The Measurement Manager: Modular End-to-End Measurement Services

Pavlos Papageorgiou
pavlos@eng.umd.edu

Ph.D. Research Proposal
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD

April 6, 2007

Abstract — **Network measurement is used to improve the precision, efficiency, and fairness for a variety of Internet protocols and applications, ranging from transport protocols to overlay networks. Measurement is typically performed in one of two ways: (1) *passively*, by using existing data traffic to perform measurements, typified by transport protocols like TCP and passive measurement infrastructures or (2) *actively*, by injecting special probe packets into the network to measure network properties of end-to-end paths, typified by overlay networks that use ping, traceroute and standalone active measurement tools. Neither approach is entirely satisfactory. The passive approach is limited by the shape of application traffic and tends to be tightly coupled with the protocol implementation, hurting modularity. The active approach is faster, more accurate and more modular but imposes a significantly higher overhead, especially at large scales.**

As a remedy to these problems, this research proposal presents the Measurement Manager architecture, a modular approach to performing end-to-end network measurement that includes the best features of the passive and active measurement approaches. Like the active approach, the Measurement Manager allows measurement algorithms to be specified separately from applications, exporting their results as a service to both transport protocols and applications. Like the passive approach, however, the Measurement Manager implements active algorithms in a way that allows probes to be transparently piggybacked on top of existing transport traffic, namely TCP. The piggybacking happens *on-demand* according to various tuning knobs that effectively trade off accuracy and overhead. We propose to evaluate the Measurement Manager architecture in terms of its utility, accuracy, and performance by applying it to several real-world applications and transport protocols and performing experiments that illuminate tradeoffs in the measurement process.

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 RTT, delay, jitter and loss rate, to more involved estimates of available bandwidth, path capacity, and the detection of congestion and bottleneck links. 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.

1.1 Problem description

The network measurement process is not as useful as it could be. Current techniques for end-to-end measurement typically fall into two broad categories: (1) active techniques that inject single or groups of special packets (probes) into the network, and (2) passive techniques that observe existing traffic. Due to the end-to-end nature of the measurements, packets can be observed only as they enter and exit at the two endpoints. These raw packet observations are then post-processed to infer the desired network properties by using estimation algorithms. We perceive two problems with these approaches:

1. **Active measurement is not efficient and is disruptive** Active measurement techniques are usually implemented as standalone tools that measure a specific network property by injecting special probe packets into the network. Since they have full control over the probe sequence, active techniques are usually fast and accurate. However they tend to be intrusive and inefficient because the probe packets consist mostly of empty padding (carrying no useful data) and they interfere with the traffic we are trying to measure. As a result, this approach does not scale. For example, just ping traffic on PlanetLab averages around 1GB a day [1].

2. **Passive measurement is not flexible or modular** Passive measurement techniques are often tightly coupled with the applications using them. This close coupling stems from the desire to lower the overhead of measurement by using the application's own traffic. For example, TCP [2], 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. As another example RTP/RTCP [3] monitors the delay characteristics of the packets of time-sensitive applications. While passive techniques are efficient, the lack of control over the probe sequence makes estimation slower and less accurate than in the active case. In addition, since passive techniques are usually custom-built for each application, it is difficult to reuse them in a modular fashion.

In effect, current measurement techniques are either fast and accurate but not efficient, or efficient but slower, less accurate and ad-hoc. As a remedy to these shortcomings we propose a Measurement Manager, a *hybrid approach* to network measurement that combines the best features of both passive and active approaches. The Measurement Manager system is comprised of two components. The first is an estimator component that exports measurement services to applications and transport protocols and takes the form of an estimation service accessed both from user-space and the kernel. The second is an in-kernel component that sends probes and integrates with Layer 4 of the network stack.

1.2 Proposed approach: The Measurement Manager

The Measurement Manager coordinates all measurement between two end-hosts so that applications and transport protocols can treat measurement as a separate service, using the Measurement Manager to estimate network properties such as available bandwidth, capacity, RTT and loss rate. Our system separates the *estimation process* from the *probing process* and in doing so enables us to bridge the gap between passive and active measurement. The Measurement Manager grew out of our work on merging network measurement with transport protocols [4]. The original idea was inspired by the Congestion Manager [5], an end-to-end system that coordinates congestion information between Internet hosts.

The goal of the Measurement Manager architecture is to provide a measurement service that enables active-style measurement with overhead that is tunable and ranges from completely active to completely passive. Our architecture has four main characteristics: modularity, generality, efficiency and flexibility. It is *modular* because it separates estimation from probing and exports measurement as a service. It is *generic* because it is not tied to any specific transport protocol or any specific estimation technique. Any estimator can be built on top and any transport protocol can contribute packets. It is *efficient* because it makes use of contributed packets to piggyback on empty probes. And finally, it is *flexible* because all estimators can operate as if they used active-style probing and let the Measurement Manager deal with reducing overhead through piggybacking.

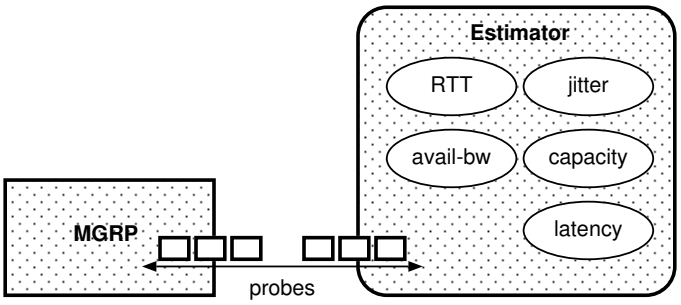


Figure 1: The Measurement Manager is comprised of two components: (i) the *Estimator* that implements estimation algorithms and provides measurement services to the applications, and (ii) the MGRP protocol that sends out probes as efficiently as possible. The *Estimator* is implemented as a service in user-space (for applications) or kernel-space (for transport protocols). MGRP is implemented as a network protocol that is integrated at Layer 4 of the network stack.

To satisfy these objectives, we propose that the Measurement Manager has two components: the *Estimator* and the *Measurement Manager Protocol (MGRP)* (Figure 1). Each of these components provides a different level of abstraction for the measurement process and one builds on top of the other. The *Estimator* is the top component and its main service is to perform measurements on behalf of applications and transport protocols. When applications need a particular measurement, they query the *Estimator* which in turn measures the network and responds with an estimate. To come up with an estimate, the *Estimator* needs to send out probes and collect observations. This is the responsibility of the MGRP protocol, the bottom component of the Measurement Manager, whose main task is to send out probes efficiently.

The key idea is that MGRP can reduce the bandwidth overhead consumed by the empty padding in the probes by filling it with transport payload, in effect piggybacking transport

packets on network probes. When it is time to *actually* send out the probes our system attempts to combine them with useful payload to achieve passive-like efficiency. This payload comes from transport protocols that contribute packets so that the Measurement Manager can use them to fill the empty probe padding. This provides a tuning knob that estimators can use to control the amount of probe reuse; the longer probes can wait, the higher the chance that they will be reused and thus the lower the measurement overhead they incur.

1.2.1 Benefits

Low overhead Our work brings together transport protocols and network measurement. While there has been a lot of work on both passive and active network measurement, previous work focuses on each approach independently. Our work aspires to fill the gap and explore hybrid solutions that leverage actual transport payload to perform active-style measurement with passive-like overhead. Our Measurement Manager turns each packet sent by a transport protocol into a potential network probe, not only by passively capturing timestamps but by actively using the transport payload to occupy what would otherwise be empty padding. The result is that probes can be sent out with lower overhead in a way that is transparent to the sender of the probes. This approach has the additional benefit of making active probing less disruptive to the traffic that we are trying to measure, since most of the probes are piggybacked on existing traffic.

Flexibility Our approach is a hybrid between active and passive measurement because probe reuse happens on-demand; 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. This trades overhead for accuracy. Our Measurement Manager thus permits non-invasive active network measurement such that the incurred overhead can be controlled to match the objectives of each estimator. The consumer of the measurement (applications, transport protocols) can control the amount of overhead they are willing to pay for each estimate.

Simplicity Aside from overhead savings, our 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 can enable probe reuse and minimize the overhead that it incurs.

Moreover, clients that do not have special measurement needs and do not want to deal with the complexity of network measurement can just pick a network property and let the Measurement Manager do all the work (i.e., probing and interpretation of the raw measurements). Providing a straightforward Application Programming Interface (API) makes it more likely for applications to use network measurement effectively and frequently.

Clear deployment path A typical obstacle in deploying measurement infrastructures is that every participating node needs to implement the whole system and know about every estimation algorithm. On the other hand, the Measurement Manager decouples estimation from probe generation and makes estimation a sender-only component. In our system, two nodes that want to participate into measurement need only to implement the probing component, which is much simpler and does not contain any estimation algorithms. The probing component comes in the form of MGRP, a network protocol that integrates at Layer 4 of the network stack and implements probe-sending primitives. Once the probing protocol has been integrated into the stack, any estimator can combine the protocol primitives to generate probes and collect raw results.

1.3 Proposal Structure

This proposal is structured as follows. In Section 2 we present the architecture of the Measurement Manager and describe its two components, the *Estimator* and the MGRP protocol. In Section 3 we show a preliminary implementation of our architecture, an in-kernel implementation of the MGRP protocol with a user-space *Estimator* that measures available bandwidth. In the same section we also present preliminary experiments to test the impact of piggybacking in terms of efficiency and precision. In Section 4 we discuss the work that we propose to complete based on our preliminary implementation and lay out a multi-phase timeline. Section 5 discusses related work and Appendix A presents the two Application Programming Interfaces (APIs) of MGRP.

2 Architecture

This section presents an overview of the the Measurement Manager architecture and its two components, the *Estimator* and the MGRP protocol. We show how the *Estimator* answers measurement queries from applications and transport protocols and how it sends probes through MGRP. We explain how MGRP combines the probes from the *Estimator* with payload from transport protocols and we present a step-by-step example of the operation of our system.

2.1 Estimator

The main service of the *Estimator* is to provide estimates of end-to-end network properties. This service is provided in the form of queries; clients ask for estimates, the *Estimator* uses the results of probe measurements to provide a response. 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 2 shows how clients 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.

2.1.1 Overlays and the *Estimator*

As an example of an application that can benefit from using the Measurement Manager, consider a forwarding overlay such as MediaNet [6]. MediaNet uses available bandwidth estimates to determine along which paths to forward media streams. MediaNet uses TCP for its transport and needs to monitor the available bandwidth on all the paths of the overlay, whether they are used or not. Currently, MediaNet passively keeps track of data sent on each overlay path and uses that as a lower bound of the available bandwidth. But what MediaNet needs is the actual bandwidth so that it can take full advantage of existing paths and also discover new better paths on which it is not currently sending traffic.

Instead of performing the measurement themselves, overlays like MediaNet can use the *Estimator*. MediaNet in particular needs low-overhead periodic estimates of available bandwidth for active overlay paths (on which it forwards traffic) and on-demand estimates for potential new paths, which must be fast but can tolerate higher overhead (since there is no known existing traffic). In both cases the overlay is trying to estimate the same

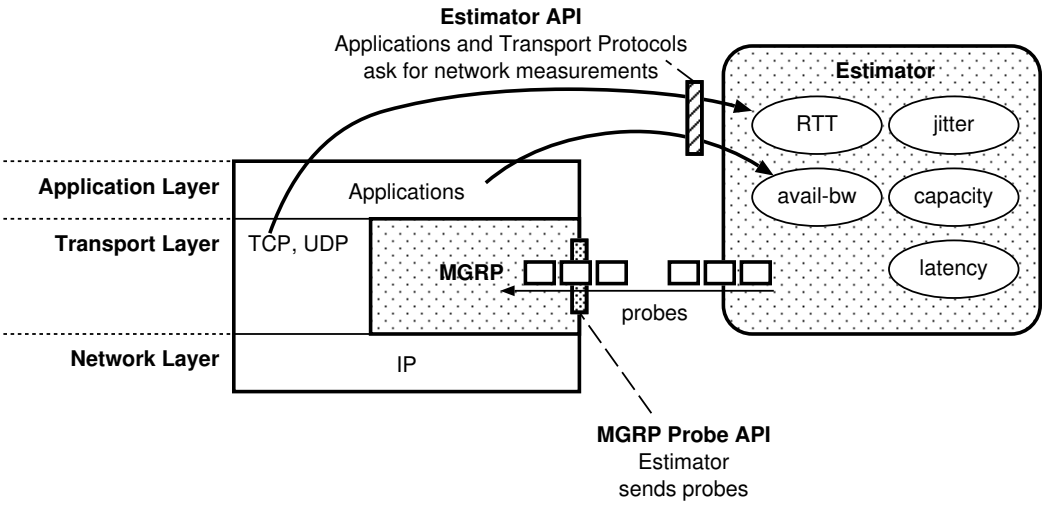


Figure 2: The estimation algorithms in the *Estimator* use MGRP to send out probes. MGRP exports probe-sending primitives through the *Probe API* that algorithms use as building blocks to create probing patterns. MGRP does not know what is being measured; it only knows how to send out a group of probes.

network property (available bandwidth) but the requirements are different, which implies that different algorithms could be used. The API includes quality constraints on the estimates so that the *Estimator* can pick the best algorithm for the job at hand. In return for this service MediaNet needs to contribute transport packets to the Measurement Manager. These packets are used by MGRP to reduce probe overhead by piggybacking the packets on the probes that the *Estimator* sends out (more details in Section 2.2).

