless segments. In addition to RTT, we can rework TCP to use the the estimator for more involved estimates, such as delay (used by TCP Vegas [86]), available bandwidth (used by TCP Westwood [85], TCP Nice [89]) or packet loss classification (used by TCP Veno [87]) to detect when losses are due to random transmission errors (as is the typical case in wireless networks) and not due to congestion. In effect, just as the Congestion Manager [20] proposes to extract congestion control services from transport protocol implementations into a separate, reusable module, the Estimation Service could allow network estimates to be handled in the same way, by a separate component.

8.2 Applications of the Measurement Manager

The creation of an Estimation Service can have a major impact on the operation of current applications and transport protocols. In this section we will describe how future work can use the Measurement Manager to optimize file downloading and improve the adaptiveness of TCP in a variety of networking environments.

8.2.1 Optimizing File Downloads

We used the MediaNet overlay in Chapter 6 to demonstrate the utility and benefits of the Measurement Manager for media streaming applications. Future work can integrate MGRP with applications that download files and show how the Measurement Manager optimizes download times. We can use peer-to-peer applications, like BitTorrent [45], or more traditional client-server approaches, like a download manager.

8.2.1.1 BitTorrent

BitTorrent [45] is a file distribution system where clients that are downloading the same file at the same time are uploading pieces of the file to each other. At any given time a BitTorrent peer is connected to multiple other peers (either downloading, uploading or both) and chooses those peers at random or with minimal measurement. This approach can yield suboptimal download times [65]. Future work can show that by using the Measurement Manager, a BitTorrent peer can enhance its operation in multiple ways:

1. **Find best download peers** When a peer downloads from other peers, it needs to find those peers with the best download rate. By using the Measurement Manager a

BitTorrent peer can measure the available bandwidth and capacity of each network path for both used and unused paths and pick the best peer subset.

2. **Find best upload paths** When a peer has finished downloading a file, it needs to determine which peers have the best upload rate, and start uploading pieces of the file to them. By using the Measurement Manager, the peer already has estimates about the upload capacity of paths it has used (since it can measure whenever it exchanges traffic) and can very quickly evaluate the remaining paths (by initiating short uploads).

3. **Detect shared bottlenecks** In both cases (download and upload), a BitTorrent peer can use the Measurement Manager to detect when it is using peers that share the same bottleneck link, which can result in suboptimal performance.

A good BitTorrent client candidate for MGRP modifications is Transmission [90], which is written in C. Future work can modify this client to use the Measurement Manager for the network measurement operations we have outlined above.

Compared to MediaNet, BitTorrent uses TCP in a different way to transport data. MediaNet uses long term TCP sessions between a small set of nodes to send a steady stream of media packets, does not saturate the path and is more interested in the timeliness of packet delivery. BitTorrent, on the other hand, uses much shorter TCP sessions between a potentially large number of nodes, saturates the path and is interested in throughput. So MediaNet and BitTorrent combined provide a well rounded evaluation of the benefits of the Measurement Manager.

8.2.1.2 Download Managers

For a more traditional approach to file downloading, future work can integrate the Measurement Manager with a download manager, such as aria2 [91]. The problem of downloading a file from a number of possible locations is a common problem and websites typically require users to pick a mirror by hand. Download managers are designed to optimize file download speeds. One way that they use is to perform segmented downloading from multiple mirror locations. We can demonstrate how the Measurement Manager enables a download manager to quickly measure the available bandwidth and capacity of all available mirrors, detect bottlenecks and then pick the best subset of mirrors to download from, while all along downloading parts of the file. Our approach is to separate the file downloading into two phases. The first phase is a short measurement phase and the second is a longer download phase.

1. **Measurement Phase** In the measurement phase the download manager measures roughly the downlink paths of all possible mirrors using the Measurement Manager (using fast and high-overhead measurement algorithms). But since all probes can carry transport payload, we can combine this phase with segmented downloading to download chunks of the file inside the probes, in effect canceling out the high overhead. Due to the way that measurement algorithms operate, we may not want to induce congestion during this phase (by using TCP that saturates all available bandwidth). Part of this future work will be to implement a version of TCP (similar to TCP Nice [89]) that sends only when there are probes to piggyback on.

2. **Download phase** Measurements from the first phase will help the download man-

ager to pick a small subset of high-quality nodes (with respect to download speed) and enter the download phase where the remaining segments of the file are downloaded at full speed. For longer downloads (in the order of minutes) we may need to repeat the short measurement phase multiple times to ensure that we keep downloading from the best subset of mirrors.

