TCP RAPID: FROM THEORY TO PRACTICE

Qianwen Yin

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2017

Approved by:

Jasleen Kaur

F. Donelson Smith

Kevin Jeffay

Jay Aikat

Ketan Mayer-Patel

ABSTRACT

QIANWEN YIN: TCP RAPID : From Theory to Practice
(Under the direction of Jasleen Kaur)

Delay and bandwidth-based alternatives to TCP congestion-control have been around for nearly three decades and have seen a recent surge in interest. However, such designs have faced significant resistance in being deployed on a wide-scale across the Internet—this has been mostly due to serious concerns about noise in delay measurements, pacing inter-packet gaps, and required changes to the standard TCP stack. With the advent of high-speed networking, some of these concerns become even more significant.

This thesis considers Rapid, a recent proposal for ultra-high speed congestion control, which perhaps stretches each of these challenges to the greatest extent. Rapid adopts a framework of continuous fine-scale bandwidth probing and rate adapting. It requires finely-controlled inter-packet gaps, high-precision timestamping of received packets, and reliance on fine-scale changes in inter-packet gaps. While simulation-based evaluations of Rapid show that it has outstanding performance gains along several important dimensions, these will not translate to the real-world unless the above challenges are addressed.

This thesis identifies the key challenges TCP Rapid faces on real high-speed networks, including deployability in standard protocol stacks, precise inter-packet gap creation, achieving robust bandwidth estimation in the presence of noise, and a stability/adaptability trade-off. A Linux implementation of Rapid is designed and developed after carefully considering each of these challenges. The evaluations on a 10Gbps testbed confirm that the implementation can indeed achieve the claimed performance gains, and that it would not have been possible unless each of the above challenges was addressed.

ACKNOWLEDGEMENTS

I would like to express my special appreciation and thanks to my advisor Professor Dr. Jasleen Kaur, who has been showing considerable diligence, patience and kindness in mentoring me. Prof. Kaur's advice on both research as well as on my career is priceless. I would also like to thank my committee members, Prof. Kevin Jeffay, Prof. F.Donelson Smith, Prof. Jay Aikat, and Prof. Ketan Mayer-Patel for serving as my committee members even out of business. I also want to thank them for letting my defense be an enjoyable moment, and for their brilliant comments and suggestions. I would also give thanks to staff members in the department of computer science, especially the ITS team. Their tremendous efforts in maintaining the testbed makes this thesis possible!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

TCP	Transmission Control Protocol
ACK	Acknowledgement
RTT	Round Trip Time
OWD	One-way Delay
NIC	Network Interface Card
avail-bw	Available Bandwidth
sendgap	Inter-packet Gap intended to be created at the sender
recvgap	Inter-packet Gap observed at the receiver side
cwnd	Congestion window

CHAPTER 1: INTRODUCTION

1.1 Motivation

1.1 Issues with Loss-based TCP Congestion Control

TCP congestion control protocols were originally designed to reduce persistent network congestion, under which TCP flows suffer from massive packet losses and achieve extremely low throughputs. TCP congestion control protocols detect congestion on end-to-end paths and adjust TCP senders' transmission rates to address the congestion; different protocol designs adjust the senders' transmission rates differently. The original congestion control protocol, Tahoe, was designed in 1987; Tahoe and its successors, Reno, NewReno, and SACK, have been widely deployed for over three decades as the mainstream end-to-end congestion control mechanisms for the Internet.

These legacy congestion control protocols maintain a congestion window (*cwnd*) at the senders; this congestion window limits the number of packets that are unacknowledged during transfer to the number that can be sent within one round-trip time (RTT). The congestion control algorithms consist of two phases. In the first phase, after initialization, a TCP flow enters the *slow-start* phase, during which it aggressively doubles *cwnd* every RTT; during this phase, no packet from the past RTT is lost. Once *cwnd* reaches some threshold (SS_THRESH), the flow enters the second phase, the *congestion-avoidance* phase; during this phase, it carefully and slowly acquires more bandwidth. In this second phase, *cwnd* is updated based on the additive-increase and multiplicative-decrease (AIMD) principles: it is increased by one if there are no packet losses detected, and if packet losses are detected, it is reduced by half.

The 1990s and 2000s witnessed a rapid growth of network speeds; speeds increased by several magnitudes, from below 100 Kbps to over 10 Gbps. These faster speeds cause problems for the legacy congestion control methods; after the loss of a single packet, the conservative AIMD mechanism significantly slows TCP flows in order to fully utilize the path [1, 2]. Dozens of new congestion control protocols have recently been proposed;

these new protocols adjust *cwnd* more aggressively in the congestion-avoidance phase [1, 3, 4, 5, 6, 7]. These methods have been shown to significantly improve path utilization on high-speed links with large RTTs [8].

But no matter how fast these protocols increase their *cwnd*, their performances are constrained by their loss-based congestion-control design: they rely on packet loss to infer the presence of network congestion. Packet loss is a binary congestion signal (either a packet gets lost or it does not), and it therefore does not reflect the degree of congestion on the network. Because packet loss offers limited feedback about the network status, protocol designers are more likely to design conservative congestion-control algorithms, rather than aggressive ones; loss-based control mechanisms must significantly reduce transmission rates after losses are detected in order to alleviate network congestion, and must increase transmission rates slowly enough to avoid repeated losses. As a result, loss-based protocols usually overreact to packet losses, wasting abundant network capacity. [2] has shown that even protocols [1, 5] designed for high-speed networks require hundreds of RTTs to regain full path utilization after a single loss on a 1 Gbps path. Packet loss is also a delayed indicator of congestion—loss-based protocols neither perceive congestion nor take action to ameliorate it until the congestion is bad enough to overflow the network buffers. To address congestion before it becomes critical, TCP protocols should use congestion signals other than packet loss.

1.1 Promises of Delay-based and Bandwidth-based Protocols

To address the drawbacks of loss-based congestion control, researchers have been actively looking for novel congestion indicators over the past decades. These efforts have led to two new types of congestion control protocols: delay-based and bandwidth-based protocols.

Delay-based Protocols

Some protocols using delay to signal congestion have been developed: these protocols measure an increase in either the one-way delay (OWD) or the RTT. When paths are congested, packets will be queued, and the queueing delay will increase OWD/RTT measurements. TCP protocols that infer congestion from RTT and/or OWD measurements are called "delay-based" protocols.

Delay measurements have two significant advantages over packet loss as congestion signals. First, delay meansurements enable protocols to handle congestion earlier—when the queue starts building up, before packet loss occurs. Second, unlike the binary packet-loss signal, delay contains much richer information about the congested path. Queueing delay scales with the extent of congestion—more severe congestion leads to longer queueing delays. Depending upon the delay increments measured, delay-based protocols can

adjust their transmission rates by different degrees. These two features help delay-based protocols to reduce packet losses, achieve more stable path utilization, and increase fairness among competing flows [9, 10, 11].

Several delay-based protocols were developed in the 1990s and 2000s, including [9, 12, 13, 10, 14, 11]. During this period, many researchers also began to use delay measurements to enhance the performance of loss-based protocols [6, 15, 7, 16, 17, 18]. In 2015, the protocol TCP TIMELY [19] introduced delay-based protocol design to data centers for the first time.

Bandwidth-based Protocols

Instead of seeking alternative congestion signals, "bandwidth-based" congestion control methods identify congestion by directly probing for the transmission rate and comparing it to the maximum transmission rate that the path affords.[1]

Bandwidth-based protocols often borrow techniques from bandwidth estimators [20, 21, 22, 23] to probe the path for unused bandwidth, which is also known as available bandwidth (avail-bw). Bandwidth-based protocols adapt transmission rates to avail-bw; they are able to quickly identify and converge to changed bandwidth, and they can therefore utilize paths efficiently without causing excessive congestion.

Among existing bandwidth-based protocols (PCP [24], UDT [25], NF-TCP [26], and RAPID [2]), RAPID stands out for its performance, which is remarkably close to optimal on gigabit networks in NS2 simulations. It discovers and adapts to changing avail-bw within 1 to 4 RTTs; it ensures fair throughput among multiple RAPID flows with heterogeneous RTTs; and it has a negligible impact on legacy TCP flows sharing the path.

1.1 Hurdles to Widespread Adoption

Although delay-based and bandwidth-based protocols have demonstrated considerable performance gains over packet-loss protocols, there is significant resistance to their broad deployment across the Internet. Researchers seriously doubt whether these protocols' performance in simulations or controlled testbeds can translate well to real-world settings, especially at ultra-high speeds, due to the following issues.

- **Insufficient Timing Accuracy**

  To measure delay in RTTs or OWDs, accurate timestamping is required. To obtain RTTs, the sender records the arrival time of acknowledgements; to obtain OWD information, the receiver records the

[1] Some protocols follow a router-assisted approach, where the router allocates bandwidth for each flow. We focus on end-to-end designs in this thesis.

packet arrival time and returns the timestamp to the sender in its acknowledgement. To accurately measure changes in delay magnitude, timestamps must be measured at finer granularity than queuing delay measurements. However, the necessary measurement granularity gets smaller as network speeds scale up. For example, on a 10 Mbps link with a frame size of 1500 B, the queuing delay is measured at scales of $\times 1.2\ ms$; an end-host with 1 ms timing granularity ($jiffies$ in Linux) can easily perceive queuing from these delay measurements. However, once the network speed scales up to 1 Gbps, the queuing delay scales down to $\times 12\ \mu s$ —in this scenario, 1 ms is so coarse-grained that the same host will observe no queuing delay until the queue length reaches around 100. On even higher network speeds, of 10 Gbps and 40 Gbps, timestamping with at least $1\ \mu s$ accuracy is required.

Unfortunately, timestamping granularity is implementation-dependent. [27] shows that the majority of commercial servers only provide 10 ms–100 ms granularity — μ-scale resolutions are not available for most servers.

- **Weak Correlation Between Delay and Congestion**

  Although delay-based and bandwidth-based protocols use delay measurements in different ways, all of them rely on the assumption that an increase in RTT or OWD is a strong sign of congestion. However, previous research [28, 29, 30] has shown that delay variations are only weakly correlated to path congestion, mainly because of transient queuing—packets arrive at routers in a very bursty manner [31, 32], and queues are built up and drained off at very short timescales. Because of this burstiness, packets may experience transient queuing even in the absence of congestion, and they may not be queued at all in the presence of congestion. Thus, increases in delay measurements cannot be relied upon to robustly indicate path congestion.

- **Creating Inter-packet Gaps Is Challenging**

  Many protocols that involve bandwidth-estimation processes [14, 24, 25, 26, 2] send packets with finely controlled inter-packet gaps , and avail-bw is estimated by observing how those gaps change over the entire path. Thus, it is crucial to create accurate inter-packet gaps . But on high-speed networks, inter-packet gaps can be at very fine scales; probing on a 10 Gbp path, even with jumbo frames (9000B), requires gaps as small as 7.2 μs. Such fine-scale gap creation is fairly sensitive to fine-scale noise, which imposes considerable challenges to end-hosts.

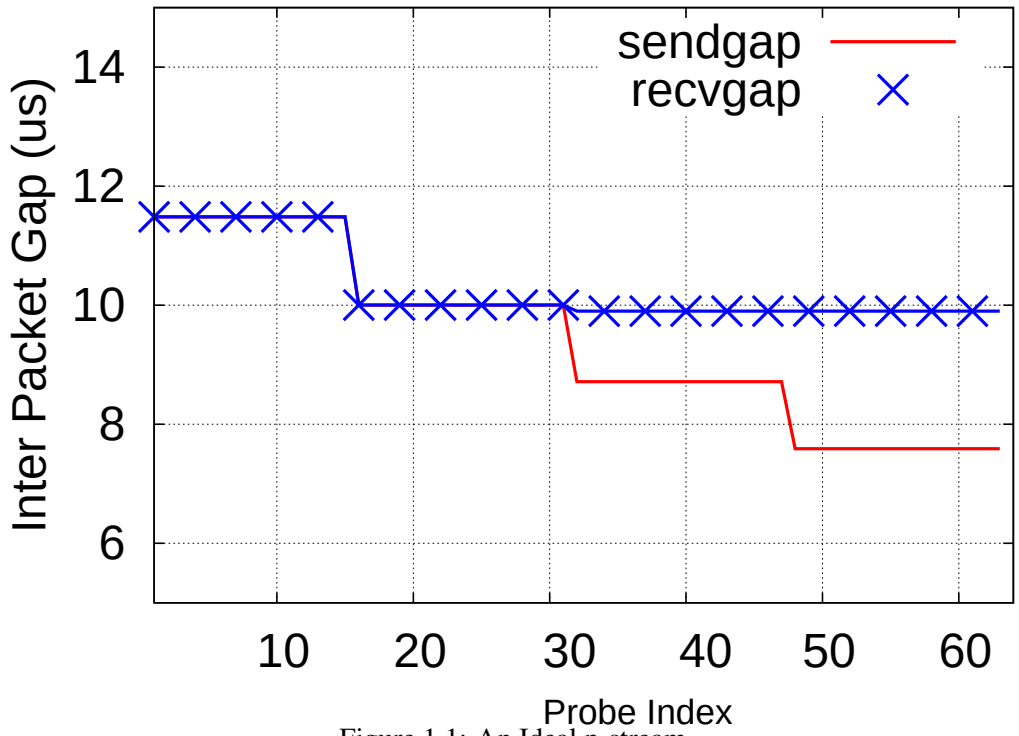


Figure 1.1: An Ideal p-stream
In noise-free networks, the inter-packet gaps at the receiver (recvgaps) are expected to be consistently larger than inter-packet gaps generated at the sending host once the probing rate starts to exceed avail-bw.

- **Incompatibility with Existing Protocol Stack**

  Many protocols [14, 24, 25, 26, 2] also require the creation of time gaps between packets, or "inter-packet gaps." In these protocols, packet transmission is "gap-clocked"—the transmission of each individual packet is determined by the time gap between it and the preceding packet. Unfortunately, this is a complete departure from widely deployed TCP protocol stacks, which couple packet transmission with ACK arrivals ("ack-clocked"). These new protocols inspire strong resistance—why would an operator of production servers get rid of a protocol that has been used for more than three decades in favor of a new protocol?

1.1 RAPID Stretches Challenges to Extreme

1.1 TCP RAPID —a Packet-scale Congestion Control Protocol

TCP RAPID transmits packets in logical groups. Each group, referred to as a p-stream, probes for a set of exponentially increasing rates. The probing rate of the i-th packet R_i is achieved by controlling the

inter-packet gap between R_i and its previous packet, according to $g_i^s = \frac{P}{R_i}$, where P is the packet size and g_i^s is the time gap created by the sender between the ith and i-1th packet. Each p-stream observes the inter-packet gaps when packets arrive at the receiver (denoted as g^r) and compares them with g^s. Bandwidth is estimated by searching for a signature of **persistently increasing queuing delay**: once the probing rate exceeds the avail-bw, inter-packet gaps keep increasing over the path due to persistent congestion. Therefore, avail-bw is the highest probing rate beyond which g^r is consistently larger than g^s. For instance, Fig 1.1 shows the g^s and g^r within a p-stream encountering 2.45 Gbps average cross-traffic on a 10 Gbps path that is free of noise; g^r becomes consistently larger than g^s after the 33rd packet. In this scenario, the $ABest$ is the probing rate of the 32nd packet. Once the avail-bw is computed, RAPID updates the average transmission rate of the subsequent p-stream accordingly.

Mainstream protocols, whether loss-based, delay-based or rate-based, all control congestion by adjusting *cwnd* sizes; the congestion control is reflected by the *cwnd* increment or by the reduction in the next RTT. In RAPID, every packet performs bandwidth probing, and every packet plays a key role in bandwidth estimation. To distinguish TCP RAPID from other congestion control methods, we refer to it as *"packet-scale" congestion control*.

1.1 Challenges in Practice

Perhaps even more than the existing delay-based and bandwidth-based protocols, RAPID encounters each of the above challenges to implementation.

- Delay-based protocols (e.g. Vegas and Fast) usually indicate congestion and react to it using the mean of all delay measurements within one RTT, rather than using every single delay sample. This averaging provides a denoising effect, alleviating the impact of noise caused by transient queuing and timestamping inaccuracy. However, our proposed packet-scale congestion control protocol uses *every* inter-packet gap sample to estimate avail-bw, and accurate arrival timestamping for *every* packet is crucial.

- Most delay-based [9, 11] and rate-based protocols [25] are able to perceive congestion as long as packets experience *any* amount of queuing delay. However, the avail-bw estimation algorithm used by RAPID has a more stringent requirement: it must observe *persistently increasing* queuing delays within a p-stream. This requirement is not affected by transient queuing.

- As mentioned in Chapter 1.1.3, scheduling packet transmissions based on inter-packet gaps is a key challenge on high-speed networks. Other rate-based protocols [24, 25, 26] create inter-packet gaps only intermittently, or use bandwidth probing and estimating merely as an enhancement to loss-based congestion control methods. In contrast, *every* packet transmission in RAPID is "gap-clocked." In other words, gap creation plays a crucial role in RAPID, which requires accurate throughput for the entire connection.

1.1 Goal of this dissertation

In this thesis, we ask the following question: can the challenges faced by delay-based and bandwidth-based protocols be addressed in real-world high-speed networks, enabling them to perform as well as promised? We explore this question using RAPID. If a protocol as demanding as RAPID can be put into use on ultra-high speed networks, that would be a convincing argument for the practical adoption of delay-based and bandwidth-based designs.

The goal of the thesis is thus to demonstrate how well RAPID performs in practice, and to test whether its real-world performance can match its outstanding performance in simulation. We identify challenges that prevent RAPID from delivering on its promises, address each of them with a novel or existing mechanism, and evaluate whether our Linux implementation meets performance claims in a 10 Gbps testbed. We also show that neither the challenges nor the mechanisms that address them are specific to RAPID; other delay-based and bandwidth-based protocols will also benefit from adopting those mechanisms.

1.2 Challenges of RAPID in Practice

In this section, we identify four challenges RAPID faces in implementations on high-speed networks. For each challenge, we briefly describe the requirement imposed by RAPID, the obstacles to achieving that requirement, and the way this thesis addresses the challenge.

1.2 Creating Accurate Inter-packet Gaps

1.2 RAPID Requirement

RAPID requires TCP senders to send out packets with *high-precision, fine-grained* inter-packet gaps, because its bandwidth estimation logic depends on the intended probing rates being precisely reflected in the

inter-packet gaps created at the sender. However, this kind of accuracy in gap creation is highly demanding. When probing high-speed networks, the required gaps are at μs timescales. For instance, in order to probe a 10 Gbps network, even with jumbo-sized frames, the spacing must be as small as a few μs. A gap inaccuracy of even 1μs can lead to nearly 100% error in the intended probing rate.

1.2 Challenges

Creating such fine-scale, high-precision inter-packet gaps is fairly challenging for today's end-hosts for two main reasons. First, modern operating systems are interrupt-driven; the gap-creation process can easily get interrupted, lose control of the CPU, and not be scheduled again before the gap time elapses. The resultant send gaps are unpredictable and imprecise. Second, before packets are transmitted to the outbound link, they can buffer at several places: (i) in the socket send buffer, when they are handed by the application to the kernel and are waiting to be processed by the TCP/IP protcol stack; (ii) in the interface queue associated with each outbound NIC; and (iii) in the NIC internal buffer when they are being transmitted out. Buffering puts packets back to back in the queue, closing up the inter-packet gaps.

1.2 Goal

The first goal of this thesis is to consider and evaluate techniques that create inter-packet gaps in order to find a mechanism that both ensures high-precision gaps and is resistant to interrupt and end-host buffering, and to tailor these techniques for RAPID.

1.2 Noise Removal to Achieve Accurate ABest

1.2 RAPID Requirement

Bandwidth estimation is the key component to RAPID, which searches for the signature of *persistently increasing queueing delay* within each p-stream. As described in Section 1.1.4.1, avail-bw is estimated as the highest probing rate before the signature is detected. RAPID's robustness is based on the assumption that the inter-packet gap between the *i-1*th and the *i*th packets expands if and only if the probing rate of the *i*th packet R_i exceeds the avail-bw. In other words, packets must experience increasing queueing delays if they keep probing at rates higher than avail-bw. This assumption holds on an ideal network environment, where packet

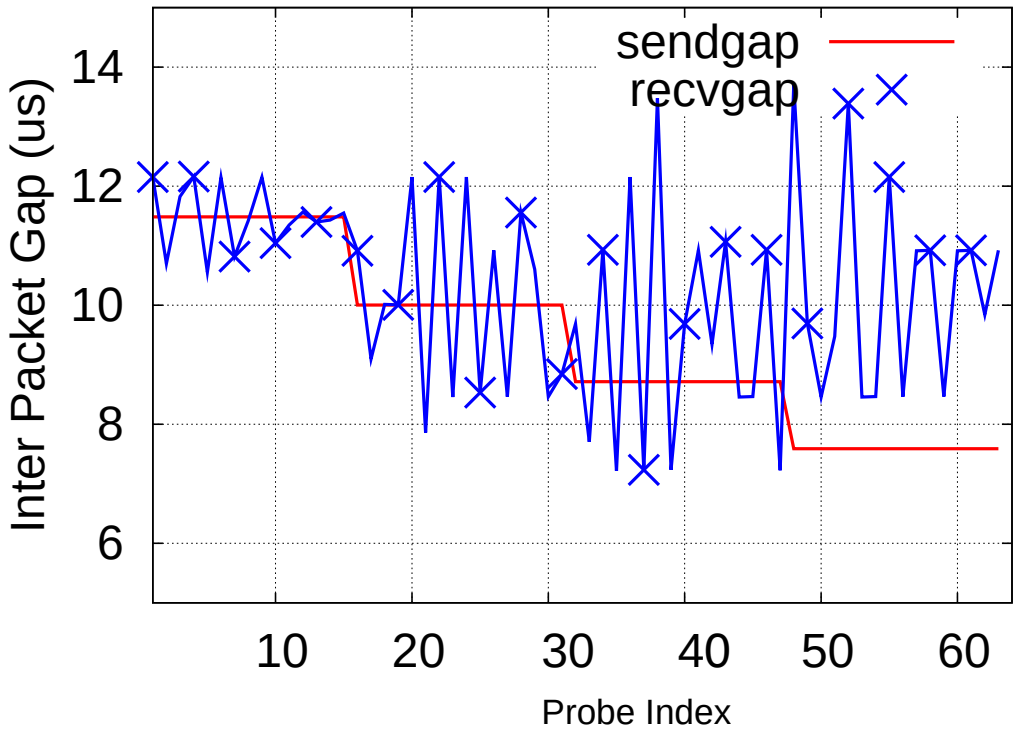


Figure 1.2: p-stream after the bottleneck link

arrivals at routers are uniformly distributed and packets experience queueing delays only at bottleneck routers when congestion is present.