2.1.2 Transport Protocols and the *Estimator*

Transport protocols can also benefit from the measurement services of the *Estimator*. 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 [7]), available bandwidth (used by TCP Westwood [8], TCP Nice [9]) or packet loss classification (used by TCP Veno [10]) to detect when losses are due to random transmission errors (as is the typical case in wireless networks) and not due to congestion.

2.2 MGRP: Measurement Manager Protocol

The *Estimator* provides its service through a collection of estimation algorithms that need to send network probes and collect the resulting raw measurement data. It is the responsibility of the Measurement Manager Protocol (MGRP) to fulfill that role by sending probes on

behalf of the estimator, as shown in Figure 2. In a typical usage scenario the estimator asks MGRP to send out packet trains, for example 20 probes with a 10 msec inter-packet gap, and return back packet timestamps. MGRP does not know about the big picture; it does not know what the estimator is estimating. MGRP exports probe primitives that estimators can combine to send out probe patterns according to their estimation algorithm.

While MGRP provides a useful service, the innovative aspect is how this service is implemented to improve performance. One way to reduce the probe overhead is to develop estimation algorithms that use fewer probes. But MGRP follows a complementary strategy: it targets the probes themselves by trying to reuse transparently the empty padding that probes carry. The key component to making this idea work is that transport protocols can share their packets with MGRP (before going over IP) so that when there is an empty probe waiting (and going to the same destination) MGRP can combine the two packets instead of sending out separately the probe (with empty padding) and the transport packet. This has the added benefit of reducing the total number of packets on the network, since transport packets piggyback on probes.

2.2.1 Piggybacking payload on probes

Piggybacking happens transparently to its clients (transport protocols and estimators): they push packets in on one end and packets pop up on the other end (Figure 3). Internally, MGRP distinguishes two types of packets: (i) *probe packets*, which are packets that have space to carry other payload, and (ii) *payload packets*, which are packets full of payload that need to get across, like transport packets. The whole operation of MGRP centers around matching probes with payload, without violating the constraints of each. Once a match is found, the probe and the payload are combined in an MGRP packet, sent across the network and demultiplexed at the receiver. This process of matching probes and payload while abiding by the individual packet timing and spacing concerns presents us with interesting engineering challenges.

The fact that MGRP tunnels transport packets led us naturally to integrate MGRP at Layer 4 as a protocol in the network stack, sitting just above the IP layer. Unlike transport protocols at Layer 4 , MGRP can be thought of as occupying the space between transport protocols and IP. Transport protocols share their packets with MGRP by passing their packets to MGRP before sending them over IP. For times when applications may want to access it directly, MGRP resembles a transport protocol and exports a socket API.

2.2.2 MGRP API

MGRP exports two APIs: (i) a *Probe API*, that is used by estimators to send probes, and (ii) a *Payload API*, that is used by transport protocols to contribute packets.

Probe API The Probe API exports measurement primitives that can be used to send probes and collect the resulting raw measurement data. After studying a number of existing standalone measurement algorithms, pathchar [11, 12], pathload [13], pathchirp [14], pathrate [15], pathneck [16], and a number of measurement services, Periscope [17], Scriptroute [18], precision probing [19] and pktd [20], we have identified a number of

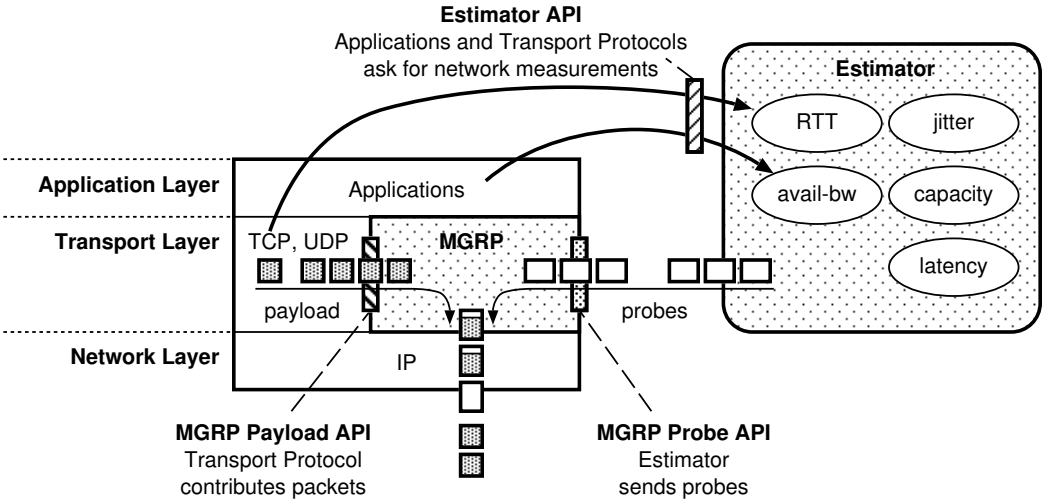


Figure 3: MGRP receives *probe* packets from the *Estimator* and *payload* packets from transport protocols. The main objective of MGRP is to fill the empty padding in the probes with as much payload as possible. The process of piggybacking transport packets on probes happens transparently and results in lowering the overhead of sending out probes.

building blocks that need to be supported by our probing API:

- Send single probe
- Send packet pair
- Send packet train with uniform or variable inter-packet gaps
- Send a hybrid packet train mixed with special TTL-limited or ICMP probes

When the estimator needs to send probes, it picks a probe primitive, creates all the probe packets and hands the packets to MGRP along with instructions for each packet. MGRP needs to know which portion of the probes can be reused *(where is the empty padding?)*, the time gap between packets *(what is the shape of the packet train?)*, the constraints on the probes *(can they wait?)* and the information the estimator needs back *(which timestamps to collect?)*.

Payload API To interact with transport protocols for piggybacking, MGRP exports the Payload API. When a transport protocol *contributes* payload the given payload is transported in an MGRP packet that goes over IP. But MGRP is not as simple as "just another tunneling layer". It combines packets and may delay them so that it can maximize probe reuse. Such actions may interfere with the operation of a transport protocol, especially TCP. That is why MGRP provides the Payload API so that transport protocols can cooperate with MGRP for the piggybacking in an efficient and non-invasive manner. Transport protocols use the API to describe their time constraints *(this is the packet deadline)*, get feedback *(tell me when the packet actually left)*, and allow re-packetization *(here is how you can segment the packet)*.

2.3 The Measurement Manager by Example

Consider an overlay like MediaNet that uses TCP as its transport and needs to constantly monitor the available bandwidth of a particular path. Let us review the example in Figure 4 where each circled step corresponds to an entry in the following list:

1. MediaNet sends periodic queries to the *Estimator* for available bandwidth with specific constraints about the quality, timeliness and overhead of the measurements. The *Estimator* picks the appropriate algorithm and starts generating probes by sending them to MGRP. Assume that the estimation algorithm generates a combination of packet pairs (two probes) and packet trains consisting of three probes. MediaNet can tolerate some overhead so MGRP can send out probes even if they are not fully reused (see packet train in (ii)).

2. MediaNet uses TCP to forward overlay traffic as usual.

3. Independently from MediaNet, TCP uses the *Estimator* and requests continuous RTT estimates. Contrary to the probes generated for the available bandwidth estimation, the *Estimator* generates probes for RTT estimation that need to be fully reused; in effect forcing MGRP to send out a probe only if a TCP packet is available (see packet train in (i)).

4. TCP contributes its packets to MGRP instead of directly passing them to the IP layer. If no probes are found for piggybacking, MGRP sends the TCP packet encapsulated in an MGRP packet (instead of piggybacking it on a probe, see packets in (iii)).

5. From MGRP's perspective, the probes that it receives from the *Estimator* look like empty vessels and the transport packets it receives from TCP look like riders full of payload. MGRP tries to match riders with vessels, combines them in one MGRP packet and pushes the packet into IP.

6. The MGRP packets look to IP as packets carrying a new transport protocol and are forwarded across the network as usual. Each packet carries an IP header, an MGRP header and one of three types of payload (bottom of Figure 4): (i) a probe with a transport packet piggybacked, (ii) a probe with empty padding (first two probes; the third is partially reused), and (iii) a transport packet without any probe attached.

7. The receiver receives the packets and the IP layer passes them to MGRP. IP cannot tell whether the packets contain TCP payload or probes.

8. MGRP reverses the piggybacking process and demultiplexes riders from vessels. Specifically: (a) MGRP extracts the piggybacked TCP payload before passing it to TCP, and (b) reconstructs the container probe before passing it to the *Estimator*.

9. The *Estimator* (at the receiver) processes the probes and returns the estimate out-of-band to the *Estimator* (at the sender).

10. The *Estimator* (at the sender) returns the estimate to MediaNet.

This example demonstrates how transport packets piggyback on probes and how MGRP combines the two types of packets. Both the overlay and TCP are separately doing

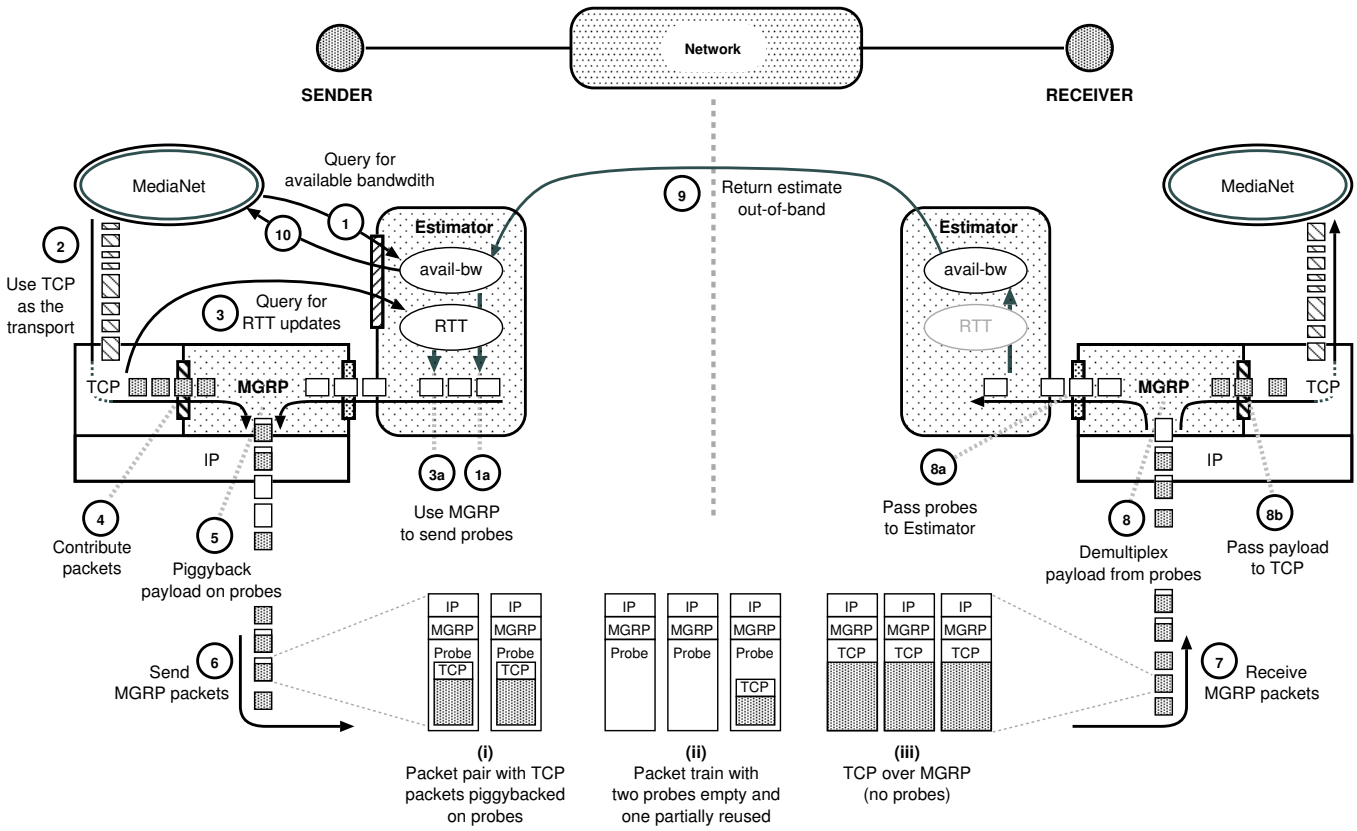


Figure 4: A step-by-step example to demonstrate the operation of the Measurement Manager. MediaNet uses the *Estimator* to get periodic estimates of the available bandwidth of the path. The *Estimator* generates probes that MGRP combines with TCP packets that MediaNet is sending. Depending on the timing, MGRP packets can contain probes that are fully reused (i), probes that are partially reused (ii) or when no probes are available MGRP packets contain only TCP payload (iii). The combined packets are demultiplexed by MGRP at the receiver.

measurement with the same packets that also happen to be their own. The separation of estimation from probing makes this possible. The example shows how probes are reused on-demand; some MGRP packets carry only probes, some only TCP payload and some carry probes with the TCP payload piggybacked. MGRP combines TCP payload with probes transparently and with minimal changes to the natural flow of TCP packets, which always leave in the order intended. The portion of the TCP packets that are reused (typically a small fraction of the overall TCP traffic) may leave with different segmentation (so that they can fit in the probes) and with different timing (to conform to the probe timing).