8.2.2 Optimizing TCP

In addition to providing payload for piggybacking, TCP may also use the *Estimation Service* for measurements (which we touched upon in Section 8.1). Future work can start by modifying existing TCP implementations incrementally to use the *Estimation Service* for each of the following measurements:

1. **RTT estimates** First we can modify TCP to query the *Estimator* for RTT estimates that TCP uses for the estimation of the retransmission timeout (RTO). This will provide TCP with more RTT samples (since it can use packets across all TCP streams), which improves the accuracy of the RTT estimate and leads to a more responsive TCP.

2. **Loss detection** We can take this one step further and delegate loss notifications to the estimator; TCP instructs the estimator to monitor the path, to detect losses and to inform TCP about them. Detecting losses in the *Estimator* instead of in TCP has the potential benefit of differentiating between losses due to errors (such as in wireless environments) and due to congestion. Instead of modifying the TCP algorithm for every special network environment, the *Estimator* can run more elaborate

algorithms (that can change as the network environments change). When TCP gets a loss notification from the *Estimator* it also gets an indication about the nature of the loss, which TCP can use to increase throughput (for example, by not reducing the congestion window in the presence of losses due to errors).

3. **Special measurements** In a similar way we envision using the *Estimator* for more substantial measurements that now are performed in TCP. In future work we can select one of the TCP varieties that need special measurements, such as TCP Westwood [85] (available bandwidth), TCP Vegas [86] (delay) or TCP Veno [87] (wireless), and integrate it with the Measurement Manager.

8.2.2.1 Adaptive TCP

We expect that by removing estimation algorithms we can make TCP simpler and easier to adapt when new algorithms come along or new network conditions become more prevalent (such as wireless environments). In future work we can explore an *Adaptive TCP* approach where TCP detects the current network conditions using the *Estimator* and selects the most appropriate congestion control algorithm on the fly. Contrast this with the typical approach where users need to select manually the best TCP algorithm when requiring optimal performance out of TCP transfers (very high bandwidth transfers, wireless environments, backup transfers). This stems from the fact that TCP was never intended to be optimal in every environment. *TCP can use the Measurement Manager to switch automatically between a number of specialized congestion control algorithms and deliver near-optimal performance for a number of different networking environments.*

8.3 New Probing Paradigms

By retrofitting existing probe tools we do not leverage the full power of MGRP. That is because the current paradigm for probing is to try to send as probes with as little overhead as possible. Tools accomplish this by using specially crafted probe sequences that they expect to be sent with precise timing. These sequences can be quite long (like pathload's 100-probe streams), but since they are spaced apart they do not incur high overhead. We will refer to this as *inelastic probing*, as it does not allow for any flexibility with the probe sizes or gaps. As we demonstrated in Chapter 5, this kind of probing makes suboptimal use of MGRP piggybacking because the burst patterns of probes and applications differ greatly.

What if we change the way that we do probing, now that we have access to application traffic through MGRP? We believe that MGRP can enable a new kind of probing that is more efficient and more effective. Tools can become more flexible in specifying probe sequences, so that MGRP can maximize piggybacking, and in return they can send out many more probes.

We list here new probing techniques that appear promising with MGRP:

1. **Opportunistic Probing:** MGRP can turn every application packet into a probe by wrapping timing information around it. Tools do not have any control over probe sizes or probe spacing, but can have an endless supply of probe data. This resembles passive measurement, but provides additional high-resolution information about the timing of the probes and can collect both sender and receiver statistics. Also, because of the bursty nature of most transport protocols, the "random" probes that are

generated by using application packets are not so random. They look like packet trains with small packet spacing, which can be used by the tool for estimation. This is how a tool would request the probes from MGRP:

```
PROBE SEQUENCE

num of probes   any
probe size      any
probe spacing   up to 200 microsec
```

2. **Elastic Probing:** Opportunistic probing may lack the structure for meaningful estimation of certain network properties. The next step is to add some structure to these “random” packet trains by turning them into “proper” packet trains. The idea is that we micro-shape every TCP burst into a set of ”primitive” packet trains that then we can then plug into estimation algorithms. As an example:

```
PROBE SEQUENCE

num of probes   any
probe size      fixed, 300 bytes
probe spacing   fixed, any up to 200 microsec
```

3. **Semi-Elastic Probing:** If Elastic Probing is still not sufficient for a tool to apply its estimation algorithm, then MGRP can use a relaxed variation of the Inelastic Probing (current probing method used by tools). Instead of demanding an exact probe sequence, the tool can relax the packet size and number of probes in the sequence. The tool can specify a range of valid packet sizes, a range of packet train lengths and minimum thresholds. If for example the tool needs to probe at a certain rate, MGRP can adjust the probe spacing on the fly based on the packet size chosen.