1.2 Challenges

On real-world networks, there are two types of noise sources that void the assumption that packets must experience increasing queueing delays if they keep probing at rates higher than avail-bw: burstiness at bottleneck resources and transient queueing at non-bottleneck resources.

- *Burstiness in cross-traffic at bottleneck resources:*

  In a packet-switched network, traffic arrival can be fairly bursty at small timescales of 1 ms [**?** 32], and this burstiness can introduce noise into the persistent queueing delay signature. Fig 1.2 illustrates the inter-packet gaps observed after the bottleneck link of a p-stream encountering the same avail-bw as in Fig 1.1. It fails to identify persistent queueing until the 49th packet.

- *Transient queueing at non-bottleneck resources:*

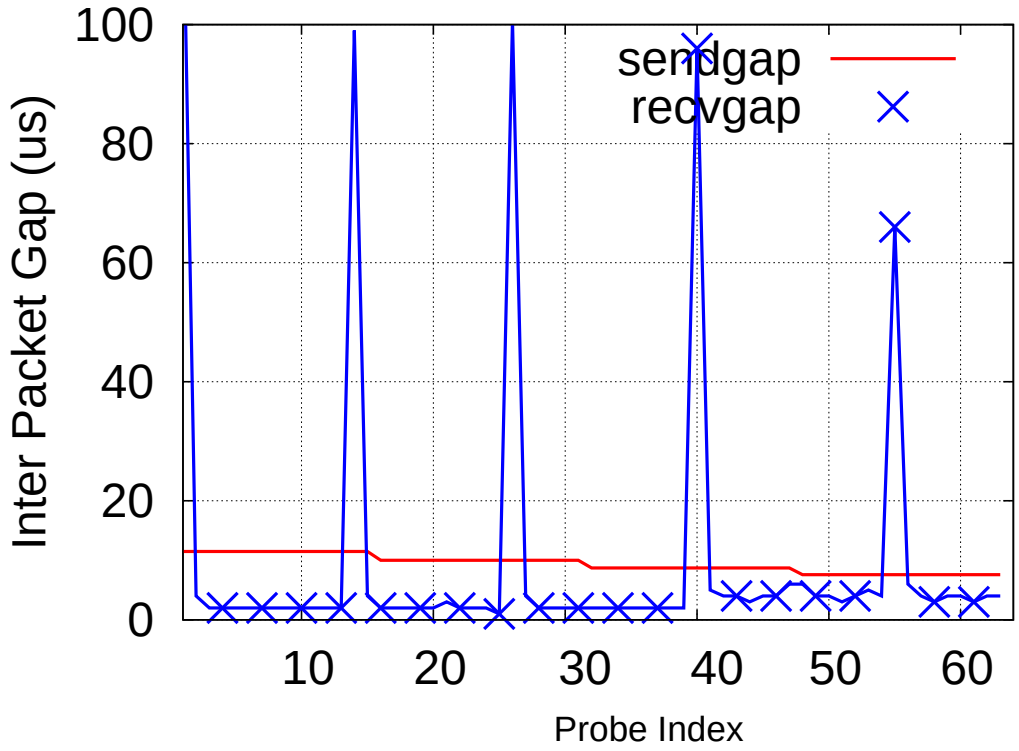


Figure 1.3: Probe Streams at the receiver
Inter-packet recvgaps present a "spike-dips" pattern due to interrupt coalescence at the receiving NIC.

Even a non-bottleneck resource can induce short-scale transient queues when it becomes temporarily unavailable because it is servicing competing traffic. This can happen, for instance, while accessing high-speed cross-connects at the switches, or while waiting for CPU processing after packets arrive at the receiver-side NIC.

In fact, *interrupt coalescence* has been identified as the major noise source for $ABest$ even on low-speed networks [20, 22, 33, 34]. Interrupt coalescence is a mechanism invented to improve system performance by reducing interrupt handling overhead. The NIC does not generate an interrupt for each packet arrival event; instead, it waits for more packets to arrive and then generates a single interrupt that hands all packets to the operation system for TCP/IP processing. This forces packets to queue at the receiving NIC before being handed to the OS for timestamping, even if the CPU is available. The waiting time can be as long as hundreds of microseconds. Fig 1.3 plots the typical receive gaps at the receiver OS. The inter-packet gaps show a "spike-dips" pattern: the spike corresponds to the first packet in the queue, which is preceded by a large g^r that has accrued since the previous queue's delivery; the following dips correspond to the rest of the packets in the queue, which are buffered back

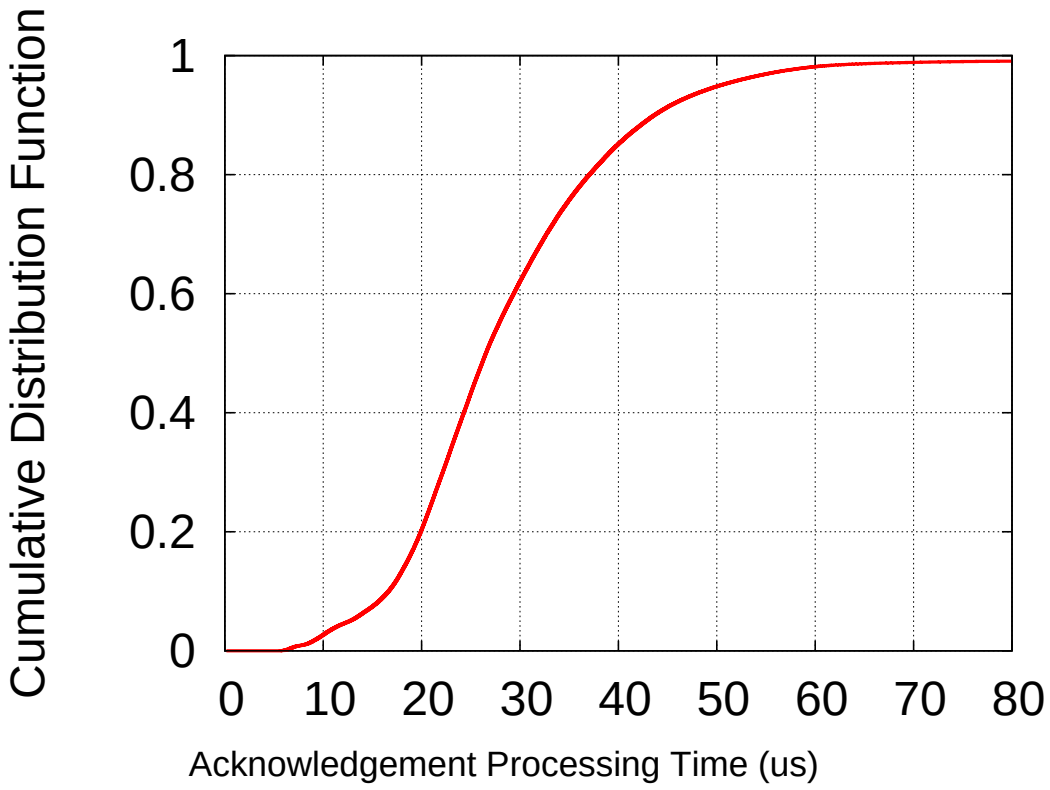


Figure 1.4: Processing Delay From a Packet Arrival to an ACK generated

to back and show negligible gaps. With interrupt coalescence, the *persistently increasing queueing delay* signature is completely unrecognizable. For every p-stream, avail-bw is estimated as the highest probing rate, leading to persistent overestimation.

1.2 Goal

Our second objective in this dissertation is to consider and evaluate techniques for reducing the impact of fine-scale noise in inter-packet gaps, which will help to achieve robust bandwidth estimation.

1.2 Stability/Adaptability Trade-off

1.2 RAPID Requirement

RAPIDrequires two features, adaptability and stability, for optimal functioning.

- *Adaptability:* RAPID should be able to track the variations in avail-bw closely at short intervals to ensure that it can observe and react to abrupt avail-bw changes in a timely fashion.

- *Stability:* However, RAPID should only adapt to avail-bw at a longer timescale; with relatively stable avail-bw, the longer timescale gives accurate results, and with highly volatile avail-bw, there is little point in adapting to an $ABest$ value that may have already changed drastically.

1.2 Challenges

Unfortunately, RAPID's requirements for stability and adaptability cannot be simultaneously satisfied. After estimating avail-bw for each p-stream, RAPID then adjusts the average sending rate for the next p-stream based on the updated $ABest$. It uses the same timescale (given by the number of packets in each p-stream) to probe for avail-bw and to adapt to changes in avail-bw. Thus, there exists an intrinsic *trade-off* between the two timescales. The stability requirement favors longer p-streams, whose longer probing timescales are believed to be less impacted by noise and to yield more robust $ABest$ [35]. These longer p-streams also allow RAPID to adapt its transmission rate to a longer probing timescale, at which the avail-bw is stabler and less noisy. The adaptability requirement, however, favors shorter p-streams. A shorter probing timescale enables the protocol to sample avail-bw more frequently and track it more closely, as well as to respond more quickly to changes in network condition.

1.2 Goal

Our third goal is to design new mechanisms into the RAPID control algorithm that can alleviate the trade-off between stability and adaptability, allowing RAPID to track changes in avail-bw closely, yet to only adapt to avail-bw at stabler timescales.

1.2 Deployability Within TCP Stack

1.2 RAPID Requirement

The dominant transport protocol used by most applications is TCP. For RAPID to achieve widespread impact without requiring applications to change their code-base, RAPID should work with existing APIs and with existing TCP headers. This will ensure that applications and network edge devices will continue to work seamlessly after deployment of the protocol. RAPID should also be pluggable within widely deployed TCP stacks; system administrators require that the protocol implementation be able to be loaded (or unloaded) on the fly without bringing down a server and without requiring a complete kernel build.

```
struct tcp_congestion_ops {
    ...
    /* initialize private data (optional) */
    void (*init)(···);
    /* return slow start threshold (required) */
    u32 (*ssthresh)(···);
    /* do new cwnd calculation (required) */
    void (*cong_avoid)(···);
    /* call before changing ca_state (optional) */
    void (*set_state)(···);
    /* call when cwnd event occurs (optional) */
    void (*cwnd_event)(···);
    /* call when ack arrives (optional) */
    void (*in_ack_event)(···);
    /* hook for packet ack accounting (optional) */
    void (*pkts_acked)(···);
};
```

Figure 1.5: The Linux *tcp_congestion_ops* Interface

1.2 Challenges

Ensuring that the RAPID implementation integrates seamlessly with existing protocols and servers is a significant challenge for two reasons, which will be explored in the next two sections of this thesis.

1.2 "gap-clocking" vs "ack-clocking"

RAPID's packet-sending process is fundamentally different from that of the state-of-the-art TCP. Since Linux is the most widely used open-source system, and the system upon which all TCP protocols except Compound are tested, we use the Linux TCP packet-sending process as an example to illustrate the difference between RAPID and TCP. In Linux, the packet-sending process is "window-controlled" and "ack-clocked." The "window-controlled" aspect means that a sliding window of *cwnd* size is maintained for outstanding packets. The "ack-clocked" aspect couples the receiving of ACKs with the transmission of new packets; once an ACK for the 1st packet in the *cwnd* arrives, TCP updates *cwnd*, advances the sliding window, and sends more packets, for as long as *cwnd* allows.

In contrast, RAPID has no notion of *cwnd* —it keeps sending p-streams as long as there are enough packets from the application. Further, RAPID's transmission of packets is not triggered by ACK arrivals, but is "gap-clocked": packets are not sent until a relative gap from the preceding packet has elapsed. The receiving of ACKs and the sending of packets are completely independent of each other.

The current Linux kernel provides a congestion handler interface ($tcp\_congestion\_ops$) that implements various congestion control protocols in a pluggable manner (Fig 1.5 lists all interfaces). Using **cong_avoid**, it allows changes to the amount by which *cwnd* changes when ACKs are received, but it provides no method of controlling when a packet is sent out. Clearly, to support "gap-clocking," Linux needs modules beyond the TCP congestion control framework. Instituting support for gap-clocking would not only support RAPID; other protocols that rely on bandwidth estimation, even intermittently, also require "gap-clocking" to be realized.

1.2 Fine-scale Arrival Timestamping

In order to calculate avail-bw at ultra-high speeds, the timestamps of packet arrivals must be observed by the RAPID receiver, and communicated back to the sender, at μs levels of precision. It is intuitive to use the timestamp fields in TCP headers to communicate packet arrival times; once the receiver finds that the TSval field is set in a packet header, it echoes that value in the TSecr in the returning ACK, and writes the ACK generation time in the TSval field. However, the granularity of TCP header timestamping is implementation dependent, ranging from 500 ms to 1 ms [27]. Linux can provide accuracies of only 1 ms—a granularity that is still much coarser than the inter-arrival gaps, which are measured at μs timescales. With such coarse granularity, the inter-arrival gaps become straight 0s. Moreover, timestamps are produced when ACKs are created, rather than when the respective packets are received. This belated timestamping means that gaps are subject to variable ACK processing times. Fig 1.4 plots the time difference between packets' actual arrival times—the times when packets are handed from the receiving NIC to the receiver OS—and the value recorded in TCP timestamps in the returning ACKs. We find that the two can differ by up to 60 μs.

The high-precision timing requirements of RAPID packet arrivals require some mechanism beyond simply relying on TCP timestamping.

1.2 Goals

Our third goal is thus to consider mechanisms that enable RAPID to be loaded as a pluggable module on widely deployed TCP stacks, while supporting its high-precision, fine-scale gap-clocking and timestamping requirements.

1.3 Thesis Statement

Because of its continual bandwidth probing and adapting, TCP RAPID extends the practical challenges faced by other delay-based and rate-based protocols to extreme levels. By carefully addressing each of these challenges, the performance demonstrated by RAPID in previous simulation-based work is deliverable in real-world networks.

1.4 Contributions

This thesis investigates mechanisms to address each of the identified challenges faced by packet-scale congestion control protocols in real networks. In the following sections, we summarize the key contributions of this thesis.

1.4 Creating Accurate IPGs

We survey and evaluate several state-of-the-art methods of creating inter-packet gaps using only software. We find that the method used in PSPacer [36] is able to create inter-packet gaps on 10 Gbps links with errors of less than 0.5 μs.

1.4 Denoising for Accurate Bandwidth Estimation

We evaluate several state-of-the-art denoising techniques for bandwidth estimation (Pathload, IMR-Pathload and PRC-MT) and find that none of them scale well to 10 Gbps links. Pathload and IMR-Pathload yield highly inaccurate gaps, and PRC-MT requires extremely long p-streams of at least 320 packets, introducing considerable overhead to the network. To address this, we develop a novel denoising technique, Buffering-Aware Spike Smoothing (BASS), and apply it to different bandwidth estimators. We find that, with BASS, those estimators are able to give more robust estimations at 10 Gbps links, even with short p-streams of only 64 packets. In addition, we use a machine-learning approach to establish the relationship between noisy inter-packet gaps and avail-bw, and we use this automatically learned knowledge to compute avail-bw from the noisy p-streams. We find that the machine-learned estimator achieves estimation accuracy far superior to that of any of the previously-mentioned human-designed techniques, including BASS.

1.4 Addressing the Stability/Adaptability Trade-off

To address RAPID's trade-off between stability and adaptability, we decouple the probing timescale from the adapting timescale. Instead of updating the average transmission rate of p-streams for every avail-bw estimate made, RAPID is set to probe avail-bw with shorter p-streams and to adapt its sending rates to the mean of the latest γ bandwidth estimates only every γ p-streams.

1.4 Implementing on Standard Protocol Stacks

We develop mechanisms to realize gap-clocking at the sender and microsecond-resolution packet timestamping at the receiver within standard Linux protocol stacks.

1.4 Realizing gap-clocking

To realize gap-clocking, we design a packet scheduler in the link layer that shares some data structures with the TCP module. It groups packets in the interface queue into p-streams, computes inter-packet gaps based on the average sending rate computed by the TCP module, and sends packets to the outbound NIC according to inter-packet gaps .

1.4 Achieving Fine-scale Arrival Timestamping

To timestamp packet arrivals with at least μs accuracy, we develop a set of packet-scheduler modules: two modules at the receiver (one to timestamp incoming packets with μs resolution and another to replace this high-resolution timestamp in the ACK TSval field before ACKs are sent out), and one module at the sender (to record the timestamp in the returning ACKs and replace it with the ms-resolution value to ensure proper TCP header processing).

1.4 Evaluation of RAPID in Ultra-high-speed Testbed

Incorporating the previously mentioned mechanisms, we implement RAPID as pluggable kernel modules in Redhat Linux 6.7 with kernel version 2.6.32. Specifically, we

- employ the PSPacer mechanism in the packet scheduler to create precise inter-packet gaps;
- use BASS to denoise p-streamobservations for robust bandwidth estimation;

- decouple the probing timescale from the adapting timescale with $X = 3$; and
- use three packet-scheduler modules for timestamping with μs accuracy.

We evaluate this implementation in our 10 Gbps testbed and compare its performance to that of other congestion-control protocols, and we find that our implementation lives up to its performance in simulations: it is adaptive to random packet losses and to bursty cross traffic, consistently providing higher throughputs than other protocols while remaining non-intrusive to coexisting conventional TCP flows.

We then test which of these mechanisms has the strongest effect on the RAPID implementation by removing them one by one and repeating the previous evaluations without each one. We find that each of these mechanisms is crucial: RAPID's performance gains cannot be achieved without all four of the mechanisms in place.

1.4 Apply the Lesson to Other Delay- and Rate-based Protocols

We adapt the mechanisms designed for RAPID to other delay-based and rate-based protocols. Specially, for TCP Vegas we measure RTTs at a finer-grained level by recording packet arrivals with the Qdisc mentioned in 1.4.4; we also use the "Dummy-packet" mechanism to reduce traffic burstiness for TCP pacing, We find that these mechanisms also help other delay- and bandwidth-based protocols deliver better performance that is adaptable to real-world high-speed networks.

1.5 Outline

The following is the roadmap for this thesis. Chapter 2 provides background knowledge necessary to understand the novel contributions of this thesis, including an overview of state-of-the-art congestion control protocols, a description of the TCP RAPID congestion control process, and an explanation of the bandwidth estimation logic used in RAPID. Chapter 3 describes the 10 Gbps testbed that hosts all experiments in this thesis. Chapters 4 through 7 address the four practical challenges of RAPID that have been briefly described in this introduction; chapter 4 presents our explorations of different methods of gap creation to ensure high precision in gap timing, chapter 5 focuses on our efforts to achieve robust bandwidth estimation, chapter 6 proposes a novel method of bandwidth estimation inspired by machine learning, and finally, chapter 7 describes how we implement RAPID in a standard Linux protocol stack—a pairing that has until now been regarded as intrinsically incompatible. In Chapter 8, we describe the results of intensive evaluations of our

RAPID implementation in our testbed, showing that its experimental performance confirms the performance gains promised in [2].

CHAPTER 2: BACKGROUND

This chapter provides background knowledge that is related to this thesis, including basics and evolution of TCP, RAPID congestion control algorithm that was proposed in its original paper [2], common methods of bandwidth estimation, and the brief introduction of machine learning algorithms that are mentioned in this thesis. Readers familiar with this background can skip this chapter.

2.1 Historical Perspective of TCP Congestion Control

In this section, we briefly review the evolution of TCP transmission control protocol in the past decades, focusing on its congestion control algorithm. We first describe the functionality of TCP congestion control, discuss its deficiency in newer high-speed networks, and introduce state-of-the-art solutions to address it.

2.1 Transport Control Protocol

Computer networks are often described using a layered conceptual model. TCP/IP reference model is the most widely accepted model by industry, which abstracts the network into four layers. From top to down, they are: application layer, transport layer, internet layer, and network access layer. Each layer uses some *protocol* — a set of rules regarding the procedures that two entities communicate with each other — to carry out different functionalities. In the topmost application layer, applications exchange information between remote end-hosts according to application protocols to perform specific tasks, e.g. file transmission (which uses FTP protocol), and solving domain names (which uses DNS protocol). Protocols of transport layer are responsible for delivering data between the two application processes on the two end-hosts. Protocols of Internet layer are responsible for delivering data from the sending host to the receiving host over the Internet. The network access layer moves data without error on the physical link between two directly-connected hardwares. The upper-layer protocol relies on the functionality provided by the layer below it.

Transmission control protocol (TCP) sits at transport layer, and uses the IP addresses of the two end hosts, as well as the port numbers of the two corresponding processes as the unique identifier for each

end-to-end (from the sending host to the receiving host) connection. It provides reliably transmission using its loss-recovery mechanism — if the receiver fails to acknowledge the sender on the receipt of a packet within a certain amount of time, the sender immediately retransmits that packet, and repeats this process until this packet is acknowledged. Besides, it employs mechanisms that react to network congestion (which will be described in Section 2.1.2). The congestion control mechanism allows each connection to detect congestion and slow down transmission accordingly, preventing the Internet from "congestion-collapse", where each connection experiences losses repeatedly due to persistent congestion, and fails to transmit data efficiently. TCP has been extensively utilized by many popular applications such as web, email, file sharing and video streaming. According to the monthly report by Caida [37], TCP carries the majority of total traffic on the current Internet backbone nowadays.