3 Preliminary Implementation and Results

Of the two components of the Measurement Manager architecture, so far we have implemented the bulk of the MGRP protocol and a small part of the *Estimator*. We have implemented MGRP as a Layer 4 protocol in the Linux kernel and we modified TCP in Linux to contribute packets to MGRP. For the *Estimator* we have modified pathload [13], an existing measurement tool, to support available bandwidth estimation. Figure 5 shows our current implementation of the Measurement Manager. Using our implementation we conducted several experiments to understand the impact of piggybacking in terms of network efficiency and measurement precision. In particular, we measured potential bandwidth savings due to probe piggybacking as well as the impact of piggybacking on the accuracy of the measurement results.

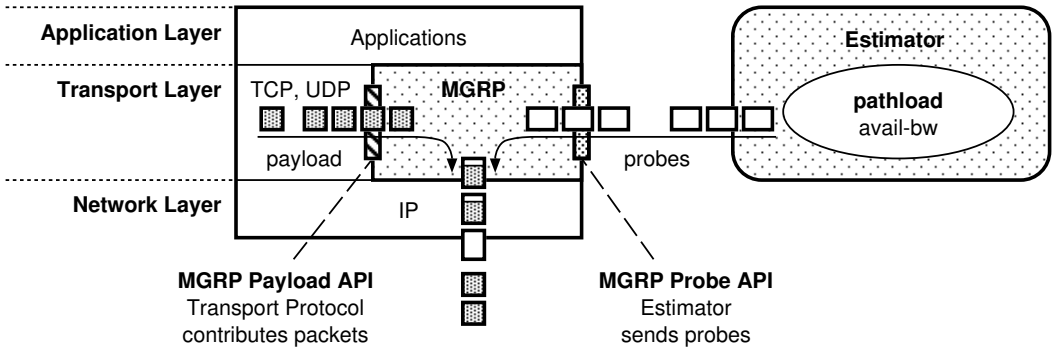


Figure 5: Our current implementation of the Measurement Manager. MGRP has been implemented in the Linux kernel as a network protocol at Layer 4 . TCP has been modified to contribute packets to MGRP instead of sending them directly to the IP layer. The *Estimator* resides in user-space and sends its probes through the *Probe API* (a socket API). The *Estimator* implements the pathload algorithm that measures available bandwidth (extracted from an existing standalone tool).

3.1 Estimator

In our design, the estimator is comprised of a collection of estimation algorithms. It was natural to start by adapting existing tools into the architecture. We started with pathload [13], an active measurement tool that measures available bandwidth, which we modified to send probes through the MGRP Probe API instead of using UDP probes. Pathload was a good candidate for many reasons: it measures available bandwidth that is a property needed by many applications and overlays that we will experiment with, like MediaNet [6] and BitTorrent [21]. It produces simple packet trains with uniform gaps so the probes were simple for MGRP to generate, it 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, and the tool is robust and fairly accurate [22, 23].

3.1.1 Pathload

Pathload measures the available bandwidth on an end-to-end path between a source and a destination. The basic idea in pathload is that the one-way delays of a periodic packet stream show increasing trend when the stream rate is larger than the available bandwidth [13].

Pathload operation Pathload operates in rounds. In each round, it generates trains of probes that it sends from source to destination. Each train consists of 100 probes (but this is configurable) that are equally sized and equally spaced and correspond to a certain instantaneous probing rate. The packet sizes and gaps remain the same during each round but can change between rounds as the probing rate changes. Pathload estimates the available bandwidth by analyzing the one-way delays of its probes, in essence analyzing how the gaps between packets get distorted. Pathload consists of two applications: (i) $pathload\_snd$ runs at the source node and generates the packet trains between source and destination, (ii) $pathload\_rcv$ runs at the destination node, receives the packet trains and analyzes the one-way delays to estimate the available bandwidth.

Modified operation In its original implementation, pathload generates its probes from user-space using UDP. The gap between the probes is generated by a combination of busy looping and sleeping between successive invocations of $send()$. In our modified version, pathload uses the Probe API to schedule packet trains with MGRP that generates the gap between the probes and sends them out. When the pathload packet trains are handed to MGRP, it tries to find payload to piggyback on the empty probes.

MGRP may delay the whole packet train for a few milliseconds ($2 - 15msec$) to increase the chance of reusing the probes. The piggybacking happens transparently to pathload. Since MGRP may delay the probes, all timestamps are taken in the kernel and MGRP at the receiver makes sure to overwrite pathload's timestamps with the actual send timestamps before handing the probes to Pathload (send timestamps are stored in the probe as application data).

3.2 MGRP: Measurement Manager Protocol

We have implemented MGRP as a loadable Linux kernel (2.6.18) module. Our module is written in C and requires no changes to the kernel. We have built upon the kernel mechanisms introduced by DCCP [24, 25, 26], which generalize the parts of the Linux network stack that deal with transport protocols, such as registering with the IP layer and exporting a socket interface. Once the kernel module is loaded, the behavior of the Linux kernel changes in the following ways:

- The kernel knows about a new Layer 4 protocol, the MGRP protocol, and hands all incoming IP packets with the MGRP protocol field to our module. For experimental purposes [27, 28] we picked 254 as the MGRP protocol number.
- Transport protocols can use the Payload API to contribute their outgoing packets to MGRP before they go over IP.

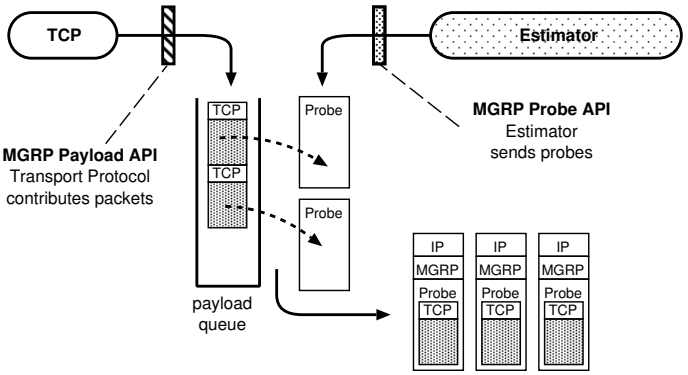


Figure 6: MGRP uses the payload queue to decrease the chances of a probe going out without transport payload. MGRP buffers all transport packets in the queue (for a small fixed delay around $2 - 15msec$) and uses them to fill probes as the come in.

- Applications can open an MGRP socket, a new type of datagram socket, that they can use to send probes over MGRP. This is the socket version of the Probe API.
- Estimators can use the Probe API to send probes using MGRP.

To the entities above (estimators and transport protocols), it appears that MGRP serves two unrelated roles: (i) sending probes through the Probe API, and (ii) sending transport packets through the Payload API. But in fact those two roles complement each other since MGRP internally matches payload packets with probes. As a result probes do not need to contain so much padding (since they carry payload) and the overhead for sending probes is reduced. The two APIs are illustrated in Figure 5 and are presented in more detail in Appendix A.

Probe packets MGRP processes probe packets as follows. A probe request comes from an estimator and typically involves a number of probes that need to be sent out as a group (a packet train is a good example). The estimator does not generate the gaps between the packets; it just hands all the probes to MGRP along with instructions about how to send them. These instructions describe the gaps between packets, the type of packets and most importantly which portion of the packet is empty padding. Instead of probes, MGRP sees partially filled "vessels" that have empty space and can still carry payload. So MGRP tries to match the probes with payload, in effect piggybacking transport payload on probes. Once a match is found (or the deadline for the request expires), MGRP sends out the whole group of probes with the proper gaps. This kind of piggybacking happens on-demand; if no data is found to fill the empty padding then the probes go out empty. The chances of reuse are good because the first probe of the group can tolerate some delay, as long as the estimator is aware of it. But once the first probe goes out, the rest of the probes need to be sent out at precise intervals relative to the first one.

Payload packets A payload packet comes from a transport protocol that offers the packet for piggybacking but still expects MGRP to send the packet across (i.e., it does not abandon the packet). MGRP treats contributed packets as "riders" that are looking for an empty vessel and adds them to a payload queue (Figure 6). MGRP uses this queue as a

look-ahead buffer to decrease the chances of a probe going out with empty padding instead of payload. The effect of the queue on transport protocols is to increase the perceived RTT by a configurable amount ($2 - 15msec$ in our implementation). The Payload API provides ways for transport protocols to limit buffering and set temporal constraints on their packets.