Here is an example how pathload could mitigate the effect of its long probe train when there is no transport payload to piggyback on:

```
PROBE SEQUENCE

num of probes   40..100
probe size      fixed, 300 bytes
probe spacing   fixed, 200 microsec
```

This guarantees to pathload that MGRP will send the probe stream with precise timing and fixed packet sizes, but allows MGRP not to send more than 40 probes per stream if there are no payload packets to piggyback on.

These relaxed constraints allow MGRP to fit better the probes to the available TCP packet bursts and make better use of piggybacking. It also gives the tool a simple way to control the overhead. If there are no TCP packets, then the probes incur the minimum acceptable overhead; if there are TCP packets available, then the packet train expands to make use of them, incurring no extra overhead.

Chapter 9

Conclusions

In this dissertation we have presented the Measurement Manager, a practical, modular, and efficient architecture for performing end-to-end network measurements between hosts. Our core insight was that applications can pool together their data packets to be reused as padding inside network probes. This piggybacking technique saves network bandwidth and reduces overall network losses.

To substantiate our claim we have designed and implemented a new transport protocol, the Measurement Manager Protocol (MGRP), that combines probes and data payload on the fly. In MGRP, active measurement algorithms specify the probes they wish to send using a *Probe API* and applications allow MGRP to use data from their own packets to fill the otherwise wasted probe padding.

We have conducted an extensive series of experiments to empirically evaluate our system. We ported active measurement tools to use MGRP and examined each aspect of piggybacking in detail. As a case study, we built an *Estimation Overlay* that we used with MediaNet, an existing media forwarding overlay. We showed that, compared to sending active probes without reusing their empty padding, piggybacking can improve application throughput, is fair to cross traffic, and reduces the time required to complete measurements.

In this dissertation we have demonstrated that the Measurement Manager architec-

ture is superior to existing measurement services and techniques because it is (1) *flexible*, by providing a generic Probe API that is sufficient to implement a range of active measurement algorithms, (2) *efficient*, by reusing the empty probe padding in a systematic way, and (3) *modular*, by allowing tools and applications to be written separately and seamlessly combined.

In summary, this dissertation makes the following contributions:

- We designed a new architecture that provides a systematic and transparent way for applications to reuse their own traffic for network measurement.
- We designed and implemented a new transport protocol, the Measurement Manager Protocol (MGRP), that implements piggybacking and integrated it with the Linux kernel network stack.
- We analyzed a range of active measurement tools and derived a generic Probe API that tools use send their probes through MGRP.
- We provided experimental evidence that piggybacking data packets on measurement probes is not only feasible but improves source and cross traffic as well as the performance of measurement algorithms, while not affecting their accuracy.
- We showed that the Measurement Manager can be used to build a generic *Estimator Overlay* which can serve the measurement needs of Application Layer Overlays, and provided a specific example with an existing overlay.

The ability of MGRP to *piggyback any data packet* on *any probe* is pivotal in making our measurement system unique in the sense that any measurement algorithm can now

be written *as if active*, but implemented *as if passive*. The Measurement Manager is an architecture with broad applications that saves bandwidth, improves the responsiveness of measurement tools and can be used to build a generic *measurement overlay network* as well as expanding the solution space for estimation algorithms.

Bibliography

[1] "What is BitTorrent," http://www.bittorrent.org/introduction.html.

[2] M. Crovella and B. Krishnamurthy, *Internet Measurement: Infrastructure, Traffic and Applications*. Wiley, July 2006.

[3] J. Postel, "Transmission Control Protocol," RFC 793 (Standard), Sep. 1981.

[4] V. Jacobson, "Congestion Avoidance and Control," in *ACM SIGCOMM*, 1988.

[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550 (Standard), July 2003.

[6] M. Jain and C. Dovrolis, "Pathload: A Measurement Tool for End-to-End Available Bandwidth," in *Passive and Active Measurement Workshop*, 2002.