2.1 Loss-based Congestion Control

Network congestion occurs when the intermediary devices on the path, i.e. switches and routers, receive data packets faster than they can process. Packets are buffered in the internal queues of these devices, waiting to be served. Packet loss is a deterministic consequence of persistent congestion — with the queue building up, further incoming packets are discarded once the queue overflows.

The legacy TCP congestion control protocol (Tahoe, Reno, and New Reno) regards packet-loss as the indicator of network congestion, and developed an additive-increase and multiplicative-decrease (AIMD) strategy to react to it. We briefly describe the strategy as follows.

The AIMD strategy relies on the notion of congestion window (*cwnd*), which sets the upper bound of packets that can be in flight (unacknowledged) at any time. In other words, within each RTT, only a $cwnd$ worth of packets can be sent. The data transmission process consists of two phases — the connection enters *slow-start* phase in the beginning of the transmission, where *cwnd* is aggressively incremented by 1 upon every acknowledgement; once a packet-loss is detected, the connection halves *cwnd*, and switches to *congestion-avoidance* phase, where *cwnd* is gradually increased by $\frac{1}{cwnd}$ upon each acknowledgement.

With the advent of higher network speeds recent years, the legacy AIMD congestion-control strategy has been shown to utilize the path capacity inefficiently — the slow *cwnd* increment in congestion-avoidance phase causes it to take hundreds and thousands of RTTs to fully utilize gigabit paths [2]. Besides, the drastic *cwnd* reduction upon each detected packet loss [5, 4, 1] costs it hundreds of RTTs more to obtain full link utilization again — a connection requires an extremely low loss rate of 0.000001% to maintain high link

utilization. To address the issue, new protocols have been proposed that behave more aggressive in both slow-start and congestion-avoidance phase than AIMD — BIC, Cubic, HTCP, HighSpeed, and Scalable are prominent examples of them. These protocols have been reported to improve link utilization on highspeed links, and have been deployed in the Linux kernel protocol stack. We refer to these congestion-control protocols that only respond to packet-loss as "loss-based", and briefly describe the algorithms of those loss-based protocols used in the test-bed evaluation in Chapter 8.

2.1 NewReno

TCP NewReno was developed upon earlier TCP protocols Tahoe and Reno with a more efficient loss-recovery mechanism. It was standardized by IETF in 2004 [38], since then has been referred to as the standard TCP. Its congestion control inherits the AIMD strategy:

$$cwnd = \begin{cases} cwnd + 1 & \text{, upon a receipt of acknowledgement in slow start phase} \\ cwnd + \frac{1}{cwnd} & \text{, upon a receipt of acknowledgement in congestion avoidance phase} \\ \frac{cwnd}{2} & \text{, upon a loss detected by duplicate acknowledgements} \end{cases} \quad (2.1)$$

Its effect is that *cwnd* gets doubled in slow-start phase, and only increased by 1 in congestion-avoidance phase.

2.1 BIC

BIC [4] addresses the inefficiency of NewReno by increasing *cwnd* more boldly. Similar as NewReno, BIC doubles *cwnd* every RTT in slow-start phase, until a packet loss is detected. It then sets $cwnd_{max}$ as the *cwnd* when loss occurs — the upper bound of *cwnd* value the path can handle must be smaller then $cwnd_{max}$. It also sets $cwnd_{min}$ to $\beta \times cwnd_{max} (\beta < 1)$, which is regarded as a "safe" *cwnd* that will not overload the path. Then BIC enters congestion avoidance phase, performing binary search between $cwnd_{max}$ to $cwnd_{min}$. $cwnd_{max}$ and $cwnd_{min}$ are updated accordingly every time loss occurs.

2.1 Cubic

The binary search technique can be too aggressive for TCP on low-speed networks, which tends to cause repeated packet losses. To address this issue, TCP uses a new cubic *cwnd* growth function as follow:

$$cwnd = c \times (t - K)^3 + cwnd_{max} \tag{2.2}$$

where t is the elapsed time since the last window reduction after packret loss, $K = \sqrt[3]{\frac{cwnd_{max} \times \rho}{C}}$, and C is a scaling factor. Similar to BIC, it denotes the *cwnd* before a packet loss occurs as $cwnd_{max}$. After a window reduction by half after loss, Cubic grows its *cwnd* rapidly. When it gets closer to $cwnd_{max}$, it slows down the increasing trend — *cwnd* increment becomes zero at when $cwnd = cwnd_{max}$. After that, *cwnd* grows in a reverse fashion, probing more and more boldly towards even higher rates. [5] has shown that Cubic enhanced the stability and link utilization of BIC, and also highly scalable at high-speeds. Cubic has been the default congestion control protocols since Linux kernel 2.6.19.

2.1 HighSpeed

For legacy TCP, the *cwnd* reduction after packet loss is always proportionally to the *cwnd* value before loss occurs — the larger *cwnd* grows to, the more drastically *cwnd* gets reduced. HighSpeed TCP [1] made a sharp observation that such severe reduction is unnecessary when *cwnd* is large, which often indicates that the network capacity is high. Instead, with a larger *cwnd*, the flow should behave more aggressively to quickly grab the abundant bandwidth resource after packet loss. HighSpeed modifies the *cwnd* growth at each acknowledgement as follows:

$$cwnd = \begin{cases} cwnd + \frac{a(w)}{cwnd} & \text{, congestion avoidance} \\ (1 - b(w)) \times cwnd & \text{, upon loss} \end{cases} \tag{2.3}$$

It uses a table to match the current *cwnd* to $a(w)$ and $b(w)$ — a higher *cwnd* corresponds to a higher $a(w)$, and a lower $b(w)$. For low-speed links (when *cwnd* is smaller than some threshold), HighSpeed behaves identical as TCP NewReno, without starving low-speed legacy TCP flows.

2.1 Scalable

Like HighSpeed, Scalable TCP [3] also aims to make *cwnd* grow more quickly than the one packet per RTT, as well as to improve the loss recovery time. It also has a threshold *cwnd* size, below which Scalable TCP reconciles to the legacy TCP. When *cwnd* increases beyond that threshold, Scalable updates *cwnd* as follows upon every packet acknowledgement:

$$cwnd = \begin{cases} cwnd + 0.01 & \text{, congestion avoidance} \\ cwnd - 0.125 \times cwnd & \text{, upon loss} \end{cases} \tag{2.4}$$

The congestion window is increased by 1% of the current *cwnd* value each RTT, and reduced less drastically after loss is detected.

2.1 Hybla

In both slow-start phase and congestion avoidance phase of legacy TCP, the *cwnd* increasing rate is inversely proportional to RTT. This penalizes long-RTT flows with lower throughput when sharing paths with short-RTT flows, especially for those terrestrial satellite traffic where RTTs can reach 500ms and losses can be frequent due to terrestrial link errors. TCP Hybla [10] specifically targets at this issue for satellite flows, and proposes to remove performance dependence on RTT using the following algorithm:

$$(\frac{RTT}{RTT_0})^2 \times \frac{1}{cwnd} + cwnd \tag{2.5}$$

RTT_0 is a reference value used for all Hybla connections. The Hybla mechanism allows satellite traffic to increase *cwnd* at a rate proportional to RTT, significantly reducing the performance disparity against these long-RTT flows.

2.1 Delay-based and Rate-based Congestion Control

As mentioned in Section 1.1.1, there exist two disadvantages of solely reliance on the binary signal of packet-loss to indicator congestion. Firstly, it often leads to a too conservative design of *cwnd* increment after loss recovery, in afraid of causing further losses. Secondly, it prohibits the protocol from reacting to congestion in an earlier stage before queues start to overflow on the congested path. In Section 1.1.2, we

introduced the efforts of using queuing delay and available bandwidth as alternative congestion indicators to address the issues of loss-based design. In this section, we describe the congestion control algorithms of the protocols used in the evaluation in Chapter 8. Among them, TCP Vegas is a truly delay-based protocol which solely relies on delay to perform congestion control in congestion-avoidance phase. Other protocols, namely, Hybla, H-TCP, Illinois, Yeah, Fast and Compound, follow a hybrid of delay-based and loss-based approaches.

2.1 Vegas

TCP Vegas was developed in 1994 [9], which first introduced delay as congestion indicator. The designer of Vegas pointed out that congestion can be detected in an earlier stage before triggering packet-loss. When a network path starts to congest, packets are queued for longer in the network queues, leading to larger RTTs. Thus, earlier congestion can be observed by whether RTTs packets experienced increase. Vegas keeps track of the minimum RTT throughout the transfer as $baseRTT$, computes the minimum RTT observed during the past round-trip time as $minRTT$, and update *cwnd* as follows:

$$diff = cwnd - cwnd \times \frac{baseRTT}{minRTT}, \tag{2.6}$$

$$cwnd = \begin{cases} cwnd + 1 & \text{, if } diff < \alpha \\ cwnd & \text{, if } \alpha \leq diff \leq \beta \\ cwnd - 1 & \text{, if } diff > \beta \end{cases} \tag{2.7}$$

If $diff$ is less that α, Vegas regards the path as non-congested, and increases *cwnd* linearly as NewReno does. Once $diff$ goes beyond α, it signals possible congestion on the path, and Vegas stops *cwnd* growth. A $diff$ value larger than β indicates severe queuing delays, which is a definite sign of path congestion. In this case, Vegas linearly reduces *cwnd*. [9] reported that Vegas achieved higher throughput than NewReno on congested links, due to its earlier congestion detection which avoided many packet loses.

2.1 H-TCP

Hamilton TCP [39], or H-TCP for short, uses delay to determine the amount of *cwnd* reduction once packet-loss is detected — rather than halving *cwnd*, it is reduced by a fraction of $\frac{RTT_{max}}{baseRTT} - 1$. After loss recovery, the *cwnd* increment rate α is a function of the elapsed time since most recent packet-loss — it first increases as slow as the legacy TCP does, and then slowly accelerates the increasing as time elapses.

2.1 Illinois

The idea of TCP Illinois [17] is to use AIMD strategy, but adopts delay as a secondary congestion signal to determine the amount of *cwnd* increment and decrement — once queuing delay experienced by packets increases, the *cwnd* increment rate should decrease, and reduction-rate after loss should increase, and vice versa. Illinois maintains a variable da as the average queuing delay experienced by packets within one RTT, and updates *cwnd* as follows:

$$cwnd = \begin{cases} cwnd + \frac{\alpha(da)}{cwnd} & \text{, congestion avoidance} \\ cwnd \times (1 - \beta(da)) & \text{after loss} \end{cases} \tag{2.8}$$

where $\alpha(da)$ and $\beta(da)$ are functions of the queuing delay:

$$\alpha(da) = \begin{cases} \alpha_{max} & \text{, if da} \leq \text{d1} \\ \frac{k1}{k2+da} & \text{, otherwise} \end{cases} \quad \beta(da) = \begin{cases} \beta_{min} & \text{, if da} \leq \text{d2} \\ k_3 + k_4 \times da & \text{, d2} \leq \text{da} \leq \text{d3} \\ \beta_{max} & \text{, } otherwise \end{cases} \tag{2.9}$$

where d1, d2 and d3 are pre-defined thresholds. Thus, in congestion avoidance phase, *cwnd* increment rate is a reciprocal function of queuing delay; after packet loss, *cwnd* reduction rate increases linearly with queuing delay.

2.1 Yeah

Scalable TCP has been shown to efficiently use high link utilization, but may starve NewReno flows sharing the network path. TCP Yeah [7] attempts to maintain the advantage of Scalable, and avoid its disadvantage. It uses two modes, fast mode and slow mode. In fast mode, Yeah behaves in the same manner as Scalable, while in slow mode it behaves as NewReno. The watershed of two modes are the queuing related variables L and Q. $L = \frac{RTT}{baseRTT}$, which indicates the extent of average queuing delay experienced by a *cwnd* worth of packets. $Q = (\frac{RTT}{baseRTT} - 1) \times cwnd$, which is used to infer the number of packets queued up over the path. Once both L and Q reaches some thresholds, Yeah switches to slow mode, which avoids further congestion and stays fair with lows-speed NewReno flows. Otherwise, it stays in fast mode.

2.1 Westwood

TCP Westwood [40] aims at improving the behavior of NewReno after packet losses — the halving of *cwnd* is too drastic and leads to low link utilization in loss recovery phase, especially on wireless links where losses are not induced by congestion but by link errors. Westwood proposes to set transmission rate to the available bandwidth avail-bw on the path when loss occurs. Westwood estimates avail-bw as the returning rate of acknowledgements before loss is detected — this rate is equal to the receiving rate at the receiver, which indicates the maximum rate the flow is able to achieve. This mechanism allows Westwood to maintain high efficiency after losses without causing excessive congestion, and has shown be significantly improve path utilization on lossy links [40].

2.1 Veno

TCP Veno [41] is a synthesis of Vegas and NewReno designs. For each RTT, Veno maintains a variable $diff$ the same as Vegas, and uses $diff$ to determine the amount of *cwnd* increment:

$$cwnd = \begin{cases} cwnd + 1 & \text{, if } diff \leq \beta \\ cwnd - 0.5 & \text{, if } diff > \beta \end{cases} \tag{2.10}$$

The above mechanism makes *cwnd* growth less aggressive than NewReno when queuing delay continues to increase.

On wireless links, packet losses may not be caused by congestion, but by wireless link errors. In this case, halving *cwnd* as NewReno does is unnecessary and leads to low link utilization. One highlight of Veno design is that it tries to distinguish the causation of packet-losses on wireless links, with the aid of delay — if $diff$ is smaller than β when the loss occurs, the path is non-congested and the packet-loss regarded as caused by link errors; otherwise, the loss is caused by congestion. In the former case, *cwnd* is reduced by only 20% after loss, and 50% for the latter case.

2.1 TCP LP

TCP LP [13] is short for Low-Priority. As manifested by its name, it targets low priority traffic on the network, such as background flows with no strict delay or throughput requirements. A LP flow only uses excess bandwidth left by other traffic on the path. It achieves this by pro-actively detect early congestion

through monitoring one-way delays for every packet. TCP LP removes the slow-start phase of NewReno, but starts with additive-increase phase which increases *cwnd* by 1 every RTT. Once the average one-way delay in one RTT exceeds the minimum delay observed, LP regards the path in congested state, and conducts multiplicative-decrease which reduces *cwnd* by half.

2.1 Fast

Fast TCP [11] is a delay-based protocol, which aims at maintaining fixed-queue utilization for each flow. It computes RTT for every packet, and maintains $baseRTT$ as the minimum RTT observed. At regular short intervals, *cwnd* is updated via the following weighted average:

$$cwnd(T) = (1-\gamma) \times cwnd(T-1) + \gamma \times (\frac{baseRTT}{RTT} \times cwnd(T-1) + \alpha) \tag{2.11}$$

where *cwnd* (T) is the computed *cwnd* value for te T-th interval, γ is a constant between 0 and 1, and α a constant integer. The effect of the above *cwnd* update mechanism is at any time the maximum queue size on the path is α. [11] and [42] have shown that Fast helps to achieve high throughput as well as better flow stability on high-speed links.

2.1 Compound

Compound TCP maintains two congestion windows — a loss-based window W_{loss} and a delay-based window W_{delay}. The effective *cwnd* is the sum of these two. In congestion avoidance phase, the W_{loss} window is updated in the same way as NewReno, which increases by $\frac{1}{W_{loss}+W_{delay}}$ for each acknowledgement; while W_{delay} is derived from TCP Vegas. Compound TCP has been adopted by default in Windows systems since the release of Vista in 2008.

2.2 Bandwidth Estimation Techniques — PRM

Bandwidth estimation is a key component in many domains. It is used in congestion control protocols, e.g. RAPID, NF-TCP, UDT and PCP [26, 25, 24] to directly estimate avail-bw.

Major content providers nowadays carry the majority of Internet traffic [43]. For each end user, there are multiple content servers available; and for each server, there exist multiple paths to the end user. Bandwidth estimation is also commonly involved in server-selection and path-selection that facilitate content providers to

elect a proper server for each user for best user experience. The past decade has witnessed a rapid growth in the design of techniques for estimating available bandwidth [23, 21, 20, 22, 33]. In this section, we introduce the basic techniques for bandwidth-estimation that are widely used in existing tools.

2.2 PRM-based Bandwidth Estimation

Existing bandwidth estimation techniques base their estimation logic on two prominent models, namely, the probe gap model and the probe rate model (PRM) [23]. Tool evaluations have shown that PRM tools are more robust in the presence of multiple congested links [44, 45]. As a result, RAPID employs PRM in its bandwidth estimation logic. Below, we briefly summarize the PRM approach.

PRM tools typically send multiple packets at the same probing rate, in groups commonly referred to as *p-streams*—probing rate, r_i, of the ith packet is achieved by controlling the inter-packet gap as: $g_i^s = \frac{p_i}{r_i}$, where p_i is the size of the ith packet and g_i^s is the gap between packet $i-1$ and i. We use N to denote the length of a probe-stream, in terms of number of packets.

2.2 the principal of "self-induced congestion"

PRM tools base their bandwidth estimation logic on the principle of *"self-induced congestion"* — packets are expected to experience longer and longer queuing delays if they keep arriving at the bottleneck link at a higher rate than the available bandwidth. According to this principal: if $r_i >$ avail-bw, then $q_i > q_{i-1}$, where q_i is the queuing delay experienced by the ith packet at the bottleneck link, and avail-bw is the available bandwidth on that link. Assuming fixed routes and constant processing delays, this translates to $g_i^r > g_i^s$, where g_i^r is the gap observed at the receiver between the ith and $i-1$th packets.

Most tools send out multiple packets at each probing rate, and check whether or not the receive gaps g^r are consistently higher than the send gaps g^s. They try out several probing rates and search for the highest rate r_{max} that does *not* cause self-induced congestion. This r_{max} is then regarded as an estimate of bandwidth on the path.

There are two dominant strategies for searching r_{max}— single-rate feedback-based probing and multi-rate probing. Next we describe each of these two.

2.2 Feedback-based Single-rate Probing:

The sender relies on iterative feedback-based binary search to find the r_{max}. Starting with a probing range $\{R_{min}$,$R_{max}\}$, the sender sends all packets within a p-stream at the same probing rate $R = \frac{R_{min}+R_{max}}{2}$, and waits for receiver feedback on whether the receive gaps increased or not. If such signature of self-induced congestion is found, the sender updates R_{max} as R; otherwise, it sets R_{min} to R. The process is repeated until R_{min} and R_{max} converges ($R_{max} - R_{max} < \epsilon$).

Pathload Pathload is the most prominent of such tools [20]. IMR-Pathload [33] mentioned in Section 5.1 rely on Pathload to estimate avail-bw. Here we briefly describe its estimation logic.

Pathload sends p-streams of 100 packets within each, and every packet probes for an identical probing rate R. One-way delays for every packet are observed — we denote them as a vector $D =< delay_1, delay_2, ..., delay_N >$. D is then fed into two trend-detection tests, namely, PCT and PDT, to detect whether an increasing trend of one-way delays exhibit.

PCT stands for pairwise comparison test, which tests the fraction of consecutive one-way delays that are increasing. PCT computes a metric S_{PCT}: $S_{PCT} = \frac{\sum_{k=2}^{N}(I(delay_k > delay_{k-1}))}{N-1}$, where $I(delay_k > delay_{k-1})$ is 1 if $delay_k > delay_{k-1}$ is true, otherwise this term is 0.

PDT stands for pairwise difference test, which quantifies how strong is the start-to-end one-way delay variation, relative to the one-way delay absolute variations for each pair of consecutive packets. It computes the metric of S_{PDT}: $S_{PDT} = \frac{delay_k - delay_1}{\sum_{k=2}^{N}|delay_k > delay_{k-1}|}$.

If $S_{PCT} > 0.66$ and $S_{PDT} > 0.55$, then the p-stream is regarded as strongly exhibiting signature of self-induced congestion, and consequently its probing rate exceeds avail-bw. If $S_{PCT} < 0.54$ and $S_{PDT} < 0.45$, then the p-stream is free of self-induced congestion, and R must be smaller than avail-bw. Otherwise, whether R exceeds avail-bw or not is regared as "ambiguous".

2.2 Multi-rate Probing

The sender uses **multi-rate** probing without relying on receiver feedback — each pstream includes $N = N_r \times N_p$ packets, where N_r is the number of probing rates tried out. The sender looks for the highest probing rate that did not result in self-congestion. Fig 1.1 illustrates a multi-rate pstream with $N_r = 4, N_p = 16$. The receiver inter-packet gaps (g_r) are consistently larger than the send gaps (g_s) starting with the third probing rate, so the second probing rate (r_{max} is taken as an estimate of the avail-bw. Multi-rate

probing facilitates the design of light-weight and quick tools, which transmits a smaller number of packets and estimates using much less time [45]. PathChirp is the most prominent of such tools [22], whose algorithm is described in Algorithm 1. TCP RAPID adopts this algorithm for bandwidth estimation on multi-rate p-streams.

Algorithm 1 pathchirp_abest(g_s,g_r, GAP_NS_EPSILON)

```
1: m=i=1, L=4
2: for k in 1 to N: AB_k = 0
3: while i < N     j = i+1
4:    while g_r^j ¿ g_r^i + GAP_NS_EPSILON
5:       j++
6:    if j - i + 1 >= L
7:       for k in m to j: AB_k = R_s^{i-1}
8:       i=m=j+1
9:    else
10:      i = i + 1
11: return avg(AB_i)
```

2.3 TCP RAPID

In this section, we describe the congestion-control algorithm of our targeted protocol, TCP RAPID. Instead of simply relying on packet loss as congestion feedback, RAPID is a bandwidth-based protocol — it continuously sends packets out in logical groups referred to as p-streams to estimate avail-bw, and then uses the estimates to guide the transmission rate of following p-streams. The congestion-control mechanisms consists of three components operating in a closed loop (Fig 2.1). Given an average send rate R_{avg} informed by the *rate adapter*, the *p-stream generator* sends packet out in units of p-streams. Once the ACKs for a full p-stream return, the *bandwidth estimator* calculates the end-to-end avail-bw, based on which the *rate adapter* updates the sending rate for the next p-stream.