3.2.1 Changes to TCP

New socket option While MGRP does not require any changes to the kernel, we changed the in-kernel TCP so that we could test the probe reuse feature of MGRP. We have added a socket option to TCP that makes the current TCP flow to contribute packets to MGRP (Figure 7).

```
/* 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 7: Code sample to enable TCP over MGRP operation. We have modified TCP in the Linux kernel to accept the `TCP_MGRP` socket option. Once turned on, the socket option causes the current TCP socket to contribute packets to MGRP.

TCP Segmentation Offload While not an in-kernel change, we also enable TCP Segmentation Offload (TSO) so that we avoid issues with fragmentation and small packets. In TSO mode TCP generates packets much larger than the MTU and counts on lower layers to segment the TCP packets appropriately. This gives MGRP the flexibility to fit TCP payload with probes better and avoids problems with very small TCP segments (MTU-sized TCP segments have to be broken in two segments to fit in probes due to header overhead). TSO was originally developed to optimize performance by offloading TCP segmentation to network cards but has been extended to allow optimization across all network layers (see Generic Segmentation Offload [29] in Linux).

3.3 Experiments

We have conducted experiments to begin to evaluate the costs and benefits of the Measurement Manager. First, we started to validate our implementation in two respects. We verified that pathload produces the same estimates when running over MGRP and that TCP exhibits the same behavior when it contributes packets to MGRP. Second, we studied the benefits of piggybacking and the tradeoffs between delaying transport packets and reusing probes.

3.3.1 Experiment Setup

To answer these questions we constructed an experiment on Emulab [30] and the topology can be seen in Figure 8. UDP cross traffic travels from A to Z in a stairwise fashion, as shown in Figure 9(a). The remaining traffic travels from S to D. The goal is to estimate the available bandwidth between two nodes S and D, which in effect is the available

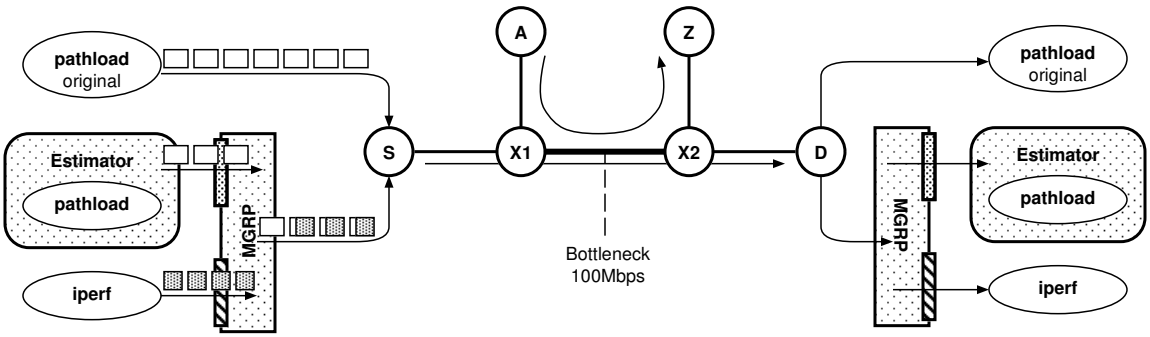


Figure 8: In this experiment we have two instances of pathload running (original and modified) between S and D, with only one instance active during any experiment run. Iperf generates multiple non-overlapping TCP sessions between S and D and all TCP traffic is contributed to MGRP. UDP cross traffic travels between A and Z and follows a stairwise pattern.

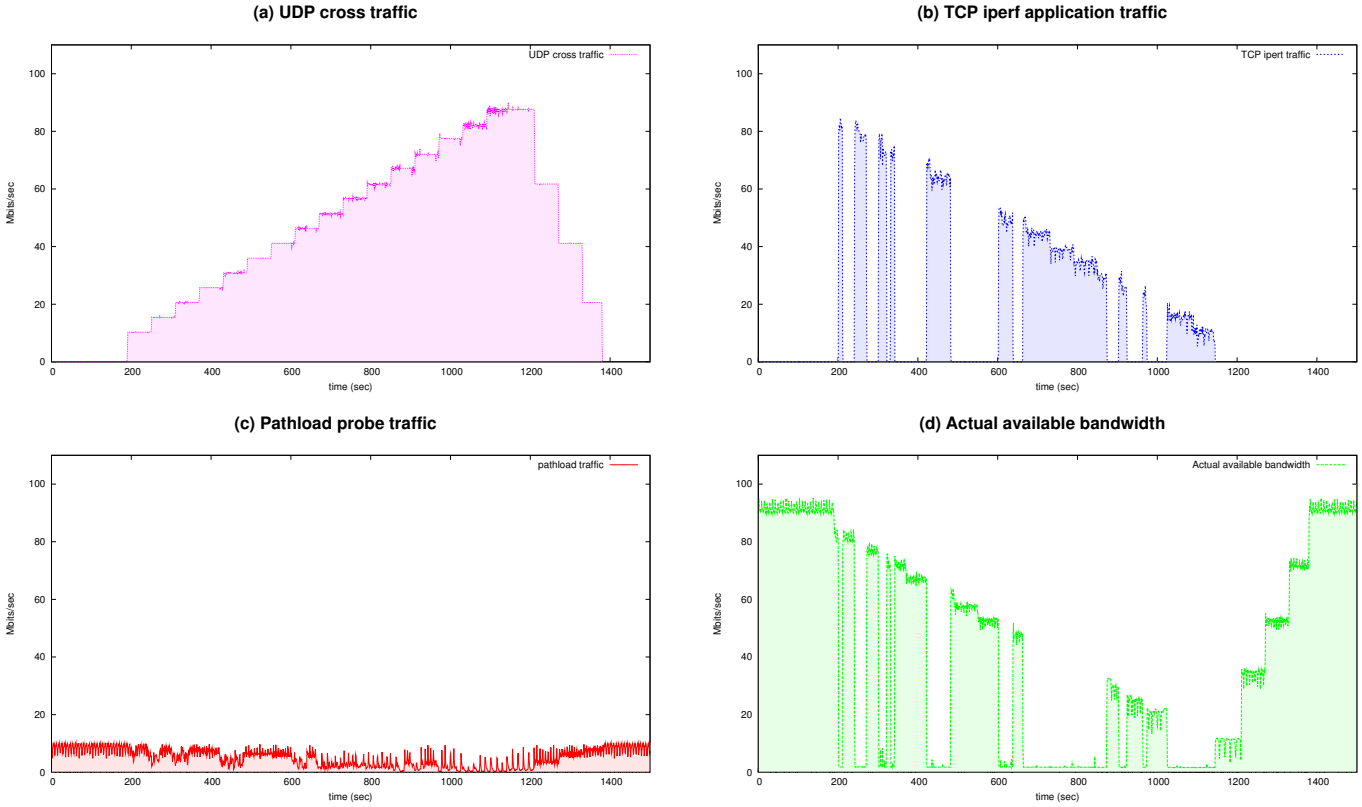


Figure 9: Instantaneous bandwidth at bottleneck link (averaged over 1sec) computed from tcpdump packet traces. Plot (a) shows the bandwidth consumed by UDP stairwise cross traffic that travels from A to Z. Plot (b) shows the bandwidth consumed by TCP traffic (multiple non-overlapping TCP sessions) that travels between the two endpoints S and D. Plot (c) shows the bandwidth consumed by pathload probes that travel from S to D. Plot (d) shows the actual available bandwidth at the bottleneck link $(100Mbits/sec - (plot_a + plot_b + plot_c))$.

bandwidth at the bottleneck link between $X1$ and $X2$. The actual available bandwidth that we are trying to measure appears in Figure 9(d). The bandwidth in the figures represents instantaneous bandwidth averaged over 1sec intervals that is calculated using tcpdump packet traces on the link $X1$, $X2$.

TCP iperf traffic We use iperf [31] to generated multiple non-overlapping TCP sessions with durations ranging from $5sec$ to $180sec$ between S and D, as shown in Figure 9(b). We have inserted delays between iperf sessions which appear in the figure as gaps between the shaded TCP traffic regions. The iperf TCP flows contribute their packets to MGRP (which then passes them to IP), so that MGRP can piggyback iperf packets on probes. To measure the available bandwidth we run pathload continuously between S and D, which consumes $8 - 10\%$ of the available bandwidth in overhead (see Figure 9(c)). Each pathload run produces one estimate and typically takes 5-30 seconds to converge. In each run of the experiment, we either use the original pathload (that goes over UDP) or our modified version of pathload (that goes over MGRP).

Parameters The experiment is parameterized in three ways: (i) the fixed delay in milliseconds that TCP packets stay in the payload queue in MGRP, (ii) the maximum delay in milliseconds that pathload packet trains can be delayed, and (iii) the probe reuse threshold over which a packet train that is waiting can leave (even though not fully reused; applies only when probes are waiting). All three parameters aim at controlling the amount of transport payload that can be piggybacked on probes.

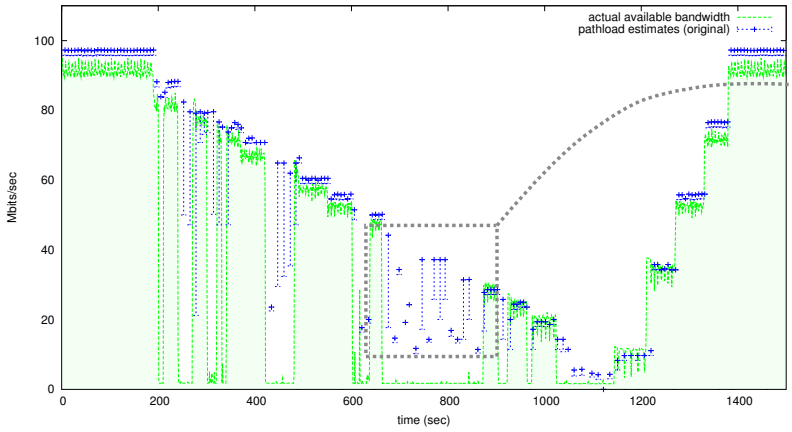
3.3.2 Pathload accuracy

We run three sets of experiments to verify that MGRP handles probes correctly. In the first set we run the original instance of pathload (over UDP), in the second we run our modified version that sends probes using MGRP but without piggybacking and in the third set we used MGRP with piggybacking. In all three cases pathload churns out estimates continuously. The UDP cross traffic travels as usual between A and Z. To eliminate any interference with the probes, we run the iperf TCP traffic in its original form (not through MGRP). The plots in Figure 10 show (a) the estimates of the original pathload, (b) the estimates of the modified pathload using MGRP without piggybacking, and (c) the estimates of the modified pathload that uses MGRP with full piggybacking.

The plots show that the modified pathload operates with the same accuracy as the original when there is no TCP traffic and only the UDP cross traffic is present (TCP sessions saturate the link and can easily be spotted in the plots by the "dips" in the actual available bandwidth). In the presence of TCP traffic coming from the measurement host (see the dotted box in Figure 10 for an example) all pathload versions overestimate the available bandwidth but the modified pathload versions appear to produce estimates that are higher than the original pathload. This happens because the algorithm is not adjusting for probe reuse. As part of the proposed work (see section 4.1.3) we are going to study how estimation algorithms can adapt their estimates when operating over MGRP.

The difference between the estimates of the original and modified versions of pathload may also point to an unintended but valuable feature of MGRP. Looking closer at the dotted box in the plots in Figure 10(b)(c), it appears that the modified pathload produces estimates that track the UDP cross traffic but not the TCP traffic that is generated from the measurement host. It appears that because of the way that MGRP sends and reuses probes it may "hide" traffic originating from the measurement host. This can be valuable because we filter out the effects of the current host (where we have control) and measure only the effect of other traffic. As part of the proposed work we are going to verify if this is indeed the case and study its implications.

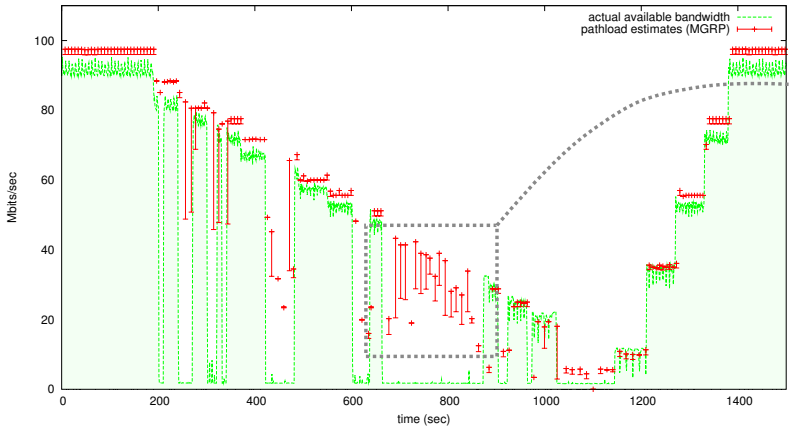
(a) Original pathload over UDP



During this time TCP saturates the bottleneck link and the available bandwidth is in effect zero.

The original pathload has some trouble but there is a clear trend for lower estimates.

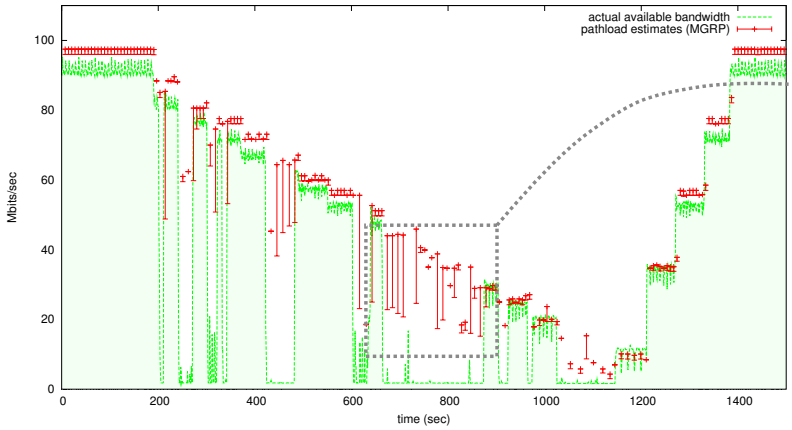
(b) Modified pathload over MGRP with no piggybacking



The modified pathload produces estimates that are higher than the original pathload. The estimates appear to follow the stairwise trend of the UDP cross traffic but with higher variation.

One interpretation for this is that because of the way we generate packet trains in the kernel, the probes do not interleave with the TCP packets coming from the measurement host. This results in some of that traffic being "hidden" from the estimator.

(c) Modified pathload over MGRP with maximal piggybacking



The modified pathload with full reuse of probes produces estimates that are a little bit higher and with less variation than the modified pathload with no reuse. The estimates in this case follow more closely the stairwise trend of the UDP cross traffic.

One interpretation for this is that most of the pathload probes have TCP payload in them, which means that the pathload algorithm over-estimates the available bandwidth.

Figure 10: Pathload estimates of available bandwidth at bottleneck link using (a) the original pathload, (b) the modified pathload over MGRP but with no probe reuse, and (c) the modified pathload with full probe reuse. Whenever there is no TCP traffic (outside of the dips in the plots) there is only UDP cross traffic present and all versions estimate accurately the available bandwidth. When there is TCP traffic (see dotted box for an example) the modified versions overestimate the available bandwidth compared to the original one. *As part of the proposed work (see section 4.1.3) we are going to study how estimation algorithms can adjust their estimates when operating over MGRP.*

3.3.3 Impact on TCP

We run three sets of experiments to evaluate the impact of MGRP on TCP. We are specifically interested in how the payload queue (where TCP payload is delayed to increase the chances of piggybacking) affects TCP throughput (see Section 3.2). The first set of experiments uses the original TCP. The second set uses TCP over MGRP with piggybacking and the third set uses TCP over MGRP but with piggybacking disabled. In cases 2 and 3 (TCP over MGRP) we vary the delay of the MGRP payload queue from $0msec$ to $15msec$ and in case 1 (original TCP) we emulate the payload queue delay by varying the link delay of the bottleneck link in the same way.

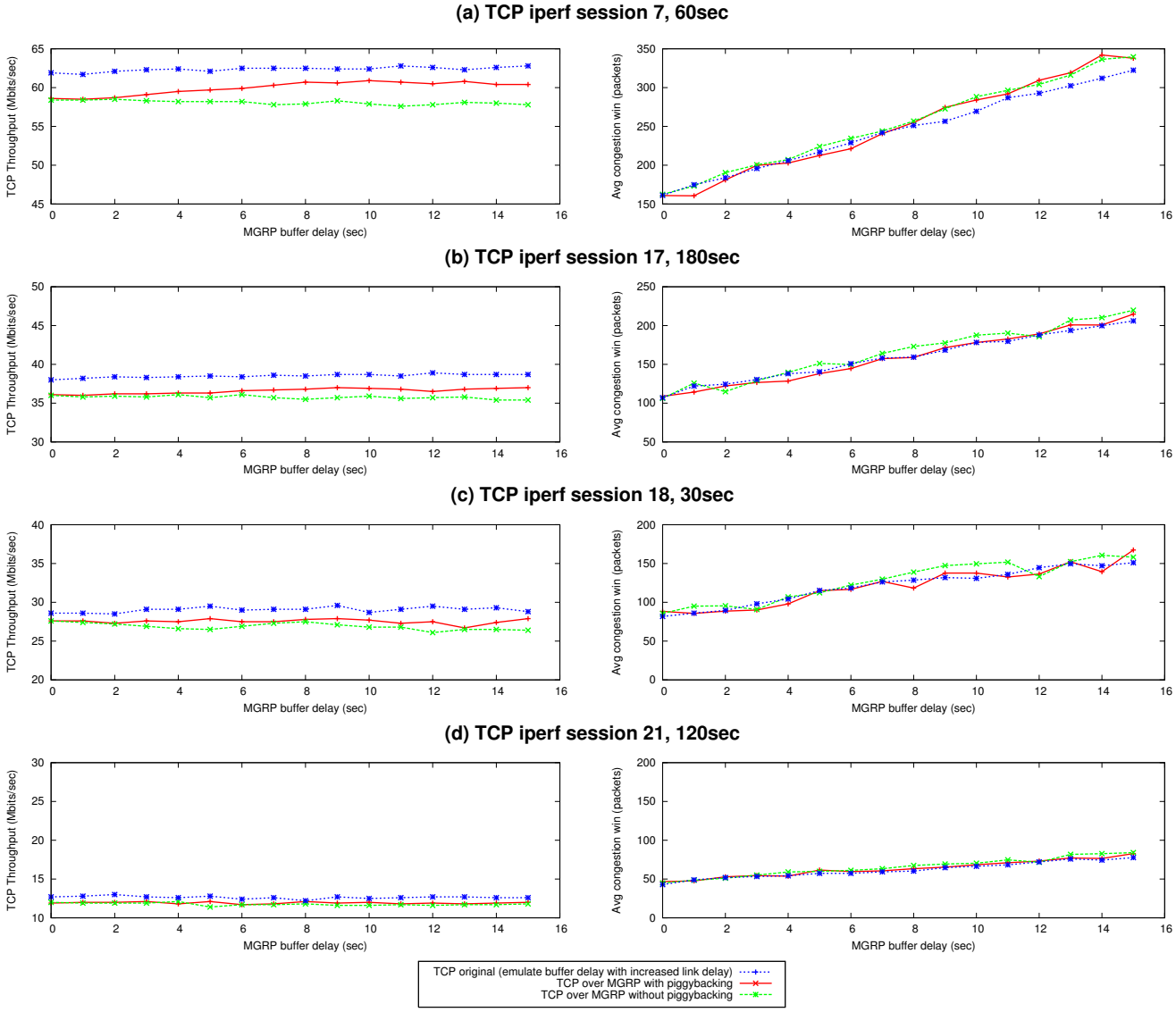


Figure 11: Average TCP throughput (left) and average congestion window (right) of four TCP iperf sessions. Each line represents one set of experiments and in each set we varied the delay that TCP packets experience. This delay is added by the MGRP payload queue and the plots show that, other than a relative (but constant) decrease in throughput due to MGRP overhead, TCP over MGRP behaves in the same way as the original TCP. The plots of the average congestion window indicate that the payload queue delay looks to TCP exactly like path delay. All throughput plots use the same scale but show different throughput ranges (same for congestion window plots).

Figure 11 shows the average TCP throughput (plots on the left) and the average congestion window (plots on the right) for four TCP sessions. From the plots, we see that the average congestion window is very similar across all three sets of experiments. This indicates that the payload queue affects TCP in the same way as a link with the same amount of delay. The throughput for both the original TCP and TCP over MGRP (no piggybacking) are not affected by the queue delay and the difference between them is due to the overhead of the MGRP protocol. The throughput of TCP over MGRP (with piggybacking) increases as the queue delay increases because MGRP combines probes with TCP packets and as a result reduces the number of packets that TCP contends with.

3.3.4 Piggybacking

We run a series of experiments to demonstrate the effect of piggybacking on the total probe overhead, and specifically how the delay of transport payload affects the probe reuse ratio.

Probe Reuse Ratio In Section 3.2 and Figure 6 we saw that one way that MGRP can control the portion of probes that are reused is to buffer transport payload for a fixed amount of time thus increasing the chance that when probes come MGRP has payload to piggyback on them. The experiments show what one would expect; the more payload is buffered the more probes are reused. Figure 12 summarizes multiple experiment runs and shows how the probe reuse ratio depends on the delay of the payload buffer. For our particular case, a delay between 6msec and 8msec gives a good reuse ratio between 0.6 and 0.8 (with 1.0 being full reuse).

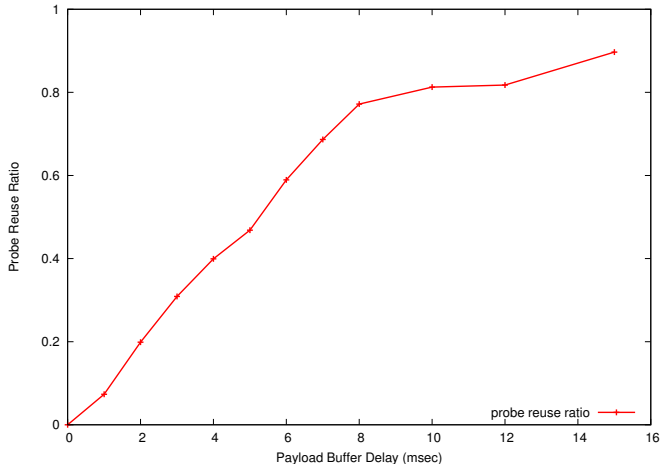


Figure 12: Effect of payload buffer on the probe reuse ratio. Each point corresponds to one experiment run and represents the ratio of probe bytes that were reused. The plot shows that when more payload is buffered (by increasing the buffer delay) the number of probes that are reused increases.

To calculate the probe reuse ratio we needed to include only the probes that could be reused. Pathload keeps sending probes but only when there is TCP traffic available, probe reuse is possible. The rule that we followed was to divide time in bins of 100msec and then include only the probes in bins where there is at least one TCP packet. This provides us with a rough estimate of the reuse ratio but counts more probes than can be reused. That is why the reuse ratio does not reach 1.0 in Figure 12.

Piggybacking Visualization We will use the the 4 iperf TCP sessions in Figure 13(a) to visualize how MGRP saves probe overhead through piggybacking. The plots in Figure 13(b,c,d,e) show the bandwidth consumed by pathload and the portion of that bandwidth that is reused by piggybacking TCP payload. Each plot in Figure 13(b,c,d,e) corresponds to a different experiment run with different delay values for the payload buffer: (b) 8msec, (c) 4msec, (d) 2msec, and (e) no buffering. We can see that probe reuse diminishes as the delay of the payload buffer decreases.