[7] C. Dovrolis, P. Ramanathan, and D. Moore, "What do packet dispersion techniques measure?" in *IEEE INFOCOM*, 2001.

[8] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient Overlay Networks," in *ACM Symposium on Operating Systems Principles*, 2001.

[9] S. Hansell, "Video is Dominating Internet Traffic, Pushing Prices Up," Article in the New York Times, October 2008, http://bits.blogs.nytimes.com/2008/10/31/video-is-dominating-internet-traffic-pushing-prices-up/.

[10] E. E. Westberg and R. A. Miller, "The Basis for Using the Internet to Support the Information Needs of Primary Care," *Journal of the American Medical Informatics Association*, vol. 6, no. 1, pp. 6–25, Jan/Feb 1999.

[11] B. Tulu and S. Chatterjee, "Internet-based telemedicine: An empirical investigation of objective and subjective video quality," *Decision Support Systems*, vol. 45, no. 4, pp. 681–696, Nov. 2008.

[12] D. Hassoun, "Dynamic stream switching with Flash Media Server 3," April 2008, http://www.adobe.com/devnet/flashmediaserver/articles/dynamic_stream_switching.html.

[13] D. Bonfiglio, M. Mellia, M. Meo, N. Ritacca, and D. Rossi, "Tracking Down Skype Traffic," in *IEEE INFOCOM*, 2008.

[14] L. D. Cicco, S. Mascolo, and V. Palmisano, "Skype Video Responsiveness to Bandwidth Variations," in *ACM NOSSDAV '08*, May 2008.

[15] V. Ribeiro, R. Riedi, R. Baraniuk, J. Navratil, and L. Cottrell, "pathChirp: Efficient Available Bandwidth Estimation for Network Paths," in *Passive and Active Measurement Workshop*, 2003.

[16] M. Jain and C. Dovrolis, “Path Selection using Available Bandwidth Estimation in Overlay-based Video Streaming,” in *IFIP Networking*, 2007.

[17] “Skype web site,” http://skype.com/.

[18] P. Papageorgiou and M. Hicks, “Merging Network Measurement with Data Transport,” in *Passive and Active Measurement Workshop*, March 2005.

[19] P. Papageorgiou, J. McCann, and M. Hicks, “MGRP: Active Measurement, Passively,” Oct. 2008, Under submission to NSDI’09.

[20] H. Balakrishnan, H. S. Rahul, and S. Seshan, “An Integrated Congestion Management Architecture for Internet Hosts,” in *ACM SIGCOMM*, 1999.

[21] M. Hicks, A. Nagarajan, and R. van Renesse, “User-Specified Adaptive Scheduling in a Streaming Media Network,” in *IEEE Conference on Open Architectures and Network Programming*, 2003.

[22] V. Jacobson, “pathchar - a tool to infer characteristics of Internet paths,” Presented at the Mathematical Sciences Research Institute (MSRI); slides available from ftp://ftp.ee.lbl.gov/pathchar/msri-talk.pdf, April 1997.

[23] J. Sommers, P. Barford, and W. Willinger, “A Proposed Framework for Calibration of Available Bandwidth Estimation Tools,” in *IEEE Symposium on Computers and Communications*, June 2006.

[24] J. Sommers, P. Barford, N. Duffield, and A. Ron, “Improving accuracy in end-to-end packet loss measurement,” in *ACM SIGCOMM*, 2005, pp. 157–168.

[25] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang, “Locating Internet Bottlenecks: Algorithms, Measurements, and Implications,” in *ACM SIGCOMM*, 2004.

[26] J. Sommers and P. Barford, “An Active Measurement System for Shared Environments,” in *Internet Measurement Conference*, 2007.

[27] K. Harfoush, A. Bestavros, and J. Byers, “PeriScope: An Active Probing API,” in *Passive and Active Measurement Workshop*, 2002.

[28] N. Spring, D. Wetherall, and T. Anderson, “Scriptroute: A facility for distributed Internet measurement,” in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[29] A. Pásztor and D. Veitch, “A Precision Infrastructure for Active Probing,” in *Passive and Active Measurement Workshop*, 2001.

[30] J. M. Gonzalez and V. Paxson, “pktd: A Packet Capture and Injection Daemon,” in *Passive and Active Measurement Workshop*, 2003.

[31] M. Jain and C. Dovrolis, “End-to-End Available Bandwidth: Measurement methodology, Dynamics, and Relation with TCP Throughput,” in *ACM SIGCOMM*, 2002.