Pstream Generator

The Pstream Generator is responsible for generating p-streams, with accurate inter-packet gaps between each pair of subsequent packets. RAPID uses each data packet sent for probing the end-to-end path for

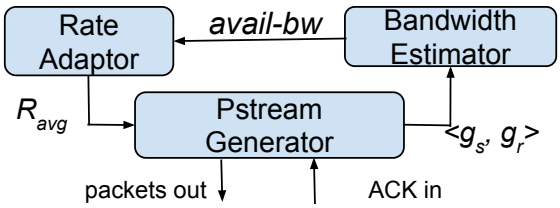


Figure 2.1: TCP RAPID Architecture

some rate R, by controlling the send-gap (g_s) from the previous packet sent as: $g_s = \frac{P}{R}$. Within a p-stream, packets are sent at N_r different exponentially-increasing rates (with N_p packets sent at each rate): $R_i = R_{i-1} \times s, i \in [2, N_r], s > 1$. Fig 1.1 plots the gaps in an intended p-stream with $N_r = 4$, and $N_p = 16$. The average send rate of the full p-stream is set to R_{avg}, informed by the rate adapter.

Bandwidth Estimator

The mission of *bandwidth estimator* is to obtain an estimate for each fully-acknowledged p-stream, by observing queuing delays experienced by each packet within it.

The RAPID receiver records the arrival time of data packets and sends back the timestamps in ACKs. Once the sender receives all ACKs for a p-stream, it extracts these timestamps, computes the receive gaps (g_r), and feeds them to the bandwidth estimator. The send and receive gaps for those multi-rate p-streams, g_s and g_r, are used to compute an estimate for the end-to-end avail-bw based on the algorithm described in Section 2.2.1.3.

Rate Adapter R_{avg} is initialized to 100Kbps. Thereafter, every time $ABest$ is updated, the average sending rate of the next p-stream is also updated by applying a conditional low-pass filter as:

$$R_{avg} = \begin{cases} R_{avg} + \frac{l}{\tau} \times (ABest - R_{avg}), & ABest \geq R_{avg} \\ R_{avg} - \frac{1}{\eta} \times (R_{avg} - ABest), & ABest < R_{avg} \end{cases}$$

where l is the duration of the p-stream , and τ and η are constants. The effect of the above filter is that it takes about τ *time units* for R_{avg} to converge to an increased avail-bw, and η p-streams to converge to a reduced avail-bw.

The low-pass filter enables RAPID to simultaneously achieve two desirable properties. First, by setting R_{avg} lower than estimated avail-bw, it prevents RAPID to transmit at a higher speed than the network can currently handle. Meanwhile, within a single p-stream packets probe for rates both higher and lower than R_{avg} at smaller timescales — this gives the protocol ability to quickly detect changes in avail-bw.

2.3 Performance Promises in Simulation-based Evaluations

[2] evaluated the performance of RAPID in NS2 simulation on a 1Gbps testbed, and demonstrate that RAPID has close-to-optimal performance along several dimensions, most notable of which are:

- discovering and adapting quickly to changes in avail-bw within 4 RTTs;
- maintaining high link utilization throughput closely tracking and adapting to changes in avail-bw;
- having low-queuing profile at switches/routers on the bottleneck link;
- being highly friendly to low-speed legacy TCP flows.

However, the related experiments were completely conducted in simulation environment, which is free of real-world noise. To be specific, the *pstream generator* is able create perfect inter-packet gaps , the cross-traffic is highly smooth, and the *bandwidth estimator* is guaranteed to estimate avail-bw precisely. Whether these properties can be translated into real-world situation is questionable, and is to be explored in this thesis.

2.4 Machine Learning Algorithms

In Chapter 6 we develop a machine-learning based methodology to estimate available bandwidth. In this section, we introduce basic concepts of machine learning, as well as those machine-learning algorithms mentioned in Chapter 6.

Machine learning is the technique which allows computers to learn an algorithm that explores the relationship of collected data without being explicitly programmed, and then uses the learned algorithm to make predictions on new data. Machine learning is employed in a wide range of computing tasks, such as spam filtering [46], pattern recognition in image processing [47], stock-market prediction [48], etc. Recently, it has been introduced to the realm of computer networks — [49] employs machine-learning technique to analyze Internet traffic for malicious intrusion detection, [50] classifies Internet traffic using a learning process, [51] predicts throughput of TCP flows by learning the history of flow statistics, [52] explored the learnability of congestion-control algorithms, etc. Furthermore, [53] collected data for connection and link status, and let the machine learn an algorithm to distinguish congestion-induced packet-losses and error-induced losses, which has long been regarded as a notorious performance hurdle of wireless congestion control.

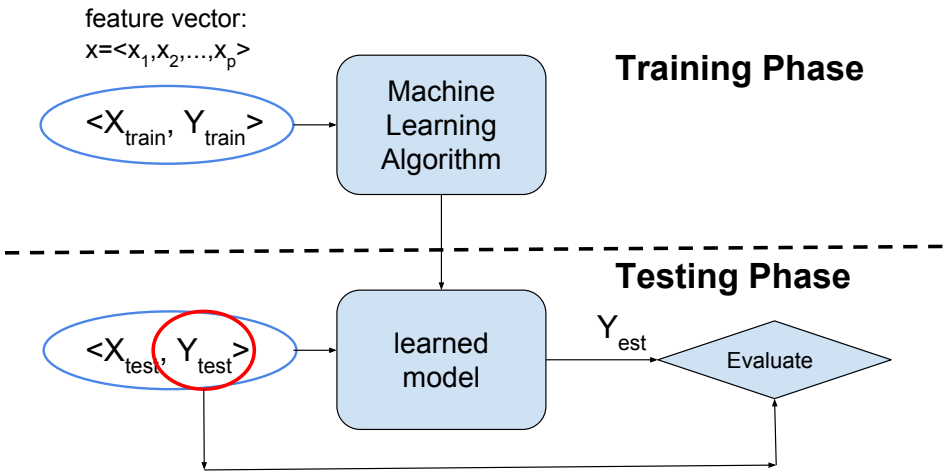


Figure 2.2: Machine Learning Training and Testing Phases

2.4 Classification and Regression

Machine learning is commonly used to solve two types of problems, Classification and Regression. Classification is the problem of identifying new observations into a known set of categories. For instance, to classify whether an incoming flow is malicious or not [49] is a classification problem, which matches a flow into two categories. Different from classification, there exists no known "buckets" for the output of regression. Rather, the aim of a regression problem is to "generate" a discrete number for a new observation, based on the relationship between existing observations it discovers. For example, predicting future trend of a stock based on its transaction history [48] is a regression problem.

2.4 Training and Testing

For both classification and regression problems, machine-learning aims to figure out the relationship between input x, and output y. $x = (x_1, ..., x_p)$, often referred to as a feature vector, consists of p features that are believed to relate to y. For instance, to learn the stock price based on monthly history, the output y is the price for day_i, and the input x is the vector of daily price for the past month from day_{i-30} to day_{i-1}.

To solve a machine-learning problem, there are two phases involved in the learning process — training phase and testing phase, as depicted in Fig 2.2. In the training phase, a set of feature vectors and their corresponding output $< X_{train}, Y_{train} >$ are collected, and referred to as the "training set". The training set are then fed to a machine-learning algorithm, whose task is to discover the relationship between X and

Y, and then to generate a learned "model" to mathematically represent the relationship. The testing phase tests the quality of the learned model against a testing set $< X_{test}, Y_{test} >$, which contains observations that not excluded from in the training set. The model takes in X_{test}, and computes the estimated result Y_{est}. Its performance is evaluated by comparing Y_{est} with the ground-truth Y_{test} in the testing set.

It is commonly acknowledged that a training set that contains large enough samples and has a good coverage of diversity in the inputs features, is helpful to generate a more accurate model [54].

2.4 Machine Learning Algorithms

In Chapter 6 several popular machine-learning algorithms are adopted to improve the performance of RAPID bandwidth estimator. In this section we describe the basic ideas of their learning processes, without diving into the algorithmic details, which is out of the scope of this thesis.

ElasticNet ElasticNet [55, 56] is one of the most prominent linear-regression algorithms, which assume a linear relationship between input vector X and output Y as follows: $Y = w_o + \Sigma_{i=1}^{P} w_i x_i$. The aim of a linear-regression algorithm is to tune the linear coefficients w_i in order to minimize the model error against the training set.

RandomForest

RandomForest, AdaBoost and GradientBoost all base their learning algorithms on decision tree method. Each decision tree solves a classification function. It regards the input vector $x = (x_1, ..., x_p)$ as a p-dimensional space, partitions the space into a set of regions, each corresponds to an output y in the training set. Then x in the testing phase is mapped into a region, whose output the corresponding y in that region.

RandomForest generates a model that consists of multiple trees. Each tree model is trained with a randomly selected subset of training data.The output of the ultimate model is the average of the output of all tree models.

AdaBoost

Similar to RandomForest, AdaBoost relies on learning a number of decision tree models. However, unlike RandomForest for which each regression tree model is trained on a subset of training set, each tree for AdaBoost is to fit the entire training set and all features.

AdaBoost algorithm learns the aggregate model in a "boosting" style. Decision trees are are built iteratively — in each iteration a new tree model is built to address the "shortcomings" of existing trees. The "shortcomings" are identified by evaluating the existing model against and training set, and highlighting those

data samples that it fails to classify correctly. When the subsequent tree model is built, it focuses on fixing those miss-classified data samples.

The output of the final model is the weighted sum of the output of all tree models, the weight of each is determined during the learning process. The model produced by such "boosting" method is considered to be more accurate than RandomForest [57, 58].

GradientBoost

GradientBoost [58] also follows a boosting learning method, but with a different aim from AdaBoost in each iteration of generating tree models. Rather than focusing on fitting those miss-classified samples, GradientBoost targets at minimizing the gradient of all data samples in the p-dimensional space.

CHAPTER 3: TESTBED

This chapter describes the testbed used throughout this thesis. Section 3.1 describes the topology of our 10Gbps testbed networks. The robustness of the bandwidth estimation logic is evaluated in the presence of different degree of cross-traffic noise in Chapter 5. Chapter 8 also studies how whether RAPID is able to maintain high link utilization with different burstiness of traffic. These require the generation of cross traffic of different levels of burstiness — section 3.2 describes how this is achieved. In Chapter 8, several congestion control protocols are evaluated against different RTTs, and random packet loss rates. Section 3.3 describes the mechanisms to emulate RTT and loss emulations for TCP flows.

3.1 Topology

Fig 3.1 demonstrates the topology of the 10Gbps testbed. The core of the network consists of two HP 2900 switches with multiple 1Gbps and 10Gbps ports, both copper and fiber. The switch-to-switch path is a 10Gbps fiber path. An Endace DAG monitor is hosted on a high-performance Dell PowerEdge R720 server. It provides line-rate capture to disk of all traffic traversing the switch-to-switch path with timestamps of nanosecond resolution and 10 nanosecond accuracy.

The hosts used to run TCP flows are a pair of Dell PowerEdge R720 servers with four cores each running at 3.3GHz. The 10Gbps Ethernet Adapter NIC at the TCP sender is a PCI Express x8 Myricom NIC with the myri10ge driver, which we refer to as *NIC1*. The receiver has the same 10Gbps adapter as the sender. We also equip the receiver with another Intel 82599ES 10Gbps adapter, *NIC2*, to study the impact of different hardwares on bandwidth estimation in Secton 6.

The testbed includes additional 9 pairs of hosts (sender and receiver) that are used to generate cross traffic sharing the switch-to-switch link. The CPU speeds of these hosts range from 1.8GHz to 3.3GHz. They are all equipped with 1Gbps NICs, and one pair of these also have 10Gbps NICs.

All hosts in the testbed run RedHat Enterprise Linux 6.8 with the Linux kernel 2.6.32-641.

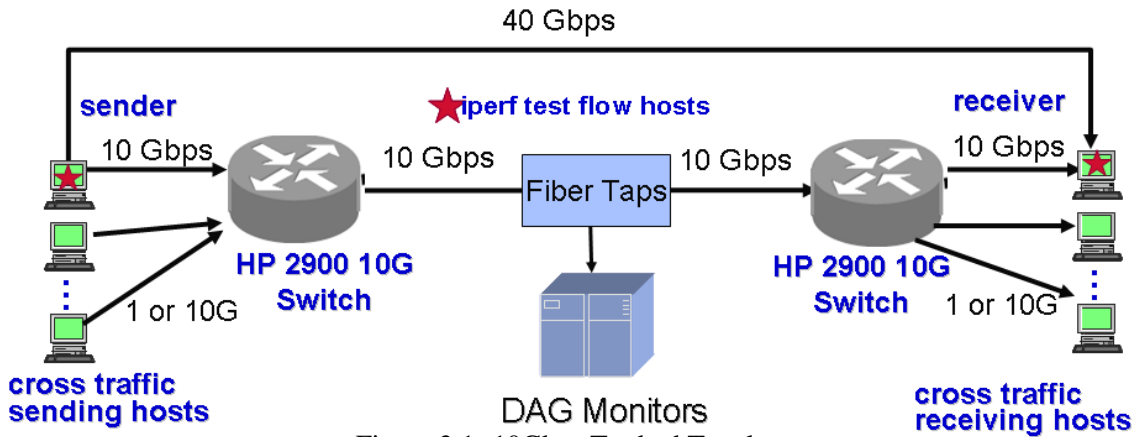


Figure 3.1: 10Gbps Testbed Topology

3.2 Generation of Traffic

3.2 TCP Test Flows

We generate TCP flows with the *iperf* [59] program, which continuously transmits data from the TCP sender (iperf client) to the TCP receiver (iperf server). We first start iperf at the receiver with the following command:

iperf -s -p listening_port -w buffersize(=BDP),

where *buffersize* is the size of the TCP buffer at the receiver. The rule of thumb is to set *buffersize* to BDP on the path to ensure that the TCP flow has the capacity to fully utilize the path. We use "-p" option to make the receiving host ready to listen to connecting request from the sender and receive data at port number $listening\_port$.

Then we start the TCP flow by running iperf at the sender:

iperf -c server_addrress -p listening_port -w buffersize(=BDP)

The iperf client establishes a TCP connection with the receiving host and transmit data via TCP. Similarly, the *buffersize* is set to BDP. The *listening_port* is the server listening port number. The "-p" option is used to notify the iperf client to transmit data to port $listening\_port$ at the receiver.

3.2 Responsive Web Traffic

In order to generate bursty traffic loads on the bottleneck link between the two switches, we used 9 pairs of sending and receiving hosts running a locally-modified version of the SURGE program for generating synthetic web traffic [60]. Our SURGE modification emulates web traffic on a network path with two

programms — a server emulation *thttpd* and a browsing user (client) emulation *thttp*. On the sending hosts we start server emulations with *thttpd*. On each of the receiving hosts we start the *thttp* client using the following command:

thttp -h 192.168.134.11 -p 5678 -b 2500 -c 4 -s experiment_time -d1 0 -d2 10.

Each host emulates 2500 web browsers with '-b' option. The '-c' option specifies the number of user sessions emulated by each browser. Each session follows an ON-OFF pattern. During 'ON' episodes, each web-user session requests a random number of files from the server — the number of bytes in these files follow a heavy-tail lognormal distribution with mean around 100KB. The '-s' option sets the length of simulation. The '-d0' and '-d1' options emulate uniformly-distributed random RTTs between 0ms to 10ms for web flows. These web flows use Cubic as the default transmission control protocol, and are responsive to other traffic sharing the bottleneck link. The average aggregate throughput from the senders with 9 pairs of machines is 2.42Gbps. The program also records the flow completion times for each TCP connection created by the traffic generator — we use it in Chapter 8 to examine how RAPID affects the completion time of these conventional TCP flows.

3.2 Non-responsive Traffic with different burstiness

An important consideration for comparing different methods for bandwidth estimation is that the cross traffic be *consistently* controlled across all experiments. On the one hand, such controlled path condition enables repeatable evaluation. On the other hand, when comparing the performance of various bandwidth estimators, this helps to ensure that they are evaluated under identical conditions, with no bias on the impact of cross traffic volume or burstiness. In particular, the cross traffic should not be responsive to the amount of bandwidth used by the TCP test flow.

BCT: Bursty traffic To eliminate the responsive behavior of each cross-traffic sender, we have each pair of web-traffic generator in the testbed running for 10 minutes, record a packet trace on the switch-to-switch link, and "replay" the trace between the same pair with the *tcpreplay* program [61] using the following command:

sudo /usr/local/bin/tcpreplay –intf1=eth0 trace.pcap,

where *–intf1* specifies the Ethernet interface to send the traffic out, and *trace.pcap* is the packet trace taken for a given pair. The aggregate average rate of the 9 replayed traffic is 2.51 Gbps, with around 289Mbps for each. In Table 3.1, we quantify its burstiness by taking a difference between the 5% and 95% of per-millisecond

Table 3.1: Cross Traffic Burstiness

Traffic	Average Throughput	5-95% of per-ms Throughput	Burstiness
BCT	2.51 Gbps	1.15 – 3.94 Gbps	2.79 Gbps
SCT	2.49 Gbps	1.78 – 3.31 Gbps	1.53 Gbps
UDP	2.47 Gbps	2.21 – 2.72 Gbps	0.51 Gbps
UNC1	3.10 Gbps	2.23 – 4.05 Gbps	1.82 Gbps
UNC2	2.75 Gbps	1.84 – 3.77 Gbps	1.93 Gbps
UNC3	3.28 Gbps	2.31 – 4.29 Gbps	1.98 Gbps

throughput samples. The burstiness of the aggregate traffic is 2.79Gbps, and we label this traffic as "BCT", which is (the most bursty cross traffic used in this testbed).

SCT: Smoother Traffic Apart from "BCT", traffic with other levels of burstiness are generated to study its impact on bandwidth estimation accuracy in Chapter 5 and on the performance of TCP RAPID in Chapter 8. We obtain a smoothed version of the "BCT' traffic by running a token bucket filter on each traffic sender using the Linux Qdisc framework:

tc qdisc add dev eth0 root tbf rate 289mbit burst 300kb.

This command applies a packet scheduler on Ethernet interface eth0 on the sending host, which caps the average sending rate at 289Mbps, and the maximum bytes of data that are allowed to be sent instantaneously as 300KB. We denote this traffic as *SCT* (smoothed version of BCT), whose burstiness is measured as 1.53Gbps.

CBR: Constant Bit-rate Traffic We also generate a constant-bit-rate traffic from each sender using iperf program:

iperf -c receiver_addr -u -b 289m,

where '-c' specifies the IP address of the iperf receiver, '-u' specifies the use of UDP as underlying transport layer protocol, and '-b' sets the constant sending rate as 289Mbps. We denote such traffic as *CBR* (constant bit-rate traffic). In Fig 3.2, we compare the burstiness characteristics of BCT, SCT and CBR cross traffic by plotting their 1ms aggregate throughput – BCT has the most bursty and CBR has the most smooth signature.

UNC egress traffic We also use three 5-minute traces collected at different times on a 1 Gbps egress link of the UNC campus network. For each trace, we run a corresponding experiment in our testbed, in which the trace is replayed concurrently by 10 cross-traffic senders (with random jitter in their start times). We label the resultant aggregate traffic aggregates as UNC1, UNC2, and UNC3, respectively. The average load of UNC1 is 3.10Gbps, UNC2 is 2.75Gbps, and UNC3 is 3.28Gbps. As shown in Table 3.1, these three traffic are more

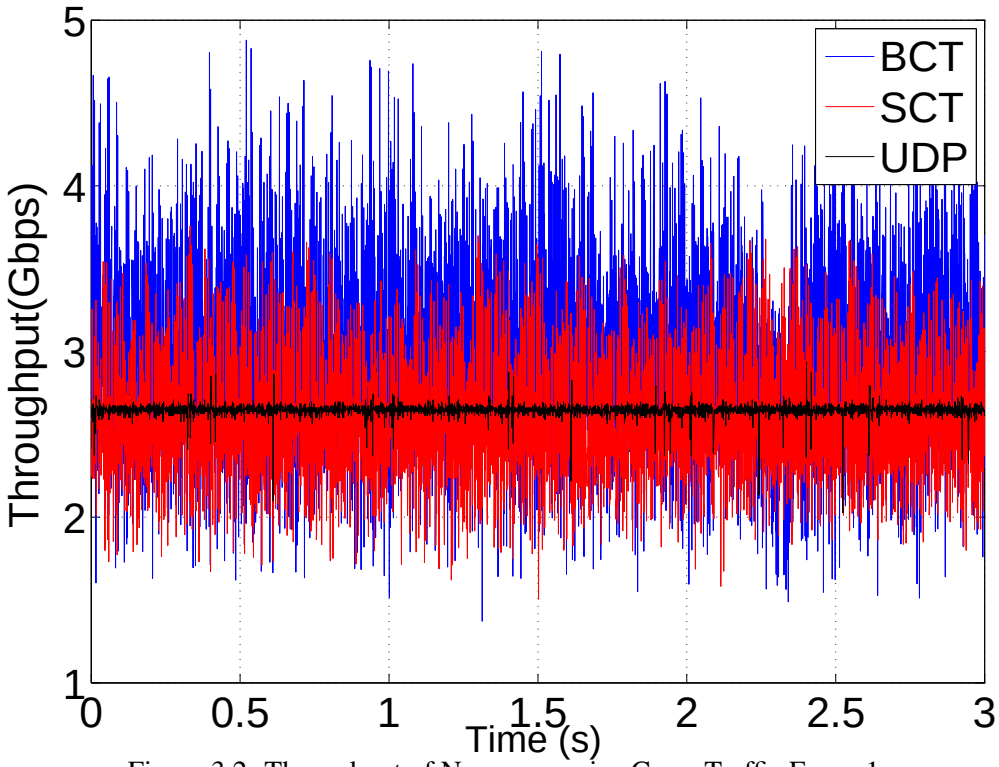


Figure 3.2: Throughput of Non-responsive Cross Traffic Every 1ms

smooth than BCT, and more bursty than SCT. They are used in Chapter 6 to evaluate the performance of a machine-learning based bandwidth estimator.