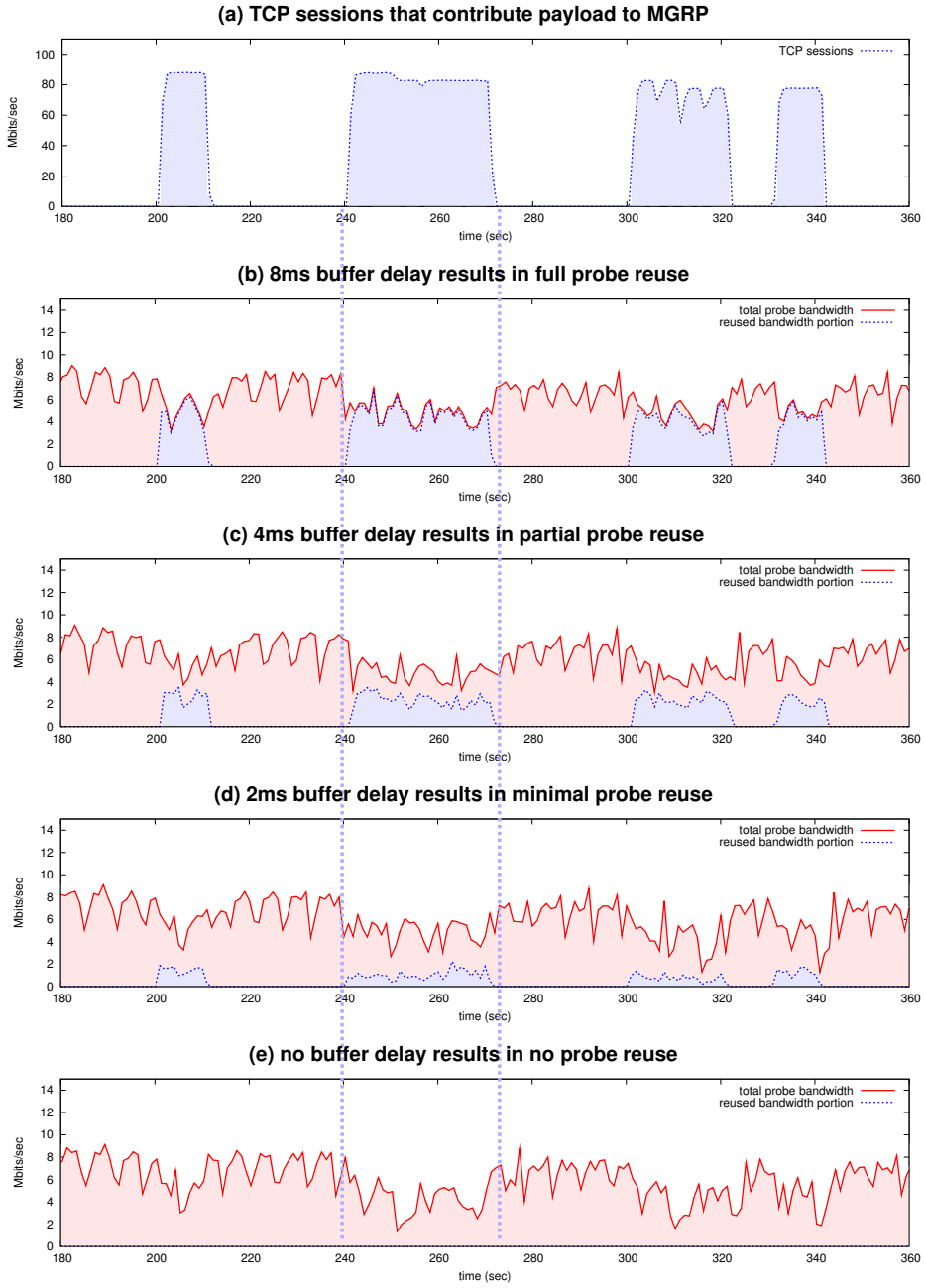


Figure 13: Instantaneous bandwidth (1-sec averages) of pathload probe traffic. Plot (a) uses a different scale on the y-axis and does not show probe traffic. It shows the instantaneous bandwidth of four TCP iperf sessions (for the same time slice) that contribute their packets to MGRP for piggybacking. Each plot (b,c,d,e) corresponds to one experiment run, each with a different payload buffer delay, and each plot shows the portion of the probe traffic that is reused to carry payload from the TCP sessions. *We can see that as the buffer delay decreases the probe reuse diminishes.*

4 Proposed Work

We have proposed the Measurement Manager architecture as a new system for end-to-end Internet measurement that is generic, modular and efficient. Our preliminary implementation and experiments show that the architecture works with real traffic and that it can send probes with configurable overhead. We propose to evaluate the Measurement Manager with respect to real-world clients, develop estimation algorithms and study the impact on transport protocols. We will extend the current prototype to make the Measurement Manager a deployable measurement system. We propose to complete the proposed work in three phases, each of which optimizes a different network operation:

- **Phase 1: Optimizing overlay measurement** In this phase we propose to study how application-layer overlays can be optimized when integrated with the Measurement Manager. 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 inline measurement. We will use MediaNet [6] as a case study.
- **Phase 2: Optimizing file downloads** In this phase we propose to study how file downloading (when we have multiple candidate servers) can be optimized with Measurement Manager's inline measurement. We will study the feasibility of one-way active measurement and we will experiment with multiple estimation algorithms. We will use two case studies to demonstrate how our system optimizes file downloads: (i) using a download manager, for a traditional client-server approach to downloading, and (ii) using BitTorrent [21], for a peer-to-peer approach to downloading.
- **Phase 3: Optimizing TCP** In this phase we propose to study how the Measurement Manager can optimize TCP throughput by switching automatically between a number of specialized congestion control algorithms. We propose to develop an *Adaptive TCP* algorithm that uses the Measurement Manager to measure the network and to decide which is the most appropriate TCP congestion algorithm to use (in terms of optimal throughput) in each networking environment.

In Sections 4.1 and 4.2 we discuss in detail all the aspects of the proposed work in terms of evaluating and extending the Measurement Manager architecture. In Section 4.3 we discuss in more detail the three phases of proposed work and we present an approximate timetable.

4.1 Evaluate the Measurement Manager architecture

Our evaluation methodology will be primarily experimental using real network traffic and real applications. We will use two network environments for our evaluation. First, we will use a controlled network testbed such as Emulab [30] to validate all components of our system and study each tradeoff separately. Second, we will use an Internet testbed such as PlanetLab [32] to evaluate the utility of our system in a realistic Internet environment and to study scalability issues. In what follows, we describe the applications we plan to use for these experiments, the tradeoffs and design choices we wish to measure, and a rough sketch of the experiments we plan to run.

4.1.1 Real-world Benchmarks

We will use real-world applications and transport protocols to demonstrate the utility of our system and to study engineering tradeoffs. Specifically, we will use the application-layer overlay MediaNet [6], which is a media streaming overlay, the peer-to-peer application BitTorrent [21], which is a file distribution system, a download manager, and we will modify TCP to query the *Estimator* for measurements it needs.

Case Study: MediaNet As we discussed in in Section 2.1.1, MediaNet [6] is an application layer overlay that forwards media streams along a fixed network of overlay nodes. MediaNet uses passive measurements of TCP throughput to keep track of the lower bound of available bandwidth on each of its overlay paths and uses that information to forward the media traffic. By using the Measurement Manager, MediaNet can enhance its operation in multiple ways:

- **Periodic measurement of active paths** For an active overlay path (one where media traffic is forwarded) MediaNet can use the Measurement Manager to get periodic measurements of available bandwidth, capacity and loss rate with minimal overhead (since MGRP can reuse most of the probes). That way MediaNet can detect immediately failures in the underlying network (abrupt changes in capacity and loss rate) or detect when the path is getting congested (available bandwidth is consistently lower) and change to a better path.
- **Fast measurement of inactive paths** When MediaNet needs to change to a better path it can either use an inactive overlay path or create a new path. In both cases MediaNet needs to quickly determine how good the path is in terms of available bandwidth and capacity (for example, between two candidate paths with the same available bandwidth the one with larger capacity wins) and can afford to trade overhead for speed. By using the Measurement Manager, MediaNet only needs to specify this tradeoff and internally the *Estimator* picks the algorithm with the fastest convergence time (i.e., time it takes the algorithm to produce an estimate).

Case Study: BitTorrent BitTorrent [21] 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, which can yield suboptimal download times [33]. By using the Measurement Manager, a BitTorrent peer can enhance its operation in multiple ways:

- **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.
- **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).

- **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 results in suboptimal performance.

For our experiments we propose to modify Transmission [34], a BitTorrent client written in C, to use the Measurement Manager for the network measurement operations we have outlined above.

Both MediaNet and BitTorrent use TCP in different ways to transport data and need to constantly measure the network, thus they are well suited for evaluating the Measurement Manager. 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. In the experiments that we describe in the following sections, we propose to use these two real-world applications to evaluate the benefits and tradeoffs of the Measurement Manager architecture in terms of estimation algorithms and transport protocols.

Case Study: Download managers For a more traditional approach to file downloading, we propose to integrate the Measurement Manager with a download manager, such as aria2 [35], to demonstrate how the Measurement Manager can be used in practice and to evaluate the benefits that it provides. 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 and one way that they use is to perform segmented downloading from multiple mirror locations. We will 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.

- **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 our work will be to implement a version of TCP (similar to TCP Nice [9]) that sends only when there are probes to piggyback on.

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

Case Study: TCP In addition to providing payload for piggybacking, TCP may also use the *Estimator* for measurements as we discussed in Section 2.1.2. We propose to modify existing TCP implementations incrementally.

- **RTT estimates** First we will modify TCP to query the *Estimator* for RTT estimates that TCP uses for the estimation of the retransmission timeout (RTO).
- **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).
- **Special measurements** In a similar way we envision using the *Estimator* for more substantial measurements that now are performed in TCP. We propose to select one of the TCP varieties that need special measurements, such as TCP Westwood [8] (available bandwidth), TCP Vegas [7] (delay) or TCP Veno [10] (wireless), and integrate it with the Measurement Manager.

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). We propose to 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. *We propose to explore how 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.*

4.1.2 *Estimator* API

The Measurement Manager presents us with the opportunity to re-examine how clients interact with measurement tools. It is not clear how useful current standalone tools are and how they are used by clients that need measurement results. We propose to formulate a measurement API (the *Estimator* API) that caters better to the needs of clients. We need to

reconcile two contradictory objectives where the API needs to be abstract enough, to give the *Estimator* enough latitude to pick the best algorithm, but also not too abstract so that clients will find the estimation service useful. We have alluded to API issues and approaches in Section 2.1 but we have not yet implemented any of them in our system.

Picking the best algorithm *We propose to formulate the Estimator API so that it allows measurement clients to specify what matters to them most when asking for a measurement (timeliness, accuracy, low overhead).* The *Estimator* then selects an appropriate estimation algorithm that meets the minimum requirements of the client (to be useful) but that does not overly exceed the client requirements (to be efficient). For example, a BitTorrent client that can connect to tens of peers may ask for a rough estimate of the available bandwidth on each path so that it can prune quickly the bad paths and then examine the remaining paths more carefully. Using a highly accurate but slow algorithm is an overkill in this case and does not serve the needs of the client. Selecting the best algorithm is not trivial. The *Estimator* must take into consideration the tradeoffs of each algorithm with respect to timeliness, accuracy and overhead. In later sections we will discuss how we will explore these tradeoffs in the measurement process, that will in turn be used by the *Estimator* to make informed decisions when selecting algorithms.

Quality Measure Current tools return only the value of the measured property to the clients but not all measurements are equal. Clients need a measure to assess the quality of the estimation algorithm, in terms of its accuracy, overhead, timeliness, etc. This measure is affected when changing the way that probes are handled (such as delaying probes and reusing probes). *We propose to develop a "quality measure" for algorithms that can be reported back to clients of the Estimator.* This measure will likely take into account the time it takes for the algorithm to converge, the required bandwidth (i.e., the probe overhead), the portion of the probes that were reused and the actual value and variability of the property under measure. Such a measure will not only be useful to compare different versions of the same algorithm (using different parameters), but also to compare between different algorithms. This is important when the *Estimator* needs to pick the algorithm best suited for a specific measurement, as we discussed previously.

Caching measurements Aside from performing measurements, the *Estimator* can also act as a measurement cache. When clients send measurement queries, the *Estimator* may not perform any measurement at all but return the result from its cache. The decision whether to use a cached entry depends on the requirements of the client; the cached entry must satisfy the quality constraints of the query (age and accuracy). Clients can use such estimates as quick hints to guide more elaborate measurements. For example, TCP sessions can get a better estimate of the RTT when they start or after long stretches of inactivity. Overlays can use cached bandwidth estimates to quickly narrow down the paths that they need to examine further. In the BitTorrent example earlier (section 4.1.1) cached estimates of capacity and available bandwidth may be enough to determine the worst peers and to prune the number of peers that need to be measurement in more detail.

We propose to add caching to the Estimator and study the effect of cached values on the measurement clients. The challenge is to find appropriate caching strategies, since every measurement is different and is relevant for very different timescales that range from

minutes or hours (capacity) to seconds (available bandwidth, RTT). In addition, the caching strategy does not only depend on the measurement but also on how a client uses it; older measurements can be useful as hints but are useless to a client that needs to react fast.

4.1.3 Adaptive Estimation Algorithms