[32] J. E. Sommers, “Calibrated Network Measurement,” Doctoral Dissertation, 2007.

[33] W. R. Stevens, *Unix Network Programming, Volume 1: Sockets and XTI*, 2nd ed. Prentice Hall, 1998.

[34] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, “Vivaldi: A Decentralized Network Coordinate System,” in *Proceedings of the ACM SIGCOMM ’04 Conference*, Portland, Oregon, August 2004.

[35] S. A. Baset and H. G. Schulzrinne, “An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol,” *IEEE INFOCOM*, pp. 1–11, April 2006.

[36] E. Kohler, M. Handley, and S. Floyd, “Datagram Congestion Control Protocol (DCCP),” RFC 4340 (Proposed Standard), March 2006.

[37] Online, “DCCP Stack for Linux,” Linux Networking Wiki, http://linux-net.osdl.org/index.php/DCCP.

[38] J. Corbet, “Linux gets DCCP,” Article on LWN.net, http://lwn.net/Articles/149756/, Aug. 2005.

[39] T. Narten, “Assigning Experimental and Testing Numbers Considered Useful,” RFC 3692 (Best Current Practice), Jan. 2004.

[40] “Assigned Internet Protocol Numbers,” http://www.iana.org/assignments/protocol-numbers.

[41] J. Corbet, “The high-resolution timer API,” Article on LWN.net, http://lwn.net/Articles/167897/, Jan. 2006.

[42] J. Postel, “Internet Protocol,” RFC 791 (Standard), Sep. 1981.

[43] D. D. Clark, “Internet Protocol,” RFC 815, July 1982.

[44] R. Braden, “Internet Protocol,” RFC 1122, Oct. 1989.

[45] B. Cohen, “Incentives Build Robustness in BitTorrent,” in *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[46] D. Pogue, “For Purists, a Cut Above in Movies,” Article in the New York Times, October 2008, http://www.nytimes.com/2008/10/02/technology/personaltech/02pogue.html.

[47] “VUDU FAQs: Networking,” http://supports.vudu.com/categories/Networking/.

[48] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *OSDI*, Boston, MA, Dec. 2002, pp. 255–270.

[49] L. Rizzo, "Dummynet web site," http://info.iet.unipi.it/~luigi/ip_dummynet/.

[50] B. Fink and R. Scott, "Nuttcp web site," http://www.lcp.nrl.navy.mil/nuttcp/.

[51] "iperf: A tool for measuring TCP and UDP bandwidth performance," http://dast.nlanr.net/Projects/Iperf/.

[52] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[53] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's fast recovery algorithm," RFC 3782 (Proposed Standard), United States, 2004.

[54] A. Turner, "Tcpreplay tools," http://tcpreplay.synfin.net/trac/.

[55] "Samplepoint-F 150 Mbps trans-pacific trace (200803201500)," March 2008, http://mawi.wide.ad.jp/mawi/samplepoint-F/20080318/200803201500.html.

[56] Y. Zhang and N. Duffield, "On the constancy of Internet path properties," in *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. New York, NY, USA: ACM, 2001, pp. 197–211.

[57] N. Vegh, "NTools traffic generator/analyzer and network emulator package," http://norvegh.com/ntools/.

[58] A. Shriram, M. Murray, Y. Hyun, N. Brownlee, A. Broido, M. Fomenkov, and kc claffy, "Comparison of Public End-to-End Bandwidth Estimation Tools on High-Speed Links," in *Passive and Active Measurement Workshop*, 2005.

[59] J. Strauss, D. Katabi, and F. Kaashoek, "A Measurement Study of Available Bandwidth Estimation Tools," in *Internet Measurement Conference*, 2003.

[60] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable Application Layer Multicast," in *ACM SIGCOMM*, 2002.

[61] L. Cottrell and SLAC, "Network Monitoring Tools," http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[62] CAIDA, "Performance Measurement Tools Taxonomy," http://www.caida.org/tools/taxonomy/performance.xml.

[63] A. B. Downey, "Using pathchar to estimate Internet link characteristics," in *ACM SIGCOMM*, 1999.

[64] V. Paxson, A. K. Adams, and M. Mathis, "Experiences with NIMI," in *Passive and Active Measurement Workshop*, 2000.

[65] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, “iPlane: An Information Plane for Distributed Services,” in *OSDI*, 2006.