3.3 Emulating Delays and Losses

For emulating path round-trip time for non-RAPID TCP flows, we use the Linux netem Qdisc at the receiver by running the following command:

tc qdisc add dev eth1 root netem delay RTT,

where *eth1* is the Ethernet interface at the receiving host, and *RTT* specifies the round-trip time to be emulated (we emulate 5ms, 30ms, 60ms, 100ms, and 200ms in this thesis). The netem Qdisc achieves this by delaying the transmission of acknowledgements back to the sender.

For emulating random loss, we use netem on the sending host using the following command: *tc qdisc add dev eth0 root netem loss ratio*, where *ratio* is the independant loss probability to be emulated. The Qdisc randomly marks outgoing packets according to the loss probability, and drops them before they are sent to the sending NIC. We emulate a wide range of loss ratios from 10^{-2} (very high) to 10^{-8} (very low) in Chapter 8.

For RTT and loss emulation for RAPID flos, we implement an ingress Qdisc and an egress Qdisc at the receiver [1]. To emulate loss ratios, the ingress Qdisc randomly drops packets from the receiving NICs, and then hands the rest packets to the kernel; to emulate RTTs, the egress Qdisc delays the transmission of ACKs generated by the RTT time.

[1]To avoid installing multiple Qdiscs, we incorporate the functionalities of emulating RTTs and packet losses in the two Qdiscs designed for RAPID implementation

CHAPTER 4: CREATING ACCURATE INTER-PACKET GAPS

The term *inter-packet gap* (IPG) refers to the time interval between the transmission of the first byte of a packet and the end of its preceding packet. TCP RAPID relies on creating inter-packet gaps to estimate available bandwidth. To develop a proper gap-creation mechanism for RAPID, we investigate in this chapter issues related to controlling inter-packet gaps on ultra-high-speed networks and evaluate three state-of-the-art mechanisms for gap creation in our testbed.

4.1 Motivation: Need to Create Fine-scale Inter-packet Gaps

Creating precisely controlled inter-packet gaps is important in at least two prominent areas:

- *Bandwidth estimation* The crucial component in RAPID congestion control, the bandwidth estimator, relies on the accuracy of inter-packet gaps—it sends each packet at an intended rate R by controlling inter-packet gaps as $\frac{P}{R}$ (P is the packet size), and it evaluates whether R exceeds avail-bw by observing how the gap changes. Inaccurate gaps make packets probe at unintended rates, and consequently lead to incorrect conclusions about the avail-bw. In addition to RAPID, many other bandwidth estimation applications [20, 22, 21, 23], as well as other TCP protocols that rely on estimating avail-bw [25, 24, 26], require accurately generated inter-packet gaps for the same reason.

- *TCP Pacing* TCP pacing controls the burstiness of a TCP flow. Instead of transmitting a congestion window (*cwnd*) worth of packets back to back, the pacing mechanisms uniformly space out packets within a round-trip time (RTT) by controlling inter-packet gaps as $\frac{RTT}{cwnd}$. This effectively reduces packet losses, improves link utilization, and achieves better fairness among multiple flows, especially on high-speed paths and in data center networks with small buffers [62, 63]. High accuracy in creating inter-packet gaps helps to reduce traffic burstiness and thereby maximizes the performance gains of pacing.

Network speeds have increased from below 10Mbps to ultra-high speeds beyond 10Gbps, and these speed increases require highly accurate gap creation. For example, on a 100Mbps link with MTU=1500B, the minimum gap between successive packets is 120μs, but this number becomes only 1.2μs on a 10Gbps link—a 1μs lapse in sending out a packet can result in an inaccuracy of nearly 100% in the intended gap. In the context of the RAPID bandwidth estimator, that means a difference of 100% in the packet probing rate and a potential 100% error rate in the estimated bandwidth!

In this chapter, Section 4.2 identifies issues that gap-creation mechanisms face on modern end systems: insufficient timer resolution, end-system buffering, and interference from interrupts. Section 4.3 considers the pros and cons of three state-of-the-art mechanisms that have been employed for bandwidth estimation or for TCP pacing, and these three mechanisms are implemented in the 10Gbps testbed. In Section 4.6, these mechanisms' gap-creation accuracy, the involved system overhead, and the impact of the resultant gaps on bandwidth estimation will be evaluated[1].

4.2 Challenges

Increasing network link speeds impose increasingly stringent accuracy requirements on inter-packet gap creation. For example, on a 100Mbps link, a 1.3μs error in the created gap makes little difference in the intended probing rate. The same error makes a 100Mbps difference in the probing rate between 1Gbps (12μs) and 900Mbps (13.33μs). And if the link speed scales up further, to 10Gbps, the same amount of error may contribute to more than a 5Gbps difference between 10Gbps (1.2μs) and 4.8Gbps (2.5μs).

Ultra-high-speed networks — networks beyond 10Gbps — require sub-microsecond gap accuracy. However, such fine-grained inter-packet gaps are difficult to achieve with high precision on modern end systems. We identify three challenges that may prevent end systems from generating precise gaps and discuss solutions to these problems, using the open-source Linux operating system as an example for discussion.

4.2 Timing Resolution

Generally, a gap-creation process checks the system time using some time-access function f immediately before it starts to wait for an intended gap. Afterward, the process persistently calls f in order to determine whether the intended gap has elapsed or not. This process requires that time be read with a granularity even

[1]The impact of gap-creation on TCP pacing will be investigated in Chapter 9

finer than the scale of the intended gaps; otherwise, gap errors can be severe. For instance, the timing function $jiffies$ relies on a jiffy [64] counter, which advances every 1ms. In other words, $jiffies$ provides 1ms resolution. Assume that we are to create a 100μs gap and that the transmission time of the previous packet is t_0. The gap-creation process is expected to keep polling system time using $jiffies$ until it discovers a current time that is greater than $t_0 + 100\mu s$; then it will transmit the next packet. However, t will not be greater than $t_0 + 100\mu s$ unless the jiffy counter advances after 1ms; the created inter-packet gap ends up being larger than 1ms—more than ten times longer than intended. For the ultra-high-speed networks targeted in this dissertation, a sub-μs resolution is required.

Usually these processes either read system time through a user interface (such as $gettimeofday()$ or $jiffies$) or rely on timer facilities (such as $timer\_list$ and $hrtimer$); however, $jiffies$ and $timer\_list$ provide 1ms resolution, and $gettimeofday()$ and $hrtimer$ provide 1μs resolution. Unfortunately, none of these have a resolution finer than 1μs—all of these solutions are too coarse-grained for the needs of ultra-high-speed networks. The time stamp counter ($rdtsc$) [65] is a register counting CPU cycles since start time, and is able to provide high resolution timing with very low overhead. Nevertheless, with the advent of multi-core machines, great care must be taken to correct these counters when they are asynchronous across cores. Thus, the use of $rdtsc$ for timing is strongly discouraged.

4.2 Interrupts

Most widely deployed modern operating systems (i.e. Linux, iOS, and Windows) use an interrupt-driven framework to manage the use of shared computational resources. The process scheduler allocates a processor (CPU) between multiple tasks (threads and processes) by interleaving their execution based on some scheduling policy, assigning a time slot for each of them. When in the running state, a task can be interrupted, allowing control of the CPU to go to another task; this can happen when the first task's time slot runs out or when other tasks with higher priorities request the CPU. The execution of gap creation can thus be suspended unexpectedly. If the system is heavily loaded and the CPU is kept busy handling other processes, the gap-creation task may not have an opportunity to be scheduled again before the gap time elapses. The resultant inter-packet gaps are therefore unpredictable and lack high accuracy.

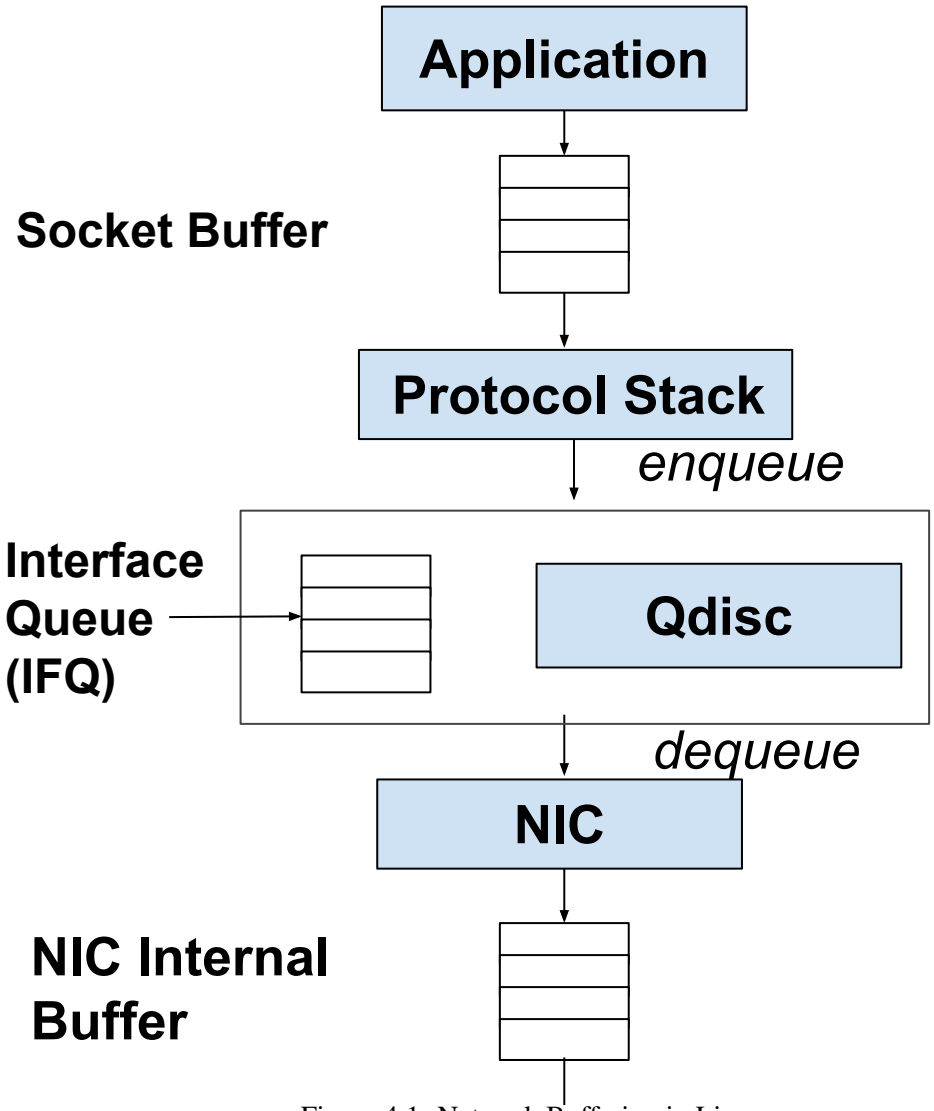


Figure 4.1: Network Buffering in Linux

4.2 Transient System Buffering

From the time they are generated by the application to the time they are transmitted on a physical link, data packets can be buffered at several places within the sending host (as illustrated in Fig 4.1):

- *Socket Buffer* Applications running in user-space generate application data and hand those data to the kernel through the send socket buffer.[2] The protocol stack will then access the send socket buffer, group data into TCP segments, encapsulate the TCP segments with IP headers, and route the packets to the proper outbound NIC based on the destination address in the IP header. Application data packets may buffer in the send socket buffer if the application's rate of data generation exceeds the processing rate of the protocol stack.

- *Interface Queue* Packets are buffered in the interface queue (IFQ) associated with each NIC before they are transmitted to the outbound link, the size of which is by default 1000 packets for the majority of Linux distributions.[3] Linux employs a Qdisc packet scheduler to schedule the transmission of packets in the IFQ. It responds to two system calls $enqueue()$ and $dequeue()$. Once a packet is ready to be transmitted, $enqueue()$ is called to place the packet into the IFQ associated with the destination NIC; the $dequeue()$ call, which is triggered immediately after $enqueue()$ by the NIC driver, determines *which* packet in the IFQ is to be transmitted, and *when* to transmit it out through the outbound NIC. The extent of buffering in IFQ depends on both the queuing discipline and the system load. By default it enforces a simple first-in-first-out rule—a packet enqueuing is coupled with its dequeuing to the NIC without intentionally holding packets in the IFQ. However, if the netem [66] Qdisc is used to emulate path delays, the Qdisc intentionally buffers packets in the IFQ, delaying their transmission by a specified duration d. The buffering is more severe with a larger d.

- *NIC Internal Buffer* In specific cases where the end system has a faster bus than the NIC link speed, packets will arrive at the NIC faster than it can send them out. To avoid dropping packets, NICs usually maintain an internal ring buffer to hold outgoing packets.[4] The transmission ring buffer does not contain actual packets; instead it contains descriptors of packets in the main memory to which the NIC

[2]The size of the socket buffer can be configured via "sysctl -w". To avoid starving the protocol stack, the applications usually ensure that the send socket buffer is set to a value that is no smaller than BDP.

[3]The IFQ buffer size can be modified with "ifconfig [interface] txqueuelen [size]".

[4]The size of the NIC internal buffer can be tuned with "ethtool -g [interface] tx [size]".

has direct access. The upper limit of the number of descriptors that can be held in the on-chip memory used for transmitting packets is platform dependent, varying from 256 to 8K [67].

When transient buffering occurs in any of queues, neighboring packets become back-to-back packets; the intended inter-packet gaps are completely destroyed.

4.3 State of the Art

Several mechanisms for controlling inter-packet gaps have been employed by existing bandwidth-estimation tools and pacing mechanisms. The majority of these are implemented only with software, but comet-TCP [68] and SoNIC [69] use hardware-assisted solutions.

Comet-TCP implements a gap-based packet scheduler on a programmable NIC Comet-NP, which maintains a large on-chip buffer space. Once a TCP packet arrives at the NIC, it buffers the packet and immediately returns an acknowledgment to the OS. The NIC shares information with the OS about inter-packet gaps, and schedules the transmission of buffered packets accordingly. The NIC is responsible for loss detection and loss recovery. It will not remove a packet from its on-chip buffer until its acknowledgment is returned from the receiver. [68] has shown that Comet-TCP accurately paces TCP traffic in the slow-start phase, and that it achieves higher throughput due to reduced packet losses with the paced traffic.

Similarly, [69] uses a software-defined NIC (SoNIC) to insert time gaps in between packets. SoNIC modifies the protocol stack, equipping every layer above with full access to the physical layer, so that the transport layer is able to control the time when each packets leaves the physical link. With the SoNIC card and its APIs, gaps can be created at the application layer with 97ps accuracy.

Despite the precision offered by these hardware-assisted approaches, they are not amenable to widespread deployment. First, these programmable NICs are not available for commodity servers. Second, in order to share gap information with the software or provide software with control to the NICs, these approaches require modifications to the kernel protocol stack. Third, these implementations are often platform dependent, and are not portable to other operating systems or NICs.

For the sake of wide deployability of TCP RAPID , we examine software-only gap-creation mechanisms. This section will explore three existing software-based mechanisms that are widely used in the area of bandwidth estimation.

4.3 Busy-waiting Loop in User-space Application

Algorithm 2 Busy-waiting(gap,$start\_ts$)

1: $elapsed = current\_ts - start\_ts$
2: **if** $gap - elapsed \geq SLEEPTHRESH$ **then**
3: sleep for SLEEPTHRESH
4: **while** $elapsed < gap$ **do**
5: $elapsed = current\_ts - start\_ts$
6: send a packet

Inter-packet gaps can be created in user-space using the logic illustrated in Algorithm 1. The application records the transmission time of the most recent packet as $start\_ts$, computes the next gap to be created, and repeatedly reads the current system time to check whether the intended gap time has elapsed or not. When the current time is ahead of the gap-elapse time by the threshold SLEEPTHRESH, the application suspends its execution to give away the control of the CPU to other processes. Once the remaining time gap is within SLEEPTHRESH, the gap-creation process enters into a busy-waiting loop in which it consistently checks whether the gap-elapse time has passed. Once the gap has passed, it exits the busy-waiting loop and sends the next packet to the protocol stack. We refer to this gap-creation mechanism as *Busy-waiting*.

The *Busy-waiting* approach has been adopted by bandwidth estimation tools including Spruce, Pathchirp, Pathload, and IGI [23, 22, 20, 21]. All of these send out data packets via UDP by calling $sendto()$ or $send()$. They retrieve $current\_ts$ by calling the interface $gettimeofday()$ and force the process to sleep by calling $usleep()$. The $SLEEPTHRESH$ is set to 2000μs for Spruce, Pathchirp, and Pathload; IGI directly enters the busy-waiting phase without suspending the process.

Discussion The challenges that are likely to impact Busy-waiting are further discussed below.

- ***Timer Resolution*** These bandwidth estimation tools use $gettimeofday()$ for timestamping. This interface offers μs resolution and promises only 10μs accuracy; not only is this too coarse-grained for ultra-high-speed networks, but $gettimeofday()$ also relies on the system wall-clock time, which may not reflect the real time if other processes modify it by calling $settimeofday()$.

- ***Interrupts*** Applications that use *Busy-waiting* run in user-space and have relatively low priority. These processes can be easily interrupted by other processes with higher priority, especially when the system is heavily loaded and when I/O intensive tasks frequently raise interrupts.

- ***Buffering*** These user applications generate UDP packets and transfer them to the socket send buffer after the gap time elapses. The sending function (e.g. $sendto()$) will block if there is no space for the packet in the send buffer. Even if the sending function succeeds, it has no control over the transmission of each individual packet as it travels from the socket buffer to lower layers in the protocol stack, and packets remain subject to buffering at several locations: the socket send buffer, the IFQ, and the NIC internal buffer.

4.3 Hrtimer-based Interrupts

The timer data structures that are offered by modern operating systems control event timing by raising an interrupt once the intended gap time has elapsed. In Linux kernel 2.6.16 and previous versions, the timer structure $timer\_list$ uses $jiffies$, which has only millisecond resolution, for timing. A high-resolution timer ($hrtimer$) with μs resolution has been available since kernel 2.6.16, and can be adopted for controlling inter-packet gaps in kernel-space. Each packet is assigned a scheduled transmission time t_s that is based on the intended time gap from its preceding packet. An $hrtimer$ with expiration time set to t_s is registered for each packet transmission. Once the timer expires, a timer interrupt is raised, triggering the callback function of the timer, which sends the packet out. This timer-interrupt mechanism is referred to as *Hrtimer*.

The *Hrtimer* approach is used for traffic pacing in kernel-space [68, 36, 70], as well as in a prototype implementation of RAPID [71]. But these differ in the protocol-stack layer in which timers are used. TRC-TCP [68] and [71] realize inter-packet gaps in the transport layer. The former [68] offers no implementation detail or any source code. The latter modifies the TCP kernel code to control when the kernel sends packets from the transport layer to the network layer. The vanilla Linux kernel transmits all packets in a TCP connection to the network layer by calling $tcp\_transmit\_skb()$ in the function $tcp\_write\_xmit()$. Instead of directly sending the packets down for IP processing, [71] computes t_s for all packets and raises an hrtimer for each. Once the hrtimer expires, the callback function then calls $tcp\_transmit\_skb()$ to pass down the actual data packet.

Pspacer/HT [72] and the Fair Queue (FQ) Qdisc [70] follow another strategy—they create gaps after the protocol stack processing. They implement a Qdisc, which shares a data structure with the protocol stack. The Qdisc computes t_s for each packet in the IFQ based on inter-packet gaps shared with the kernel protocol stack. In each $dequeue()$ call, the Qdisc initiates a $qdisc\_watchdog\_timer$ event for the transmission of the packet with the earliest t_s in the IFQ. The $qdisc\_watchdog\_timer$ structure is based on $hrtimer$, and it will not send the packet from the IFG to the outbound NIC until the timer expires.

Discussion We discuss how the three gap-creation challenges mentioned in Section 4.2 may impact the $Hrtimer$ mechanism.

- *Timer Resolution* Since kernel 2.6.16, the hrtimer has provided 1μs resolution and 10μs accuracy, which fails to satisfy the sub-μs requirements of network speeds beyond 10Gbps.

- *Interrupt* Packet transmissions are triggered by hardware timer interrupts in interrupt context, where minimal delay should be expected. However, in order to serve hardware interrupts as quickly as possible, Linux handles the critical part of the interrupt handler in interrupt context (i.e., the top half), marks the interrupt as pending, and leaves the less-urgent task in softirq context (i.e., the bottom half). The system enters softirq context before each instance of context switching and completes all pending interrupt handling. Therefore, although hrtimer interrupts the CPU at t_s, the actual transmission of packets can be deferred to softirq context, leading to unpredictable delays; furthermore, in softirq context, the packet transmission function can be preempted by interrupts with higher priority.

- *Buffering* Gaps created in the transport layer are subject to queuing in both the IFQ and the NIC internal buffer. If gaps are created with a Qdisc after the protocol stack processing, queuing at the IFQ is a necessity. After dequeuing a packet, the next packet must arrive at the Qdisc before the gap time elapses. Otherwise, the Qdisc will not find a packet transmission to schedule after the gap is achieved and the next inter-packet gap will be larger than intended. Once packets are pushed down by the Qdisc, they can still be buffered in the NIC internal buffer.

  Apart from the buffering instances mentioned in Section 4.2, the softirq context might also batch multiple pending hrtimer handlers, forcing packets to be transmitted back to back and eliminating all the intended gaps in between.