The Measurement Manager uses MGRP to optimize the process of sending probes in two ways: (i) by piggybacking transport payload on empty probes, and (ii) by sending the probes from the kernel (and not from user-space). Since current estimation algorithms are not designed to send their probes through MGRP, they cannot take full advantage of the benefits of the Measurement Manager architecture. We propose to study the new optimizations and tradeoffs that algorithm designers have at their disposal when using MGRP to conduct network measurement.

Delaying probes One way that MGRP can use to increase probe reuse is to have probes wait for transport payload. As we discussed in Section 3.2, estimators typically send out probes in groups where it is important to send the probes within the group at precise intervals. However, the absolute time that the first probe goes out may not be crucial to the estimation algorithm and may tolerate some delay. The longer the delay the more probes can be reused, thus reducing probe overhead. On the other hand, the longer the delay the longer it takes for the estimation algorithm to converge (since it may end up sending fewer probes). The algorithm needs to compensate for the additional delay and cooperate with MGRP in setting constraints so that there is no impact on the algorithm's accuracy. We propose to run a set of experiments to study the effect of delaying probes on algorithm accuracy and convergence time. These experiments will be similar to the experiments in Section 3.3 with the difference that we are delaying probes and not transport payload. *These experiments will help us understand how much delay estimation algorithms can tolerate and how much delay is needed for a meaningful probe reuse ratio.*

Impact of probe reuse One way that MGRP modifies the probe sending process is to reuse all or part of the padding in the probes. This requires some adjustment to the estimation algorithms, since part of the traffic (originating from the measurement host) is not "seen" by the measurement process and must be accounted for when reporting estimates, especially when measuring available bandwidth. We saw part of this effect in the experiment in Figure 10(c). *We will modify our algorithms to use the feedback from MGRP regarding the portion of the probes that was reused and we will experimentally verify that estimates remain the same, regardless of the probe reuse ratio.*

Another way that MGRP modifies the probe sending process is that packet trains are not interleaved with traffic from the originating measurement host. This is due to the fact that MGRP generates the probes in the kernel and tries to generate accurate inter-packet gaps. This in effect renders some traffic coming from the measurement host "hidden" from the packet train probe and may result in higher estimates of available traffic, as seen in the experiment in Figure 10(b). *We will study how estimation algorithms can compensate for traffic coming from the measurement host by using feedback returned from MGRP.*

In-kernel probe generation In the Measurement Manager architecture probes are scheduled by the estimators, either in user-space or kernel-space but are always generated in the kernel by MGRP. This provides better accuracy and timing for individual probes which becomes increasingly important as the network speeds to measure reach Gbits/sec [36]. We propose to run experiments to study the benefits of in-kernel probing and understand whether the Measurement Manager's approach to probe generation is important for high speed networking environments. *Our experiments will compare how existing algorithms improve when measuring high speed network paths (1Gbits/sec) and how that improvement correlates to the increased accuracy when sending individual probes.*

4.1.4 Optimizing Piggybacking

MGRP uses payload contributed from transport protocols to lower the overhead incurred by probes. Our preliminary implementation uses TCP as its main source of payload packets. In Section 3.2.1 we described how we modified TCP in the Linux kernel to contribute packets to MGRP. Our changes are minimal and, other than sending its packets to MGRP, the TCP algorithm is oblivious to any probe reuse or tunneling over MGRP. We propose to study in detail how TCP is affected by the transparent tunneling and the reuse of its packets. We also propose to explore ways that TCP and MGRP can cooperate so that TCP can help MGRP reuse more probes without this reuse interfering with the TCP algorithm.

Buffering TCP payload We saw in Section 4.1.3 that one way to increase the probe reuse in MGRP is to delay the probes. Another way is to buffer the TCP payload for a configurable delay ($2 - 15msec$), as we saw in Section 3.2 and in Figure 6. In our preliminary implementation this happens transparently to TCP and results in a longer effective RTT. We propose to extend our experiments in Section 3.3 to study the tradeoff between TCP throughput and probe reuse. The more we can delay TCP packets the higher the probe reuse but also the longer the effective RTT that TCP experiences. Our preliminary experiments (see Figure 12) indicate that a payload buffer with a small delay ($6 - 8msec$) provides a good reuse ratio in that specific case. *Our new experiments will help us generalize our results and determine a delay threshold that is applicable to a broader range of situations.*

Modifying micro-gaps between TCP packets When MGRP uses TCP payload to fill probe padding it alters the original gaps of the TCP packets. For example, when the congestion window opens, Linux TCP sends a burst of TCP packets, effectively sending packets back to back. But if MGRP has a packet train to send with non-zero gaps between probes, then it slows down the burst to conform to the shape of the packet train. This looks as if MGRP is interfering with the TCP self-clocking mechanism. However, our preliminary experiments (see Section 3.3.3 and Figure 11) indicate that probe reuse does not affect negatively TCP throughput; on the contrary, probe reuse increases TCP throughput since TCP packets compete with fewer probes. This can be attributed also to the fact that in our preliminary experiments the gaps between probes in the pathload packet trains are in the sub-millisecond range ($80\mu sec$) and the number of TCP packets needed for full piggybacking (and hence reshaped) is small (below 6%) relative to the total TCP packets sent.

We propose to run experiments to determine the maximum packet gap distortion that TCP can tolerate without affecting its performance. Work on TCP pacing [37, 38] indicates that a small gap between back-to-back TCP packets may actually be beneficial. By evenly spacing TCP packets ("pacing") when the congestion window opens, TCP can avoid harmful interactions with router queues. We will run experiments using packet trains with increasing gaps between probes and study how TCP's throughput is affected. We expect that the effect on TCP will be negligible for packet gap values much smaller than the RTT, which characterizes most packet train algorithms. We need to determine the upper bound of the packet gap (relative to RTT) that TCP can tolerate when MGRP reuses its payload. This means that each TCP packet shared with MGRP is marked with a deadline by which MGRP must either reuse the TCP payload or send it out without a probe.

Segmenting TCP packets When reusing TCP payload, MGRP not only alters the packet gap of TCP but also changes the size of the TCP packets. This is necessary because the MTU-sized TCP packets (which TCP generates) do not fit in the probe padding. Probe packet sizes are typically smaller than the MTU and probes contain additional headers. To avoid segmenting every TCP packet, MGRP uses the Generic Segmentation Offload [29] mechanism in Linux where TCP, instead of segmenting packets itself, generates large packets (larger than the MTU) and offloads segmentation to lower layers (see Section 3.2.1). However, when probes are small ($< 500bytes$) MGRP has to segment the TCP payload into small chunks, which increases the TCP overhead and the number of TCP packets. The increased overhead is not an issue, since the TCP payload is free-riding on probe packets, but the increased number of small TCP packets may increase the chances of a loss, which adversely affects TCP. *We propose to run experiments to determine a lower bound on the segment size that MGRP can use when segmenting TCP payload.*

4.1.5 Scalability issues

So far we have proposed many small-scale experiments to investigate the tradeoffs presented by the Measurement Manager. These small-scale experiments enable us to address and tune specific parameters of our system but do not address larger issues such as scalability and overall efficiency in a real and complex environment. *For that reason we propose to run a few large scale experiments ($> 100nodes$).*

PlanetLab [32] is an ideal platform for such an experiment but since we use a modified Linux kernel, we cannot run our implementation directly on PlanetLab nodes. Instead, we propose to use a hybrid networking environment. Nodes using the Measurement Manager (which require a modified kernel) run on Emulab, the Computer Science Department and possibly on a commercial platform such as Amazon's Elastic Compute Cloud [1]. These full-control nodes will host the end-nodes of our experiment (such as overlay nodes of MediaNet, download peers of BitTorrent, or our modified TCP endpoints). Traffic between them will be routed through PlanetLab using various IP tunneling configurations (cross-state, cross-country and world-wide paths) to experience realistic Internet conditions.

[1] http://www.amazon.com/ec2

4.2 Extend the Implementation

We propose to extend the Measurement Manager implementation in the two ways: (i) by enabling one-way active probing to make it more deployable, (ii) by including a larger set of estimation algorithms to make it more complete.

4.2.1 Enable one-way active probing

We propose to extend the Measurement Manager to enable one-way active probing from both the receiver and the sender. Such an extension would make the architecture we have described in this proposal more deployable. One of the issues with the current implementation of the Measurement Manager is that it requires the *Estimator* components at both ends of a network path to be coordinated and to implement the same set of estimation algorithms. For example, in our experiment in Figure 8 the pathload algorithm runs at both endpoints with the sender pathload sending probes and the receiver pathload calculating the estimates. This mode of operation is typical of active measurement tools but has some shortcomings: (i) it is difficult to deploy since both sides need to agree every time there is a new algorithm or the algorithm changes, (ii) it requires out of band communication when the estimates are needed at the sender (as was the case in step 10 of our step-by-step example in Figure 4).

One way to achieve one-way active probing in the Measurement Manager architecture is to make the *Estimator* on every node independent of any other *Estimator*. This requires us to add functionality to MGRP so that each peer can coordinate with its remote peer to perform probing and collect the raw measurement results. While this increases the complexity of MGRP, it has the distinct benefit that nodes that wish to perform measurement need only to agree on the MGRP protocol. Each node is free to implement a different *Estimator* that can be custom fitted to its measurement needs. MGRP is less likely to change than the *Estimator* and thus a better candidate for standardization.

We propose to extend MGRP in two ways. First, we will add to MGRP the capability to ask from the remote MGRP for the timestamps of the probes it has sent. This enables an *Estimator* to measure the outgoing path by asking MGRP to send the probes and collect the remote timestamps. This creates reverse traffic that MGRP can also piggyback on the reverse probes. Second, we will add to MGRP the capability to instruct a remote MGRP to send back probes. This enables an *Estimator* to measure the incoming path by asking MGRP to generate incoming probes.

4.2.2 Populate the *Estimator* with algorithms

The preliminary implementation of the *Estimator* implements the Pathload [13] algorithm that estimates available bandwidth. To become practical and useful, the Measurement Manager needs to provide a larger set of estimators. Properties that are useful to estimate range from basic path properties such as RTT, delay, jitter and loss rate (in both forward and reverse paths) to more complex properties such as available bandwidth, capacity, location of bottleneck links and detection of paths that share congestion. A good starting point is to look at existing active measurement tools and integrate their algorithms into the *Estimator*.

Adapting an existing tool to the Measurement Manager involves three steps. First, we

modify the tool to use MGRP to send probes. The tool still operates as a standalone tool, which typically means that there are two instances running: one at the sender and one at the receiver. In the second step we integrate sender and receiver into one instance that uses MGRP to perform one-way probing but that still still runs in standalone mode. Finally, we extract the algorithm from the second step and incorporate the algorithm into the *Estimator*. The major challenges of this process is to make a two-sided tool work as one-sided and to verify that the modified tool produces the same results as the original (even in the presence of reused probes).