[66] M. Wawrzoniak, L. Peterson, and T. Roscoe, “Sophia: An Information Plane for Networked Systems,” in *Workshop on Hot Topics in Networks*, 2003.

[67] “PlanetLab - An open platform for developing, deploying, and accessing planetary-scale services,” http://www.planet-lab.org/.

[68] A. Nakao, L. Peterson, and A. Bavier, “A Routing Underlay for Overlay Networks,” in *ACM SIGCOMM*, 2003.

[69] A. Persson, C. A. C. Marcondes, L.-J. Chen, M. Y. S. L. Lao, and M. Gerla., “TCP Probe: A TCP with built-in Path Capacity Estimation,” in *IEEE Global Internet Symposium*, 2005.

[70] R. Kapoor, L.-J. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, “CapProbe: A Simple and Accurate Capacity Estimation Technique,” in *ACM SIGCOMM*, 2004.

[71] R. Kapoor, L.-J. Chen, M. Y. Sanadidi, and M. Gerla, “Accuracy of Link Capacity Estimates using Passive and Active Approaches with CapProbe,” in *IEEE Symposium on Computers and Communications*, 2004.

[72] L.-J. Chen, T. Sun, D. Xu, M. Y. Sanadidi, and M. Gerla, “Access Link Capacity Monitoring with TFRC Probe,” in *IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2004.

[73] S. Floyd, M. Handley, J. Padhye, and J. Widmer, “Equation-Based Congestion Control for Unicast Applications,” in *ACM SIGCOMM*, 2000.

[74] “OverProbe: A Toolkit for the Management of Overlay Networks with Mobile Users,” http://www.cs.ucla.edu/NRL/OverProbe/.

[75] C. L. T. Man, G. Hasegawa, and M. Murata, “Available Bandwidth Measurement via TCP Connection,” in *IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2004.

[76] C. L. T. Man, G. Hasegawa, and M. Murata, “A New Available Bandwidth Measurement Technique for Service Overlay Networks,” in *IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2003.

[77] C. L. T. Man, G. Hasegawa, and M. Murata, “An Inline Measurement Method for Capacity of End-to-end Network Path,” in *IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2005.

[78] C. L. T. Man, G. Hasegawa, and M. Murata, “ICIM: An Inline Network Measurement Mechanism for Highspeed Networks,” in *IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*, 2006.

[79] R. Prasad, M. Jain, and C. Dovrolis, “Effects of Interrupt Coalescence on Network Measurements,” in *Passive and Active Measurement Workshop*, 2004.

[80] R. Sherwood and N. Spring, “A Platform for Unobtrusive Measurements on PlanetLab,” in *USENIX Workshop on Real, Large Distributed Systems*, 2006.

[81] R. Sherwood and N. Spring, “Touring the Internet in a TCP Sidecar,” in *Internet Measurement Conference*, 2006.

[82] S. Savage, “Sting: a TCP-based Network Measurement Tool,” in *USENIX Symposium on Internet Technologies and Systems*, 1999.

[83] S. Saroiu, P. K. Gummadi, and S. D. Gribble, “SProbe: A Fast Technique for Measuring Bottleneck Bandwidth in Uncooperative Environments,” in *IEEE INFOCOM*, 2002.

[84] J. C. Hoe, “Improving the Start-up Behavior of a Congestion Control Scheme for TCP,” in *ACM SIGCOMM*, 1996.

[85] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, “TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links,” in *ACM MobiCom*, 2001.

[86] L. Brakmo and L. Peterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet,” *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, Oct. 1995.

[87] C. P. Fu and S. C. Liew, “TCP Veno: TCP Enhancement for Transmission Over Wireless Access Networks,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 2, pp. 216–228, Feb. 2003.

[88] M. Gerla, B. K. F. Ng, M. Y. Sanadidi, M. Valla, and R. Wang, “TCP Westwood with Adaptive Bandwidth Estimation to Improve Efficiency/Friendliness Tradeoffs,” *Computer Communications Journal*, vol. 27, no. 1, pp. 41–58, Jan. 2004.

[89] A. Venkataramani, R. Kokku, and M. Dahlin, “TCP Nice: A Mechanism for Background Transfers,” in *OSDI*, 2002.

[90] “Transmission - A lightweight BitTorrent client,” http://transmission.m0k.org/.

[91] “aria2 - The high speed download utility,” http://aria2.sourceforge.net/.