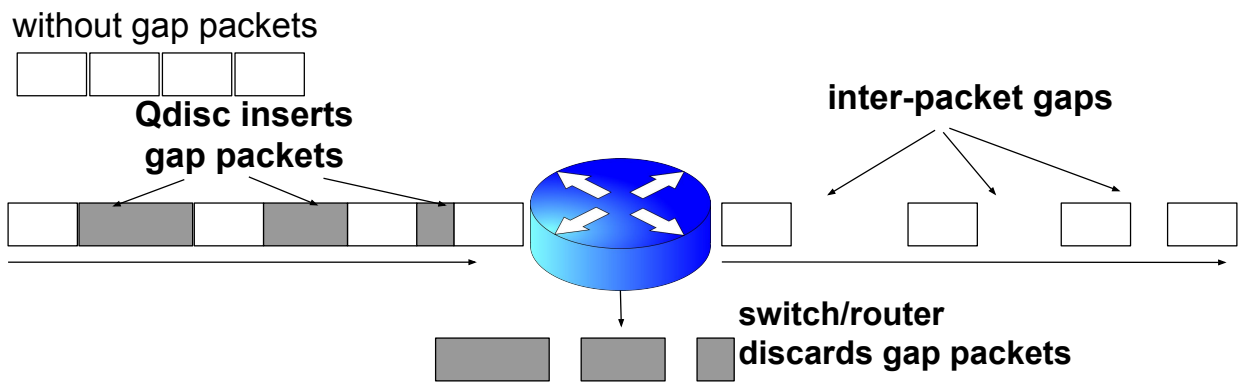


Figure 4.2: Use Gap-Packets to Create Gaps

4.3 Appropriately Assigned Dummy Gap-Packets

Instead of relying on system time, inter-packet gaps can be achieved by occupying gap time with appropriately-sized dummy "gap-packets". We refer to this mechanism as "Dummy-packet". Fig 4.2 illustrates this concept. Here a gap-packet of size $gap \times C$ is inserted between two data packets, where gap is the intended inter-packet gap and C is the link speed. Real packets and gap-packets are transmitted *back to back* at line rate C. The sizing of the "gap-packets" serves as a "byte-clock"—the timing of a packet transmission is determined by how long it took to transmit the number of bytes in the gap-packets previously sent. By precisely changing the number of bytes in gap-packets, the inter-packet gaps can be controlled precisely. Existing efforts [36, 73] adopt either Ethernet frames or UDP packets as gap-packets, whose sizes are at least the encapsulated header sizes — this imposes limitations on the minimum gap that can be created.

To avoid excessive bandwidth overhead on the network path, the gap-packets should be dropped by the router or switch that is directly connected to the sending host; the gap-packets therefore consume no extra bandwidth beyond the first hop.

The "Dummy-packet" mechanism is adopted by Pspacer [36] and Silo [73] to pace TCP traffic. They implement a Qdisc that shares inter-packet gap information with the protocol stack. The Qdisc calculates the scheduled transmission time t_s for each packet in the IFQ. Upon every $dequeue()$ call, it transmits a gap-packet and a real packet alternately to the NIC.

Pspacer and Silo use different types of packets as gap-packets. Pspacer uses IEEE 802.3x PAUSE frames, which were originally used in Ethernet flow control to pause data transmission. When a receiving Ethernet port is overloaded, it can send a PAUSE frame to notify the sending port to halt transmission for a determined

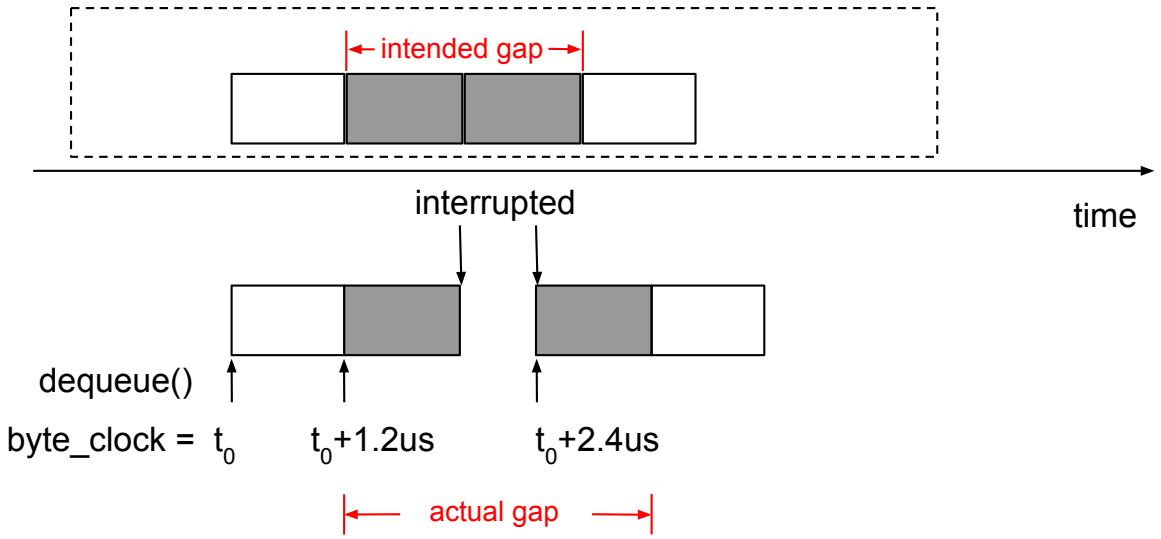


Figure 4.3: Inaccurate Gap Creation Using Dummy-packet

duration specified in the PAUSE frame header. Pspacer sets the pause time to zero in the frame headers, so that the directly connected Ethernet port does nothing but drop the frames. In Silo, UDP data packets play the role of gap-packets. Silo sets their destination MAC address to that of the source MAC, which is then discarded by the receiving port.

Discussion

- *Timer Resolution* The *Dummy-packet* mechanism does not rely on any system clock, but rather on a "byte-clock"— the gap time is clocked by the number of bytes in the gap-packets, with a resolution of $\frac{1Byte}{C}$. It is worth noticing that this number scales inversely in proportion to link speeds. The timing resolution is 80ns at 100Mbps, 8ns at 1Gbps, and 0.8ns at 10Gbps — the granularity of this byte-clocking actually becomes finer with higher link speeds, a very desirable property when scaling gap-creation to ultra-high speeds.

- *Interrupt* The *Dummy-packet* approach is usually implemented with a Qdisc. The Qdisc runs in kernel-space with higher priority than *Busy-waiting*, but the gap-creation thread can still be interrupted by tasks with higher priority.

  When using the *Dummy-packet* mechanism, the key to gap accuracy is that the physical link must continue to be occupied by gap-packets or data packets without idle periods. However, interrupts may cause the physical link to go idle, creating unexpected larger gaps. Fig 4.3 illustrates the above-mentioned phenomenon. Assume the Qdisc is to create a gap by inserting two MTU-sized

(MTU=1500B) gap-packets before the transmission of the next data packet, and that each gap-packet occupies 1.2μs link time. The Qdisc transmits the first gap-packet out to the NIC and immediately advances the byte-clock by 1.2μs. If the gap-creation process is interrupted by other tasks and $dequeue()$ fails to be scheduled until δ ($\delta > 1.2\mu s$) later, the physical link is left idle by $\delta - 1.2\mu s$, thus resulting in a larger inter-packet gap than intended.

- *Buffering*

  As in the *Hrtimer* approach implemented using a Qdisc, the next data packet must be "ready" in the IFQ when the scheduler decides that it is time to transmit it. Therefore, buffering at IFQ facilitates the gap-creation process, and it thus becomes necessary for the protocol stack to enqueue the next packet and buffer it in the IFQ before the gap time elapses.

  Once packets are dequeued by the Qdisc scheduler to the NIC, the created inter-packet gaps are not damaged by queuing. Even though packets may get buffered in the NIC internal buffer after they are dequeued by the Qdisc, the intended gaps are still preserved. Even in the queue, real packets are interleaved with appropriately-sized gap-packets, maintaining the intended gap time once they are sent out of the NIC at link speed.

4.4 Goal

The goal of this chapter is to implement and evaluate the state-of-the-art gap-creation mechanisms discussed in Section 4.3 in the ultra-high-speed testbed, where both gap accuracy and the impact of resultant gaps on bandwidth estimation. The results of the evaluations reported in this chapter inform the choice of the gap-creation mechanism to be used with RAPID.

4.5 Methodology

We conduct evaluations of the three state-of-the-art gap-creation mechanisms discussed in Section 4.3, to examine how the three challenges to gap creation (timing resolution, interrupts, and end-system buffering) affect gap creation on real-world ultra-high-speed networks.

The gap-creation mechanisms are implemented in experiments designed in the 10Gbps testbed to investigate the following questions.

1. As discussed in Section 4.2, in bandwidth estimation, the accuracy of each individual inter-packet gap is crucial to ensure that each packet probes at its intended rate. We generate inter-packet gaps to achieve probing rates from 100Mbps to 10Gbps and ask: what is the per-gap accuracy achieved by each of these mechanisms?

2. Then we consider gap creation in the context of bandwidth estimation. We generate multi-rate p-streams that share the bottleneck link with constant-bit-rate traffic and use the p-streams to estimate avail-bw. Apart from gap-creation accuracy, bandwidth estimation is also affected by noise from bursty cross-traffic and by inaccurate receiver-side timestamping. In this chapter, the impact of the latter two factors is reduced by creating very smooth cross-traffic and by timestamping packets with the DAG monitor before they reach the receiver. We ask: what is the impact of noise in gap creation alone on avail-bw estimation results?

3. Gap creation is usually a component of a more complicated system, such as bandwidth estimation applications or congestion control protocols. The gap-creation process should not take excessive computational resources or starve other processes. We generate traffic with inter-packet gaps between adjacent packets using the three mechanisms, and ask: how much CPU utilization does each of the three mechanisms require during execution?

4. Gap creation faces greater challenges with an increasing number of concurrent flows in which multiple gap-creation instances compete for computational resources. We increase the number of concurrent flows N in the sending host, and ask: to what extent can N be increased so that these mechanisms maintain their gap-creation accuracy?

Next we describe how we implement the three gap-creation mechanisms, *Busy-waiting, Hrtimer,* and *Dummy-packet*, in our testbed.

Implementing Busy-waiting We realize *Busy-waiting* using pathChirp [22] source code, which is implemented as a user-space application and generates UDP traffic between the sender and the receiver. For each experiment, we modify its p-stream transmission function to specify the length of p-streams as well as each individual intended gap required for the given experiment.

Implementing Hrtimer We adopt the mechanism used in [72] and implement a Qdisc that transmits packets by raising $qdisc\_watchdog\_timer$ interrupts. The data structures used in the Qdisc are illustrated in Fig. 4.4.

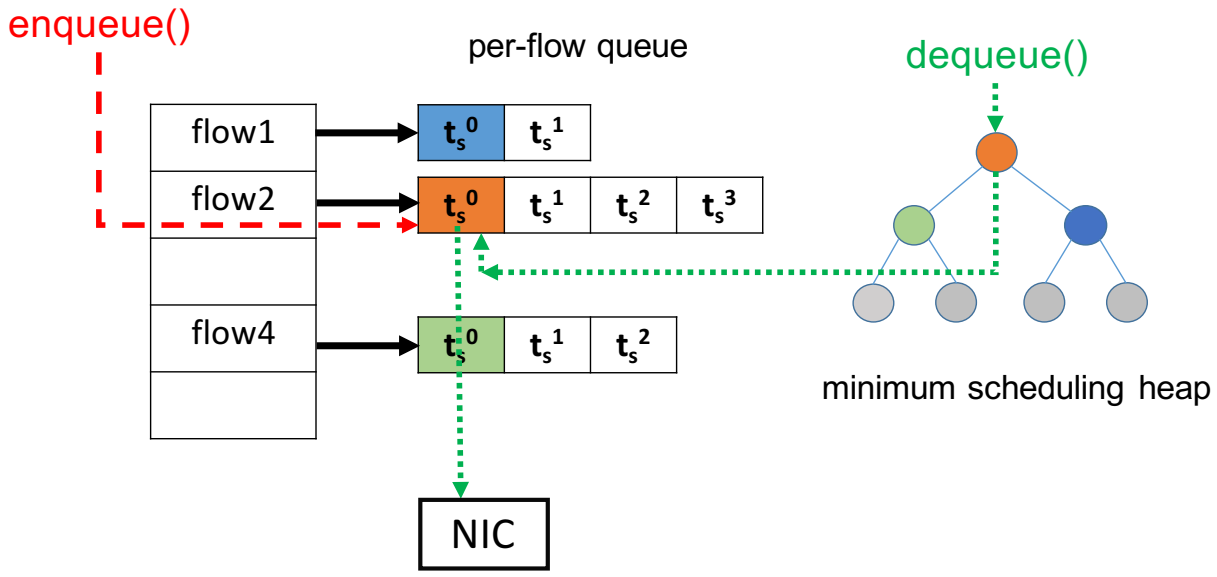


Figure 4.4: Qdisc Packet Scheduler

E_S maintains a FIFO queue for each flow and a hash table to organize multiple per-flow queues. Upon every $enqueue()$ call, it assigns each packet a scheduled send time t_s; locates its corresponding per-flow queue in the hash table by hashing its 4-tuple (source_ip, source_port, dest_ip, dest_port); and enqueues it to the tail of the queue. To interleave the transmission of multiple flows, the Qdisc maintains a minimum scheduling heap. The elements of the heap are the packets at the head of each per-flow queue, and the heap is sorted according to t_s. Thus, the packet on the top of the heap always has the earliest transmission time among all packets in the IFQ.

Upon each $dequeue()$ call, the Qdisc schedules the transmission of the packet on the top of the heap by resetting a $qdisc\_watchdog$ timer to expire at its t_s, removes the packet from its corresponding per-flow queue, and re-sorts the scheduling heap. Once the timer expires, the packet is pushed down to the physical link.

As discussed in Section 4.3.2, buffering at the IFQ is crucial for creating accurate gaps with a Qdisc. In other words, data packets must arrive at the IFQ at a faster rate than the Qdisc sends them. To achieve this, we set the send socket buffer as a larger value than BDP, so that the traffic-generating applications (iperf in our experiments) are able to continuously generate data packets to the protocol stack without starving the IFQ.

Implementing Dummy-packet

Based on E_S, we implement a Qdisc scheduler E_{pause} to occupy gap time using IEEE 802.3x PAUSE frames as dummy gap-packets. This maintains a *byte-clock* that keeps track of the elapsed time on the physical link, which is occupied by all PAUSE and real packets transmitted since the initiation of the Qdisc. The scheduled time t_s of each packet refers not to system time but to the byte-clock.

Algorithm 3 gives the pseudocode of how E_{pause} interleaves the transmission of real packets and PAUSE frames. Upon every $dequeue()$ call, instead of raising an *Hrtimer* interrupt, it checks whether t_s of the heap top is due according to the byte-clock. If the answer is yes, then it transmits the packet; otherwise, it transmits a PAUSE frame to occupy the time gap between t_s and the byte-clock. The size of the PAUSE packet must be no larger than the path MTU; it must be no smaller than 88 bytes[5]. Every time a real packet or a gap-packet is dequeued, the byte clock is advanced by the amount of time it takes to transmit that packet on the link. The effect of Algorithm 3 is the insertion of PAUSE frames of size $gap \times C + (< 88bytes)$ before the transmission of the next data packet.

For both the *Hrtimer* and *Dummy-packet* approaches, we rely on the *iperf* program running in user-space to generate TCP traffic between two machines. E_S and E_{pause} then create inter-packet gaps between adjacent TCP packets sent to the NIC IFQ.

[5]A minimum sized Ethernet frame includes at least 14 bytes of header, 46 bytes of minimum payload, and 4 bytes of CRC(cyclic redundancy check) at the end of the packet. The PAUSE frame used here requires extra 4 bytes for frame check sequence (FCS), and 20 bytes to account for the hardware preamble and indicating inter-packet pause time.

Algorithm 3 E_{pause} :: dequeue()

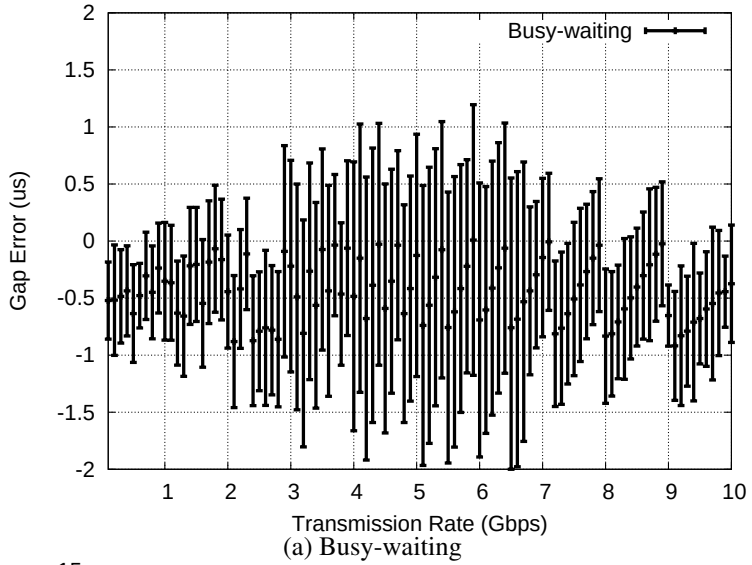
```
1: next_ts = heap_top.ts
2: if next_ts > byte_clock then
3:     dq_bytes = C × (next_ts - byte_clock)
4:     if dq_bytes > MTU then
5:         dq_bytes = MTU
6:     elif dq_bytes < MIN_PAUSE then
7:         dq_bytes = MIN_PAUSE
8:     dequeue a PAUSE frame of size dq_bytes
9: else
10:    dq_bytes = sizeof(heap_top packet)
11:    dequeue heap_top
12: byte_clock += dq_bytes / C
```

4.6 Evaluating State-of-the-art Mechanisms

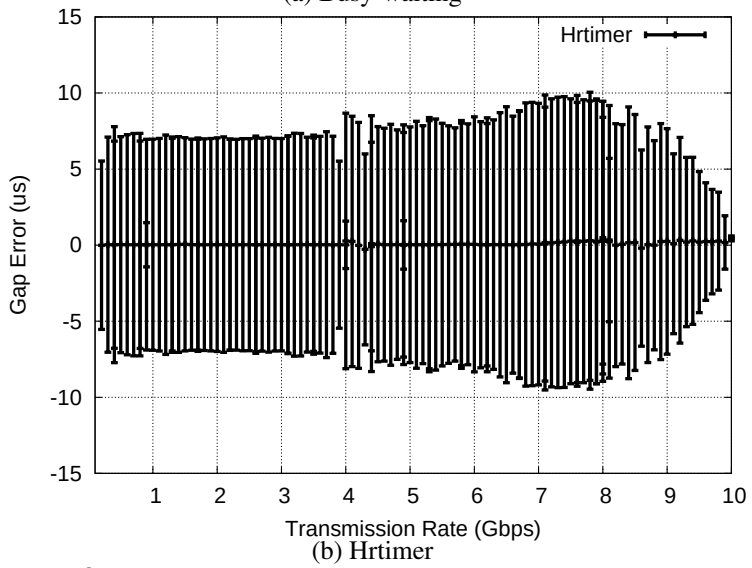
In this section, we conduct experiments in our 10Gbps testbed to answer the questions raised in Section 4.5 and present the experimental results. To ensure that gap-creation processes are not affected by other applications, I/O intensive or computation-intensive tasks are turned off on the traffic-generating host. Section 4.6.1 evaluates the accuracy of each individual gap; Section 9.2 and Section 4.6.2 examine the impact of the gaps produced in the context of bandwidth estimation; and Section 4.6.3 explores how much system overhead is involved in creating gaps with the three mechanisms. Then we increase the load on the sending host by increasing the number of gap-creation instances running concurrently; we show how gap accuracy varies with the increasing system load in Section 4.6.4.

4.6 Gap Accuracy

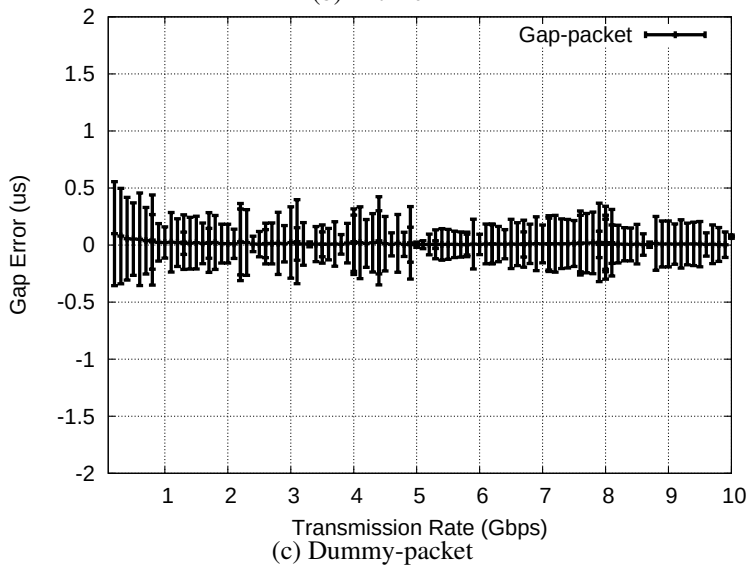
To collect a wide range of gap samples, packets are transmitted between the 10Gbps NICs on the sender and the receiver at 100 uniformly spaced rates from 100Mbps to 10Gbps, stepped by 100Mbps. The traffic at each rate lasts for at least 10s. Each transmission rate R corresponds to the intended inter-packet gap of $\frac{P}{R}$, where P is the packet size of 9038 bytes in our 10Gbps testbed. In other words, inter-packet gaps ranging



(a) Busy-waiting



(b) Hrtimer



(c) Dummy-packet

Figure 4.5: Gap Creation Errors

The mean and standard deviation of gap-creation errors are plotted as error bars while gaps are intended for the probing range from 100Mbps to 10Gbps stepped by 100Mbps. *Dummy-packet* approach yields highest gap-creation accuracy, with mean error close to 0 and less than 0.5μs deviation across all probing rates.