We propose to add a number of algorithms in the Estimator. The *Estimator* already has an algorithm for available bandwidth. We intend to implement an additional algorithm for available bandwidth based on pathchirp [14], which has lower overhead. We intend to implement one algorithm for path capacity based on pathrate [15], one for locating bottlenecks based on pathneck [16], and algorithms to estimate RTT and loss rate.

4.3 Phases and Timetable

The proposed work can be split into three phases with each phase corresponding roughly to one conference paper. What follows below is our view of how the proposed work should be divided, with approximate estimates about the time it will take to complete each phase. We estimate that in total the three phases will take approximately $60weeks$, which combined with $3months$ to write the dissertation brings the estimated time to complete the proposed work to about $18months$.

4.3.1 Phase 1: Optimizing overlay measurement

In this phase we will focus on turning the Measurement Manager into a deployable measurement system for application-layer overlays. The paper that will come out of this phase will be the first to introduce the Measurement Manager architecture as laid out in this research proposal. We will present our architecture, our implementation and include one case study to demonstrate how overlays can benefit from the Measurement Manager, as we outlined in Section 2.1.1. Our proposed timeline for this phase appears in Table 1.

Phase 1: Optimizing overlay measurement		**15 weeks**
Design	Architecture	completed
Implementation	MGRP Protocol	completed
	Available Bandwidth (pathload)	completed
	Caching	1 week
	Delay probes instead of TCP payload	1 week
	Capacity (pathrate)	2 weeks
	Integrate MediaNet	3 weeks
Evaluation	Understand the Impact on TCP	2 weeks
	Emulab experiments	1 week
	Hybrid: Emulab + PlanetLab	3 weeks
Paper writing		2 weeks

Table 1: Proposed timeline for Phase 1: Optimizing overlay measurement

Our methodology is as follows. We will first integrate the application-layer overlay MediaNet with the Measurement Manager (see section 4.1.1). MediaNet uses TCP so we will demonstrate that the operation of TCP is not affected by the MGRP piggybacking (we will extend our experiments from section 3.3.3). In addition to the available bandwidth algorithm, we will implement a capacity estimation algorithm based on pathrate [15]. MediaNet will use capacity estimates for its initial configuration and available bandwidth estimates for selecting which overlay paths to activate during its maintenance phase. Our objective is to show that the Measurement Manager fulfills the measurement needs of overlays more efficiently than what overlays could do on their own. The more overlays are using the Measurement Manager on the same node, the higher the benefit.

4.3.2 Phase 2: Optimizing file downloads

In this phase we will concentrate on estimation algorithms and deployability of the Measurement Manager system. We will use two case studies to demonstrate how our system optimizes file downloads: (i) using a download manager, for a traditional client-server approach to downloading, and (ii) using BitTorrent, for a peer-to-peer approach to downloading. We will integrate these two applications with the Measurement Manager as outlined in Section 4.1.1 and we will examine how they interact with estimation algorithms about available bandwidth, capacity and bottleneck detection. Contrary to the overlay case, in file downloads the two peers are not typically under the control of the same entity. For that reason we will extend the Measurement Manager to enable one-way active probing (see section 4.2.1). Our proposed timeline for this phase appears in Table 2.

Phase 2: Optimizing file downloads		**23 weeks**
Estimator API	Quality measure	2 weeks
	Picking the best algorithm	3 weeks
Implementation	One-way active probing	3 weeks
	Integrate download manager	2 weeks
	Integrate BitTorrent	3 weeks
	Detect bottlenecks (pathneck)	2 weeks
	TCP (that sends only on probes)	1 week
Evaluation	Emulab experiments	1 week
	Hybrid: Emulab + PlanetLab	3 weeks
Paper writing		3 weeks

Table 2: Proposed timeline for Phase 2: Optimizing file downloads

For measurements we will use three algorithms: one for available bandwidth (already implemented), one for capacity (from Phase 1) and we will implement one algorithm for locating bottlenecks (based on pathneck [16]), as we described in section 4.2.2. Our focus during this this phase will be estimation algorithms. We will study how algorithms can be integrated with the Measurement Manager (see Section 4.1.3), what the *Estimator* API should look like, how they can provide a *quality measure* for their estimates (see Section 4.1.2) and study the tradeoffs between delaying probes and probe reuse.

4.3.3 Phase 3: Optimizing TCP

In this phase we will focus on TCP as a client of the Measurement Manager. We will modify TCP incrementally to use the *Estimator* to get RTT estimates, loss indications and a number of specialized measurements as we described in section 4.1.1. Our goal will be to develop an *Adaptive TCP* algorithm that uses the most appropriate congestion control algorithm for every network environment based on measurements from the *Estimator*.

Phase 3: Optimizing TCP		**22 weeks**
Design	Adaptive TCP	3 weeks
Implementation	Replace RTT estimation	2 weeks
	Replace loss detection	2 weeks
	Integrate TCP Vegas (delay)	2 weeks
	Integrate TCP Westwood (available bandwidth)	2 weeks
	Integrate TCP Veno (packet loss distinction)	2 weeks
Experiments	Emulab experiments	3 weeks
	Hybrid: Emulab + PlanetLab	3 weeks
Paper writing		3 weeks

Table 3: Proposed timeline for Phase 3: Optimizing TCP

5 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.
- The way to piggyback probes on transport payload is an integral part of the Measurement Manager architecture. Most existing approaches to inline measurement resort instead to clever trickery to embed their probes into TCP streams.
- The Measurement Manger is independent of applications, transport protocols and measurement algorithms. Measurement tools use a small set of probing primitives to build their estimation algorithm.
- The Measurement Manager simplifies deployment by requiring only the probing component (the MGRP protocol) to be deployed.

5.1 Network measurement tools

The value of active measurement tools for the Measurement Manager architecture is twofold: (i) they shape the *Estimator* API by demonstrating what kind of measurement information applications need, and (ii) they shape the Probing API primitives of MGRP by demonstrating the types of probing structures needed. The number of available network measurement tools is quite large [39, 40]. We list here a sample of the tools that we studied in detail while designing the Measurement Manager architecture.

- **Pathchar [11, 12]** uses variable packet size probing to estimate path characteristics.
- **Pathload [13]** uses packet trains with uniform spacing to estimate available bandwidth and has been covered in detail in Section 3.1.1.
- **Pathchirp [14]** uses packet trains with variable packet spacing to estimate available bandwidth. It uses fewer probes than pathload but is not as accurate [22] (also verified by our experiments).
- **Pathrate [15]** uses packet pair probing to estimate path capacity.
- **Pathneck [16]** uses recursive packet trains (TTL-limited probes surrounded by normal packets) to locate bottleneck links along a network path.

5.2 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.

- **Periscope [17]** 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 [18]** 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 **[19]** 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 [20]** 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 [41]** 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 measurement 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 [33]** 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 [42]** is and information plane for networked systems that is deployed on PlanetLab [32]. 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 [1]** is a shared measurement infrastructure that sits between 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 [5]**, 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.

5.3 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 [43]** is a sender-side only extension to TCP that estimates the bottleneck capacity of a network path using a portion of the TCP packets as probes. TCP Probe uses the CapProbe [44] 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 [45]. 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 [46]** is another example of a piggybacking technique similar to TCP Probe. In this case, the CapProbe [44] algorithm has been integrated with the TFRC protocol [47] 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 [48] project, which provides a toolkit for the management of overlay networks with mobile users.

- **ImTCP [49]** is an inline measurement mechanism for actively measuring available bandwidth using TCP packets. ImTCP combines a measurement algorithm for available bandwidth [50] 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 [51].

- **ICIM [52]** 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 [53], which has a negative effect on network measurements.

- **TCP Sidecar [54]** 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 [55] is a measurement algorithm based on TCP Sidecar that discovers router-level Internet topology by piggybacking on TCP streams.

- **Sting [56]** 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 [57]** 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 **[58]** 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.

5.4 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 [8]** uses end-to-end bandwidth estimation to set the values of the congestion window *cwin* and the slow-start threshold *sshthres*.

- **TCP Vegas [7]** 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 [10]** 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 [59] to select *adaptively* between two estimators based on the predominant cause of packet loss.

Appendix A: MGRP APIs

MGRP exports two APIs: (i) the Probe API, intended for the *Estimator* to send probes, and (ii) the Payload API intended for transport protocols to contribute transport payload that MGRP uses for piggybacking. The two APIs appear in Figure 14.

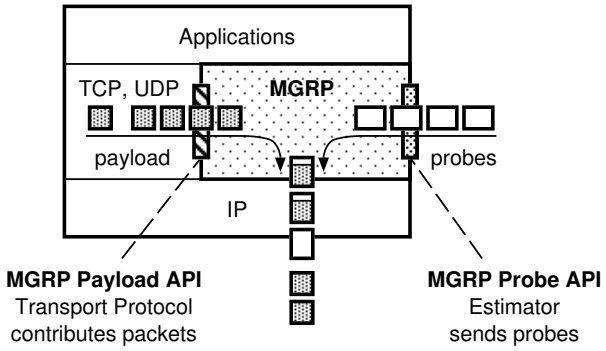


Figure 14: MGRP exports two APIs, the Probe API to send probes and the Payload API to contribute transport payload.

A.1 Probe API

In section 2.2.2 we identified a number of building blocks for sending probes: sending single probes, packet pairs, packet trains and groups of special ICMP or TTL-limited probes. We can support these building blocks with an API that exports just a few primitives.

MGRP uses the notion of *probe transactions* to implement all types of probing. Each transaction is comprised of a group of probes that need to be sent out according to a particular sending pattern. This pattern is specified by the relative gaps between packets. The defining characteristic of each transaction is that the first probe of the group may be delayed (according to constraints) but once it goes, the whole group has to go. Estimators need to break down their probe patterns into probe transactions before sending them to MGRP.

The use of transactions implies that packets enter MGRP in groups. But since MGRP is a Layer 4 protocol we wish to preserve the simplicity of the single-packet API that characterizes all protocol layers. To accomplish that, we introduce the notion of *barriers*. Whenever an estimator wishes to issue a probe transaction through the Probe API, it first *raises the barrier* in MGRP, then it sends one by one all the packet probes, and finally *lowers the barrier*. There is no need for the estimator to generate proper gaps between packets; MGRP does that after the barrier is lowered (no probes are sent before that). The estimator just needs to describe the properties of each packet. This information tells MGRP how much padding each probe has, whether it is a special packet (ICMP or TTL-limited) and what is the time gap from the previous probe.

The MGRP component is implemented as a kernel Layer 4 protocol, but that does not mean that the whole Measurement Manager needs to reside in the kernel. In fact, the *Estimator* component has a place both in and out of the kernel (so that it can be used by both transport protocols and applications). For that reason MGRP exports its Probe API in two

ways: through function calls in the kernel and through socket operations. The additional benefit of having a socket Probe API for MGRP is that it helps with testing and enables fast prototyping. All it takes for a user-space client to use MGRP is to open an MGRP socket and use the barrier method to send probes, as shown in Figure 15.