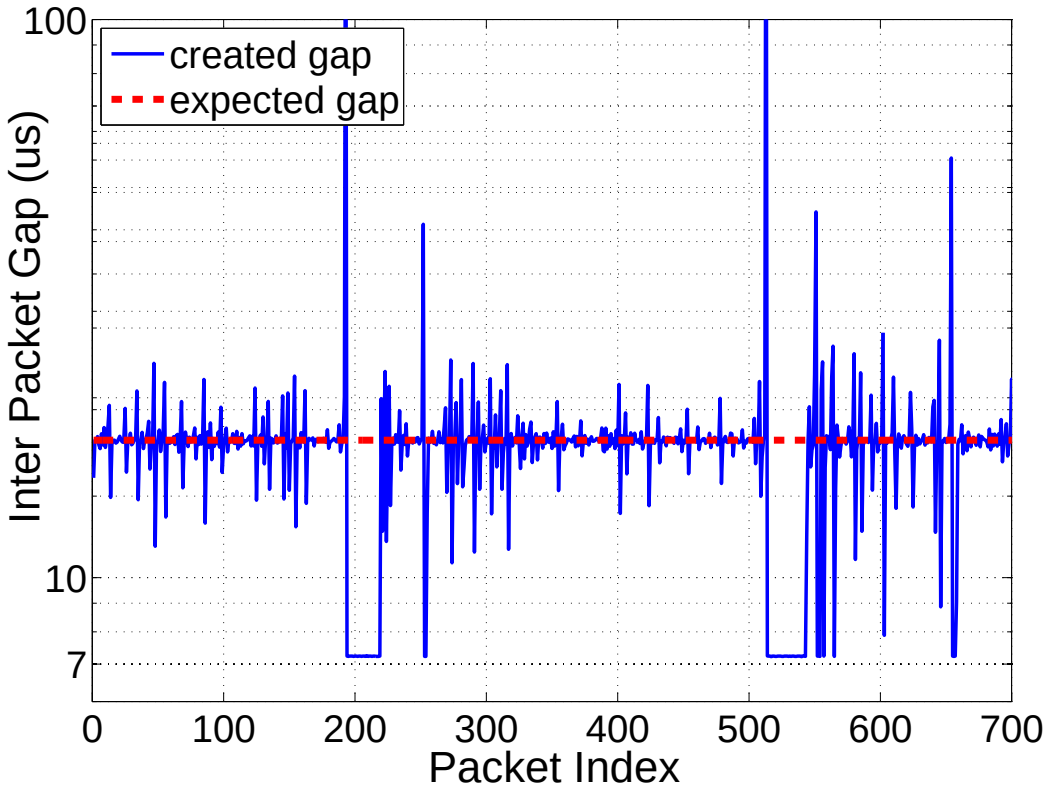


Figure 4.6: Gaps with *Hrtimer* at 4Gbps

Due to delayed interrupt handling, packet-transmissions are queued, resulting in unexpected large inter-packet gaps over 100μs immediately followed by extremely small inter-packet gaps of around 7.2μs.

from 7.2μs (when sending at 10Gbps) to 720μs (when sending at 100Mbps) are generated. For each rate, we calculate the metric of gap error as $g_o - g_i$, where g_i is the intended gap and g_o is the actual gap observed in the packet trace taken after the first hop on the switch-to-switch path in Fig 3.1.

The three figures in Fig. 4.5 illustrate the gap-creation accuracy achieved by the three mechanisms, respectively. The x-axis corresponds to the 100 different transmission rates, and the y-axis corresponds to gap-errors. Note that Fig 4.5(b) is plotted with a larger y scale than other figures for better presentation of the scale of yielded gap-error. The midpoint of the candlestick shows the *mean_error*, which is the mean of all gap errors for that transmission rate. Each candlestick plots the range of (mean_error-std_dev,mean_error+std_dev), where std_dev is their standard deviation. Shorter candlesticks indicate a higher degree of predictability and reliability in the created gaps.

According to Fig. 4.5, *Busy-waiting* limits gap errors within 2μs for all rates. It is worth noting that it persistently creates smaller inter-packet gaps than intended (with mean gap-errors always below 0). This is because the $gettimeofday()$ function provides only μs resolution, and the application in effect uses the

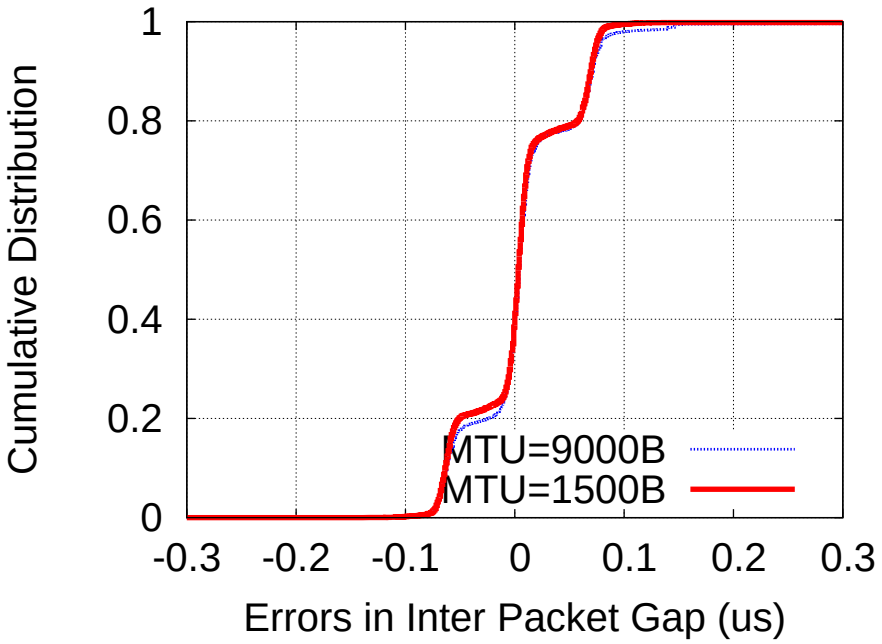


Figure 4.7: Gap Errors with Different MTU Sizes

Cumulative distribution of all gap-error samples are plotted when probing from 100Mbps to 10Gbps. Two MTU sizes yield comparably high-accuracy inter-packet gaps on 10Gbps links.

floor of g_i. Section 4.6.4 will show that when the system is heavily loaded, *Busy-waiting* may defer packet transmission by over 100μs, thus creating highly inaccurate inter-packet gaps.

The *Hrtimer* approach produces the least accurate gaps. Although for each rate the mean error is below 0.1μs, individual gaps can deviate by more than 20μs. Fig. 4.6 plots the time-series of the achieved gaps when transmitting packets at 4Gbps. The noisy inter-packet gaps are the result of delayed interrupt handling in softirq context, where the processor traverses through a list of pending interrupt handlers and transmits multiple packets in a batch. The "spike" over 100μs corresponds to the first interrupt handler batched in the softirq context, and the following "dips" around 7μs correspond to handlers processed immediately afterwards.

The *Dummy-packet* approach generates the most accurate and reliable inter-packet gaps , with less than 0.4μs errors and less than 0.5μs deviation across all transmission rates. We also experiment gap-creation at the 100 intended rates when using a smaller MTU size of 1500 bytes. Fig 4.7 shows that the smaller packet size is able to achieve highly comparable gap-accuracy with jumbo packets.

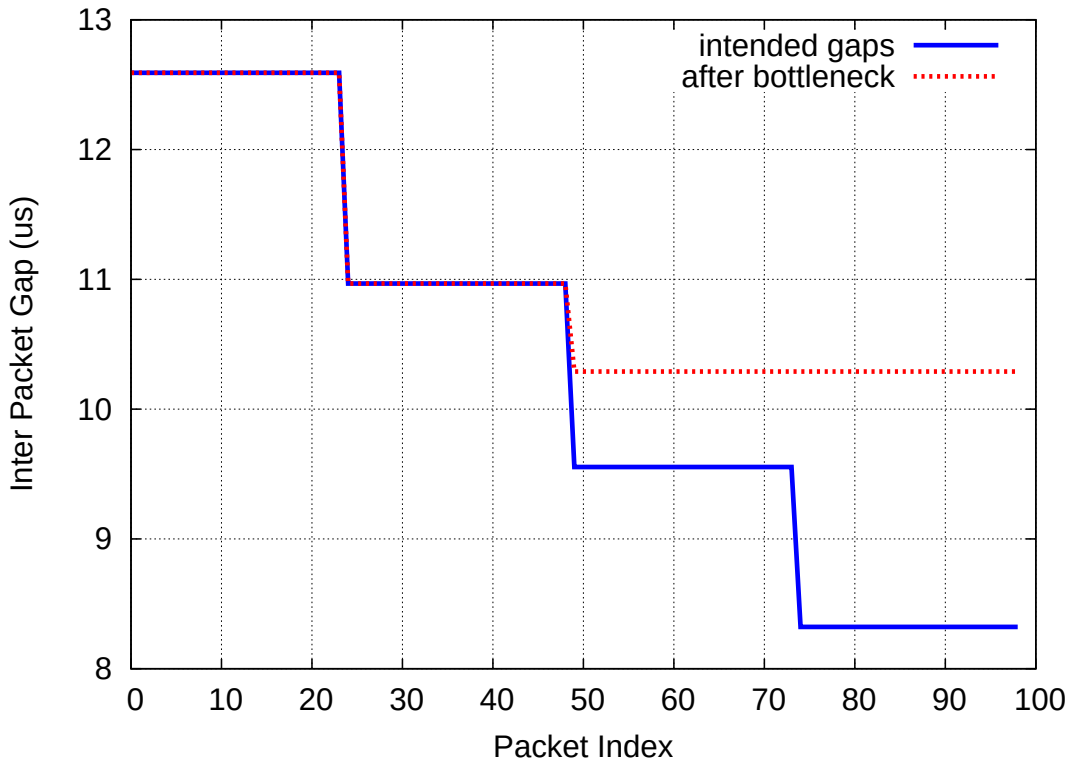


Figure 4.8: A Packet Group (Noise-free)

The blue line plots the intended inter-packet gaps created by the sender. The red line plots the expected inter-packet gaps that should be observed at the receiver. When sharing the path with 3Gbps cross traffic, these p-streams ought to yield bandwidth estimation as the third probing rates of 6.59Gbps.

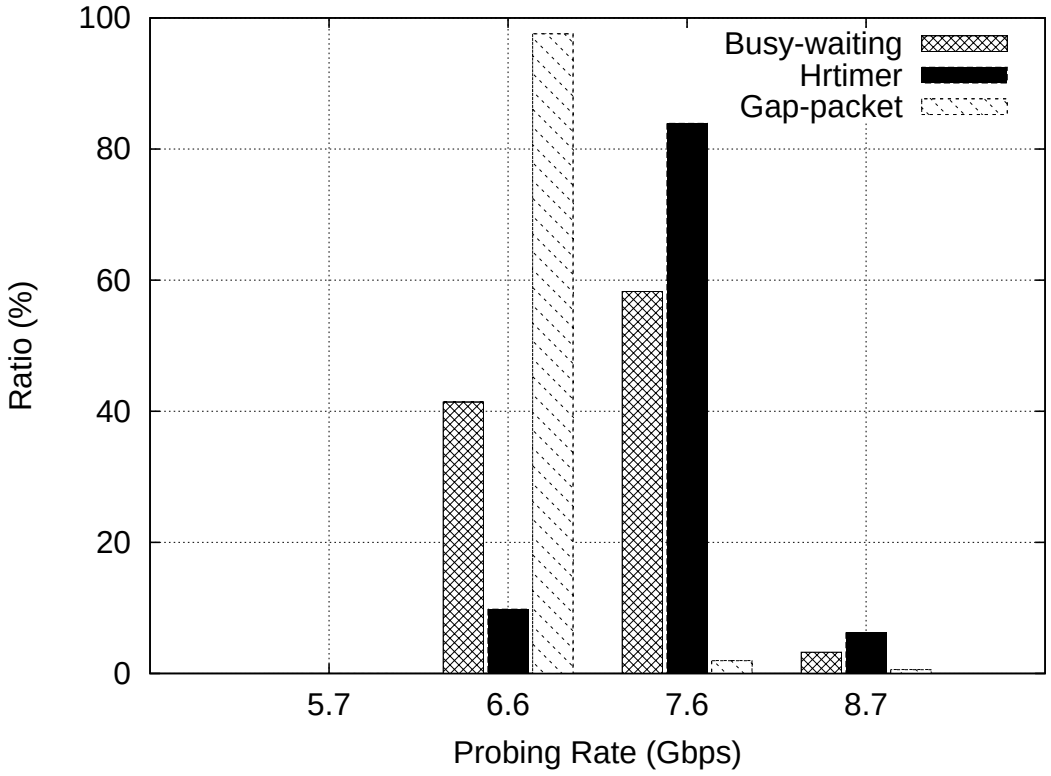


Figure 4.9: Estimated Available Bandwidth

Each bar represents the percentage of p-streams which yield bandwidth estimation at the corresponding probing rate on the x-axis.

4.6 Bandwidth Estimation

The three gap-creation mechanisms are used to to send out p-streams to estimate avail-bw, and we investigate how each of the resultant gaps impacts estimation accuracy. Multi-rate p-streams are considered, and the estimation algorithm in Section 2.3 is adopted. It estimates avail-bw as the probing rate after which g_r becomes consistently larger than g_s.

For each mechanism, we generate 5000 p-streams probing at four discrete rates within each p-stream (5.7Gbps, 6.6Gbps, 7.6Gbps and 8.7Gbps). Their average probing rates are 7Gbps. They share the switch-to-switch bottleneck link with 3Gbps cross traffic. The intended inter-packet gaps the sender creates, and the expected inter-packet gaps observed at the receiver, are plotted in Fig 4.8. If (1) inter-packet gaps (g_i) are created accurately, and (2) the cross traffic is absolutely smooth, with identical inter-packet gaps between adjacent packets, we expect to observe inter-packet gaps in the DAG trace (g_o) after the bottleneck link, as seen in the dashed line in Fig 4.8. Assume that (3) the receiver timestamps packet arrivals with 100% accuracy, and that these p-streams are expected to give estimates of 6.6Gbps, which is the highest probing rate that does not exceed avail-bw (7Gbps).

However, conditions (2) and (3) can be easily violated: Internet traffic tends to be very bursty, and packets may be buffered on the receiver side before they are timestamped by the protocol stack. In order to effectively examine the impact of (1) on bandwidth estimation alone, the influence of the other two factors must be eliminated. We minimize the impact of bursty cross-traffic by using the implementation of *Dummy-packet* to generate paced traffic at 3Gbps with MTU=1500B (Fig 4.7 has shown that the Dummy-packet mechanism is able to create very smooth traffic).[6] If the receiver accurately timestamps packet arrival, the recorded packet arrival time should be the same as the time each packet arrives at the receiving NIC. To reduce the impact of inaccurate receiver-side timestamping, we compute inter-packet gaps from the DAG traces captured after the bottleneck link and use those inter-packet gaps as g_r.

Fig. 4.9 plots the histogram of bandwidth estimates for all three mechanisms. For the four probing rates in p-streams on the x-axis, we show the percentage of p-streams that estimate avail-bw as that rate on the y-axis. We find that the gap-creation mechanism that creates the most accurate gaps also produces higher

[6]We use smaller-sized packets to generate cross-traffic, which reduces burstiness more efficiently than larger packets of 9000B. The sending host in our testbed fails to generate smooth traffic at rates higher than 3Gbps; a smaller packet size leads to smaller inter-packet gaps and a larger number of packets to be transmitted within the same amount of time, triggering excessive system overhead.

Table 4.1: CPU Utilization with Gap-creation

The three gap-creation mechanisms are creating gaps for packets transmitting at 10Gbps and 5Gbps, with 7.2μs and 14.4μs inter-packet gaps respectively.

CPU util%	7.2μs (10Gbps)		14.4μs (5Gbps)	
	mean	std	mean	std
No gap-creation	38.6	0.34	18.2	0.18
Busy-waiting	38.4	0.45	18.4	0.16
Hrtimer	54.1	0.48	20.1	0.19
Dummy-packet	42.1	0.42	19.2	0.19

$ABest$ accuracy. *Dummy-packet* outperforms the other two mechanisms, yielding the highest fraction of p-streams (97.5%) that offer correct results. *Hrtimer* leads to the most severe overestimation—more than 89.2% p-streams estimate avail-bw at higher than ground-truth levels.

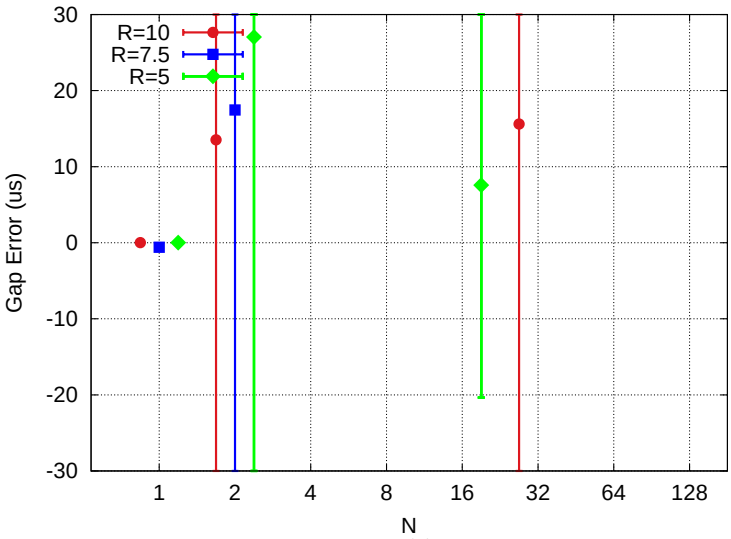
4.6 System Overhead

To compare the system overhead produced by the three gap-creation mechanisms, we perform the comparisons with only one gap-creation instance running at a time. In this experiment, we focus on CPU utilization, which is the main bottleneck for modern end-systems transmitting at ultra-high speeds over 1Gbps [74]. We first send traffic at 5Gbps and 10Gbps with the iperf program without creating inter-packet gaps . Then we transmit paced traffic at these two rates and examine the increase in CPU utilization. CPU utilization is measured with the *top* command while the gap-creation process is running. Each experiment lasts for 120s, and is repeated 5 times, and highly repeatable results are obtained. We list their average and standard deviation of CPU utilization in Table 4.1.

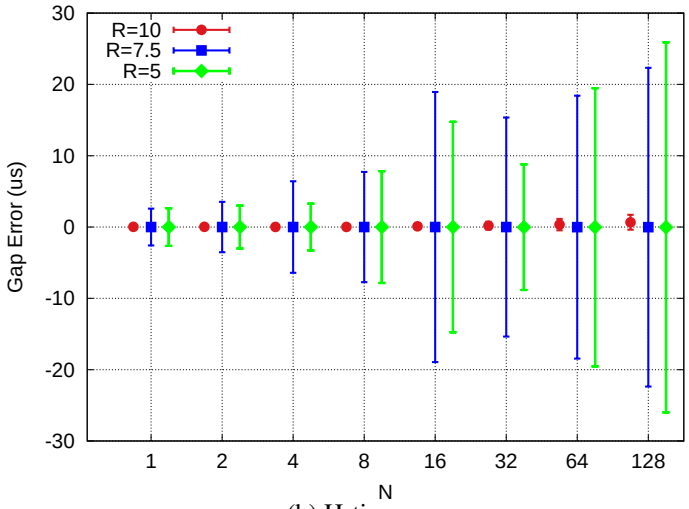
When transmitting at 5Gbps, the three mechanisms consume comparable CPU resources; see the first row in Table 4.1 (sending packets without gap creation). All three gap-creation processes increase CPU utilization by less than 2%. Once we increase traffic load to full link speed, the *Busy-waiting* approach's overhead is similar to the overhead with no gap-creation process running. The *Dummy-packet* mechanism slightly increases CPU utilization (by less than 4%). The *Hrtimer* mechanism triggers significant system overhead due to excessive timer interrupts, increasing CPU utilization by 15%.

4.6 Stress Test

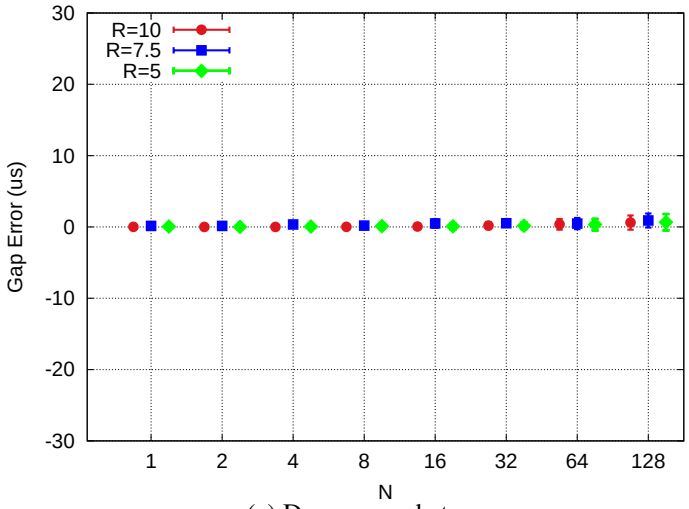
Gap-creation processes may share computation resources and bandwidth resources with other applications hosted on the same server. Furthermore, a server may have multiple concurrent flows that are active at the same time—it has to create accurate inter-packet gaps within each flow and interleave packet



(a) Busy-waiting



(b) Hrtimer



(c) Dummy-packet

Figure 4.10: Gap-error with N Flows

With N concurrent flows, each flow is sending packets at rate $\frac{10Gbps}{N}$ by finely-controlling inter-packet gaps within each. *Busy-waiting* yields significant gap errors, with many data points beyond the range of y axis. With increasing N, the *Dummy-packet* approach introduces least overhead and meanwhile maintains gap-creation accuracy.

transmission across all flows. More concurrent flows require more computational overhead. For example, if packets are created using the *Busy-waiting* mechanism, the server has to create a user process for every flow, and every process keeps polling the system time, consuming considerable CPU resources. When packets are created with a Qdisc, computational overhead still increases; a larger number of flows increases the height of the scheduling heap and thus increases scheduling complexity. Accurate inter-packet gaps cannot be guaranteed if the gap-creation process fails to obtain adequate computation resources.