```
/* send a packet train using an MGRP probe transaction */

/* 10 packets of size 1000bytes each with 1ms gap */
int probe_size = 1000;
int probe_data = 20;
int probe_gap  = 1000
int probe_pad  = probe_size - probe_data;

char probe[probe_size];

/* 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: raise barrier */
probe->barrier = 1;
probe->pad = probe_pad;
probe->gap = 0
sendmsg(sock, &msg, 0);

/* pass probes 1..9 */
for (i = 1; i < 9) {
    probe->barrier = 1;
    probe->pad = probe_pad;
    probe->gap = probe_gap;
    sendmsg(sock, &msg, 0)
}

/* pass last probe: lower the barrier */
probe->barrier = 0;
probe->pad = probe_pad;
probe->gap = probe_gap;
sendmsg(sock, &msg, 0);

/* MGRP sends the packet train */
```

Figure 15: Code sample to send a packet train using MGRP from user-space. MGRP introduces the notion of *barriers* to enable applications to schedule precise probing patterns using the familiar single-packet socket API. Before sending a probe pattern, an application raises the barrier, sends the probes using `sendmsg()` and lowers the barrier. MGRP sends the probe pattern the moment the barrier is lowered. Applications use ancillary control data to pass information about the inter-packet spacing and the padding portion of the probes.

Once the barrier is lowered, MGRP has all the information it needs to send out the probes. At this point, MGRP tries to reuse as much padding as possible without violating the per-transaction constraints. The estimator passes these constraints to MGRP when the barrier is lowered and they determine how long the first probe can wait, whether any probe can be reused and what to do if the deadline arrives and no probes were reused. For example, an estimator that prefers almost passive operation will instruct MGRP to drop the whole transaction if a high percentage of probes are not reused. In contrast, an estimator that needs results immediately will tolerate very small delay and will not care about probe reuse. At the end, MGRP completes the transaction either by dropping it (if the target reuse ratio was not met) or by sending out the probes as a group (possibly filled with payload) with the proper gap sequence.

Sending probes is only half of the story. All estimators need to collect raw probe data, such as timestamps and loss events, so their algorithms can produce their estimates. Estimators collect this information through MGRP because probes are not sent immediately

and because algorithms may need to be adjusted for the portion of the probes that was reused. Toward that goal, on the sender side, MGRP returns to the estimator the actual send-timestamps of the probes and the portion of their padding that was reused. On the receiver side, MGRP passes the send-timestamp and receive-timestamp along with the actual probe to the receiver-side estimator.

A.2 Payload API

The Payload API is used by transport protocols to contribute payload that MGRP uses for piggybacking. Transport protocols treat MGRP like the IP layer; they push packets into MGRP and expect them to pop up at the remote endpoint. What is different is that in the process the transport packets may piggyback on empty probes. The need for a Payload API stems from the need for coordination between transport protocols and MGRP so that this piggybacking is meaningful and that it does not adversely affect the operation of the transport protocol. The Payload API is an in-kernel API so that any transport protocol can use it.

For piggybacking to be meaningful MGRP needs to reuse as many probes as possible. Due to timing issues, piggybacking becomes impractical when both packets and probes pass through MGRP without delay. We saw in Section A.1 that one way to increase reuse is to delay the probes. Alternately, MGRP can buffer the transport packets in a *payload queue* and as probes pass they pick up payload from the queue for piggybacking, as seen in Figure 6. One idea we have implemented is to add a constant delay of a few milliseconds that provides MGRP with a look-ahead payload buffer and appears to the transport protocol as a slightly increased RTT. Transport protocols can coordinate with MGRP through the Payload API to set constraints on this buffering and mitigate the effect on the operation of the protocol.

Another issue with piggybacking has to do with how well transport packets fit into probe padding. Transport protocols typically segment their packets to fit the MTU, which means that most transport packets are too big to fit into probe padding (even if probes are as big as the MTU, there is header overhead). This leads to low probe reuse even though there are many donated packets. One approach is to fragment transport packets at the MGRP layer (much like IP fragmentation) but many transport protocols tend to perform their own packetization because fragmentation hurts their performance. The approach that MGRP follows is to cooperate with the transport protocol to create transport packets of the right size. One way to do this is to ask the transport protocol to defer packetization until MGRP knows the sizes of the payload chunks that it needs. Another way is for the transport protocol to pass large packets to MGRP with instructions how to segment them into smaller packets that are valid by themselves (new header + part of payload). We take the latter approach in our implementation.

References

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] V. Jacobson, "pathchar - a tool to infer characteristics of Internet paths," [Online] Talk at MSRI, April 1997.

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

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

[14] 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.

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

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

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

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

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

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

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

[22] 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.

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

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

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

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

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

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

[29] H. Xu, “Patch: GSO: Generic Segmentation Offload,”
Email on linux-netdev mailing list,
[Online] http://article.gmane.org/gmane.linux.network/37287, June 2006.

[30] “Emulab - Network Emulation Testbed,”
[Online] http://www.emulab.net/.

[31] “iperf: A tool for measuring TCP and UDP bandwidth performance,”
[Online] http://dast.nlanr.net/Projects/Iperf/.

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

[33] 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.

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

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

[36] G. Jin and B. L. Tierney, “System Capability Effects on Algorithms for Network Bandwidth Measurement,” in *Internet Measurement Conference*, 2003.

[37] A. Aggarwal, S. Savage, and T. Anderson, “Understanding the Performance of TCP Pacing,” in *IEEE INFOCOM*, 2000.

[38] D. X. Wei, P. Cao, and S. H. Low, “TCP Pacing Revisited,” in *IEEE INFOCOM*, 2006.

[39] L. Cottrell and SLAC, “Network Monitoring Tools,”
[Online] http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[40] CAIDA, “Performance Measurement Tools Taxonomy,”
[Online] http://www.caida.org/tools/taxonomy/performance.xml.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

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

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

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

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

[59] 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.