In this section, we conduct stress tests by increasing the number of concurrent flows N to examine how the three gap-creation mechanisms scale with increasing system load. For each mechanism, we ask: to what extent can we increase N while maintaining gap-creation accuracy? We force these N instances to share a single processor in our multi-core machine (see Section 3.1 for machine specifications). These instances generate aggregate traffic at rate R, each instance creating inter-packet gaps of $\frac{N \times P}{R}$. We increase N from 1 to 128, calculate gap error for packets from all N flows, and conduct experiments for different values of R.

According to Table 4.1, with even a single flow the computational overhead increases with R; for all three gap-creation mechanisms, the CPU utilization nearly doubles (from around 20% to over 40%) when the flow rate is increased from 5Gbps to 10Gbps. To study how the aggregate transmission rate R impacts gap-creation accuracy with multiple flows, we experiment with different settings of R at 10Gbps, 7.5Gbps, and 5Gbps.

For each flow, we obtain the inter-packet gaps observed from the DAG trace and compute gap errors. In Fig. 4.10, we plot the mean gap error of all flows, and their deviation, for each of the gap-creation mechanisms.

Busy-waiting For Busy-waiting, one gap-creation instance corresponds to one user-space process, which keeps using CPU during busy-waiting even when it is not transmitting packets. With a larger N, more CPU-intensive processes compete for CPU resource, leading to a higher risk of being interrupted for each of them. Besides, larger number of instances consume significant system overhead due to frequent context switching. CPU utilization reaches 100% once $N \geq 4$, generating gaps with over 100μs errors. Once N exceeds 32, the gap errors become even greater, with large deviations of up to 10 seconds.

Hrtimer Instead of establishing N busy-waiting user processes, the *Hrtimer* and *Dummy-packet* approaches interleaves packets from the N flows in a single interface queue, and control inter-packet gaps all together by assigning dequeue time for each packet using Qdisc interface. Even with *Bursy-waiting*, packets are also processed through a default first-in-first-out Qdisc. *Hrtimer* and *Dummy-packet* simply replace the default

queuing discipline and implement their own gap-aware queuing disciplines, without introducing significant system overhead. For both the *Hrtimer* and *Dummy-packet* mechanisms, we observe no significant increase in CPU utilization with large N values.[7]

With the $Hrtimer$ mechanism raising timer interrupts each time it sends a packet, the handling of these interrupts will be not executed immediately in interrupt context, but will be deferred to softirq context. Therefore, the handling of interrupts may be delayed, and interrupts may get buffered in softirq context, resulting in a higher degree of variance in the created gaps. In Fig. 4.10(b), the mean gap errors are less than 1μs even with $N = 128$, but the deviation increases beyond 20μs once N reaches 16. Such observation agrees with what we found in Fig. 4.5(b). We also observe in Fig. 4.10(b) that the deviation of gap errors increases with N. This is because, with a fixed R, a larger N requires a larger intended inter-packet gap for each flow, allowing more room for variance.

However, Fig. 4.10(b) shows that the *Hrtimer* mechanism generates highly precise inter-packet gaps —less than 0.4μs deviation across all N values when $R = 10Gbps$. This is an artifact of our experimental design: with $R = 10Gbps$, the outbound link is always fully occupied *without* any idle periods. Packets from the N flows get interleaved when they are enqueued by the Qdisc. We found in the DAG trace that on more than 99.6% occasions, packets from a specific flow are spaced by (N-1) packets from other flows, yielding highly accurate gaps.

Dummy-packet As shown in Fig. 4.10(c), for all three R values, the "Dummy-packet" mechanism produces gaps with high accuracy, even when N = 128; the mean and deviation of gap errors are consistently less than 3μs.

We re-plot the gap errors from Fig. 4.10(c) with a smaller range in the y-axis in Fig. 4.11. It is noticeable that the average gap error increases with N. However, we find that these errors are not caused by erroneous gap creation, but by the instrumental inaccuracy of DAG timestamping. In fact, *the DAG trace reports larger errors if we observe gaps at a larger timescale, even if each individual inter-packet gap is created accurately*. To confirm this, we generate a 10Gbps flow without creating inter-packet gaps and take a DAG trace on the switch-to-switch path. Then we calculate the inter-packet gaps g_o observed from the trace between every N packets—we extract the timestamps of the ith and $i + N$th packets from the DAG trace and calculate g_o as

[7] The heap scheduling complexity is $O(\log n)$, for which $N = 128$ is too small to observe any significant increase in CPU utilization. However, according to the survey in [75], the number of concurrent flows running on a single core of commercial server seldom exceeds 100.

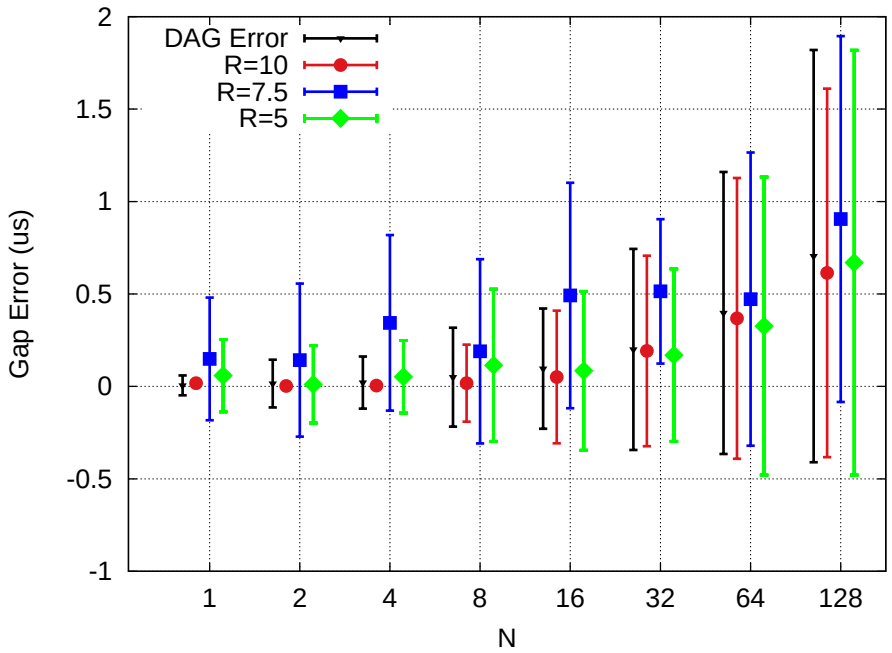


Figure 4.11: Gap Errors with Dummy-packet

Gap-error created by *Dummy-packet* approaches agrees with with instrumental inaccuracy of DAG timestamping.

the difference between the two timestamps. With this flow of full link speed, packets are expected to appear back to back in the trace, with $7.2304\mu s$ gaps between successive packets. Thus, for each N, inter-packet gaps of $g_i = N \times 7.2304\mu s$ are expected. For different N values from 1 to 128, we calculate gap errors as $g_i - g_o$, and we depict the mean and deviation as the first bar in Fig. 4.11. When $N = 1$, the gap-errors are negligible—less than 0.06μs—which confirms that packets are indeed transmitted back to back on the physical link. But once we use a larger N value, we find that gap errors tend to increase, as does the degree of variation. Thus, the gap errors shown in Fig 4.11 with $R = 10Gbps$ are partly contributed by these instrumental errors; the actual errors are smaller than the measured values.

4.7 Summary

In this chapter, we have identified three challenges for accurate gap creation on ultra-high-speed networks: timing resolution, interrupts, and transient queuing. We implement and evaluate them in our 10Gbps testbed, and find that the *Dummy-packet* approach best scales to ultra-high speed—it creates the most accurate inter-packet gaps with less than 1μs errors; yields the most accurate bandwidth estimation accuracy; takes up reasonable system overhead; and ensures gap accuracy when the system is overloaded.

Due to the outstanding performance of *Dummy-packet* among the three considered mechanisms, we adopt this approach in the implementation of TCP RAPID.

CHAPTER 5: DENOISING FOR BANDWIDTH ESTIMATION

Bandwidth estimation plays a crucial role in RAPID congestion control. The protocol continuously transmits p-streams in order to estimate avail-bw so that the sending rate of the next p-streams can be adjusted according to the avail-bw on the network path.

Robust avail-bw estimation relies on finely controlling inter-packet gaps and dealing with noise on the path. In Chapter 4, we addressed the first issue, the problem of creating precise inter-packet gaps, with our *Dummy-frame* mechanism. In this chapter, we focus on the second issue, the problem of denoising the path. Noise distorts the signature of persistent queuing delay in observed inter-packet gaps and results in inaccurate bandwidth estimation. Section 1.2.2 identifies two sources of noise that may impair bandwidth estimation: noise from bursty cross-traffic at the bottlenecks and transient queuing at non-bottleneck resources. It is commonly acknowledged that a longer probing timescale helps to reduce the impact of noise on bandwidth estimation; for example, [35] and [34] strongly recommend the use of long p-streams for robust bandwidth estimation, especially on high-speed paths. However, TCP RAPID congestion control prefers short p-streams for two reasons. First, shorter probing timescales help to reduce the network overhead involved in probing. In p-streams with exponentially increasing probing rates, the rates in the latter part of the p-stream may exceed the avail-bw, and short streams help to reduce the duration of each overloads of the network path. Second, a short p-stream timescale allows RAPID to track more closely and adapt more quickly to changes in the avail-bw on the paths.

In this chapter, we aim to develop a denoising algorithm that both achieves accurate bandwidth estimation and minimizes the length of the p-streams. We first consider and evaluate two state-of-the-art denoising techniques; we find that they fail to deliver robust estimation at 10 Gbps unless extremely long p-streams are used. Next, we develop a denoising technique called Buffering-Aware Spike Removal (BASS), which enables accurate bandwidth estimation with short p-streams. Finally, we employ several machine learning (ML) algorithms to discover the relationship between the noisy gaps and avail-bw. With a properly chosen algorithm, the ML-based approach outperforms BASS, even with even shorter p-streams.

5.1 State of the Art

One problem for bandwidth estimation, according to the developers of $ABest$ tools, is transient queuing at the receiver, which creates interrupt coalescence by attempting to group multiple packets received within a short time frame. Two promising tools, Pathload and Pathchirp, rely on detecting the increasing trends of one-way delays. They use these trends to evaluate whether a probing rate exceeds avail-bw or not—in other words, whether inter-arrival gaps at the receiver g_r are persistently larger than the intended inter-packet gaps at the sender g_s. Interrupt coalescence causes packets to buffer in batches at the NIC receiving queue before being handed to the operating system for timestamping; it thus produces the "spike-dips" pattern in the observed g_r as plotted in Fig. 1.3.

Pathload [20] attempts to identify packets that arrive within a single interrupt burst and to eliminate all coalesced packets from the burst except the last one, which experiences the minimum queuing delay at the receiver NIC. Unfortunately, [33] has shown Pathload's poor performance in the presence of non-negligible interrupt delays ($> 125\ \mu s$) on 100Mbps paths. Pathchirp [22] addresses the issue of interrupt coalescence by sending 6 packets at each probing rate (instead of one packet per rate) within a p-stream; however, this still yields a significant estimation error in the presence of interrupt delay [34].

This section focuses on testing how two promising denoising techniques from the literature, IMR-Pathload and PRC-MT, handle the noise induced by interrupt coalescence. These techniques, which focus on single-rate probing, have been shown to perform better on 100 Mbps paths than other denoising techniques.

5.1 IMR-Pathload

IMR-Pathload [33] is based on the Pathload algorithm. To deal with the effects of interrupt coalescence, it applies signal denoising techniques on the observed one-way delays. The signal denoising techniques it uses are window-based averaging, referred to as *IMR-avg*, and multi-level discrete wavelet transform, referred to as *IMR-wavelet*.

Window-based averaging is a technique that is often used to smooth out short-term fluctuations and highlight long-term trends [76]. To smooth out the buffering-related noise on each individual one-way delay, IMR-avg computes the average of every $\frac{N}{10}$ adjacent observations, then feeds the 10 averaged observations to the trend-detection algorithm, which determines whether one-way delays in that p-stream are consistently increasing.

Wavelet transforms are widely used to denoise real-world noisy data, especially when the noise corrupts the data in a significant manner [77]. A wavelet transform separates input data into two sets of coefficients: the wavelet coefficients and the scale coefficients. The wavelet coefficients represent noise in the input data and are discarded; the scale coefficients represent the input recovered from the noise. IMR-wavelet applies a three-stage wavelet denoising to the one-way delays. Each later stage takes as input the scale coefficients (the clean, denoised data) of the previous stage. In each state, the standard Daubechies wavelet transform is applied [78]. After the three-stage transform, Pathload uses the scale coefficients produced by the final stage for trend detection.

[33] evaluated both denoising techniques in their 100Mbps testbed, and reported that both window-based averaging and multi-level discrete wavelet transforms significantly improve the trend-detection accuracy of Pathload, reducing the estimation error to within 13%.

5.1 PRC-MT

Algorithm 4 PRC-MT: N-tuning Algorithm

1: ADR = get_adr()
2: N=N_{max}
3: $N_{prev} = N_{min}$
4: **while** true
5: r_{out}=send_and_recv_probe_stream(N,ADR)
6: ratio1=$\frac{r_{out}}{ADR}$
7: r_{out}=send_and_recv_probe_stream(N,ADR)
8: ratio2=$\frac{r_{out}}{ADR}$
9: **if** $|ratio1 - ratio2| < \sigma$
10: **goto** 1
11: **else**
12: **goto** 2
13: **1:** $N = \frac{N+N_{min}}{2}$
14: **2:** $N = \frac{N+N_{max}}{2}$
15: **if** $|N - N_{prev}| < \eta$
16: **break**
17: N_{prev}=N
18:
19: return N

PRC-MT [34] estimates available bandwidth by searching for the largest probing rate (r_{in}) so that the corresponding packet arrival rate (r_{out}) at the receiver is not lower than r_{in}. Its denoising technique is based on the idea that longer probing timescales help to reduce the impact of short-term noise. PRC-MT ensures that the p-streams are long enough that buffering-related noise has no significant impact on estimation accuracy.

The tool uses Algorithm 4 to tune the proper p-stream length N between $N_{min} = 60$ and $N_{max} = 3000$. It first computes the asymptotic dispersion rate (ADR) on the path, which is proven to be larger than the available bandwidth [79]. With the initial $N = N_{max}$, multiple p-streams of length N are transmitted at the

rate ADR, and their corresponding receiving rate r_{out} is monitored. If $\frac{r_{out}}{ADR}$ converges, it indicates that the value of N is large enough to reduce noise. When this happens, PRC-MT employs a smaller N as $\frac{N+N_{min}}{2}$; otherwise it tries a larger N as $\frac{N+N_{max}}{2}$. N is repeatedly tuned in this binary-search manner.

Once N is determined, PRC-MT transmits p-streams of length N at different probing rates R. Each p-stream's probing rate R is evaluated—if R does not exceed the receiving rate at the receiver, then R is smaller than avail-bw, and PRC-MT increases the probing rate of the next p-stream; otherwise, R is larger than avail-bw, and PRC-MT reduces the probing rate for the following streams.

[34] has shown that PRC-MT properly tunes p-stream length for various cross-traffic settings in their 100 Mbps testbed, and that it outperforms Pathload and Pathchirp when there is significant interrupt delay at the receiver. This research also indicates that a larger N is required for higher link speeds and larger interrupt delay values.

Goal

In this chapter, we examine techniques that help TCP RAPID produce robust bandwidth estimation with short p-streams on 10 Gbps paths. State-of-the-art denoising mechanisms, mentioned in Section 5.1, are first evaluated in our 10 Gbps testbed. If none of these techniques succeed in scaling to these ultra-high speeds, new denoising techniques will be developed to enable robust estimation using short p-streams.

5.2 How well does the state of the art work?

In single-rate probing, the decision about *whether or not the probing rate of a given p-stream is larger than the available bandwidth* is a crucial one; estimation tools use this decision to either increase or reduce the probing rate of the next p-stream. We evaluate how often the state-of-the-art techniques described in Section 5.1 make the correct decision in the presence of noise. To determine this, we run testbed experiments in the presence of the bursty cross-traffic in our 10 Gbps testbed (with avail-bw ranging from to 4.7 Gbps to 9.1 Gbps), sending p-streams of different lengths and 10 different probing rates ($5 - 9$ Gbps, stepped by 0.5 Gbps). For each probing rate, we collect 5000 p-stream samples observed at the receiver.

5.2 Using Long p-streams

We first run the N-selection logic of PRC-MT in our testbed with bursty cross-traffic. The result dictates that p-streams should be of at least 320 packets to alleviate the impact of noise on the path. Therefore, we

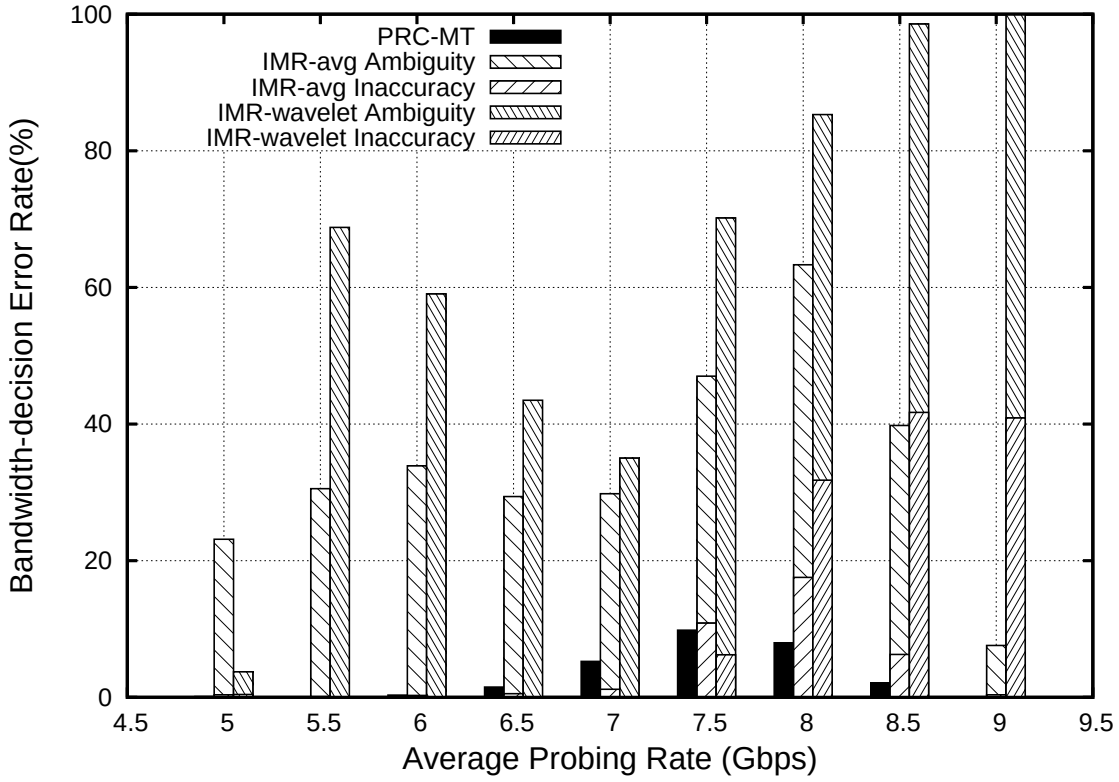


Figure 5.1: Bandwidth Decision Errors: State of the Art

first generate single-rate p-streams of $N = 320$ for each of the 10 probing rates, and examine whether such long p-streams yield robust estimates.

We then apply the denoising techniques described above to each probe-stream and evaluate each technique's accuracy in estimating whether the corresponding probing rate is higher than the available bandwidth. Recall that IMR-avg and IMR-wavelet, the denoised p-streams, are fed to the Pathload trend-detection algorithm; Pathload may mark the probing rate of a p-stream as "ambiguous" if it is not clear that there is an increasing trend in that stream's one-way delays. In contrast, PRC-MT evaluates a probing rate as being higher than avail-bw if the receiving rate is smaller than the probing rate.

Fig. 5.1 summarizes the inaccuracy results for pstreams of $N = 320$ with different average probing rates and different smoothing strategies. The *bandwidth decision error rate* plotted represents the fraction of p-streams that decided incorrectly whether a given probing rate was larger than the available bandwidth. For

IMR-avg and IMR-wavelet, we also include bars for the fraction of probe-streams that were unable to arrive at a decision (ambiguous). We find the following:

- PRC-MT is fairly accurate when each p-stream consists of 320 packets. More than 90% of p-streams accurately infer whether the probing rate is higher or lower than the available bandwidth, despite the presence of bursty cross-traffic.
- IMR techniques fail to arrive at a decision for a huge fraction of p-streams— the *ambiguity* bars in Fig. 5.1 exceed 50% for most probing rates. Both IMR-avg and IMR-wavelet yield much lower accuracy than PRC-MT for the p-streams for which they do provide an answer.

For a p-stream that probes at a higher rate than avail-bw, we collect one-way delays observed at the receiver and then apply IMR-avg or IMR-wavelet on them. Fig. 5.2 plots the smoothed one-way delays. Despite the denoising, they retain a saw-tooth pattern due to the impact of heavily coalesced arrivals, failing to reveal a robust increasing trend. The trend-detection algorithm that Pathload and IMR-Pathload rely on is not robust in ultra-high-speed networks, even with the use of denoising techniques. Therefore, in the rest of this section, we focus only on PRC-MT.

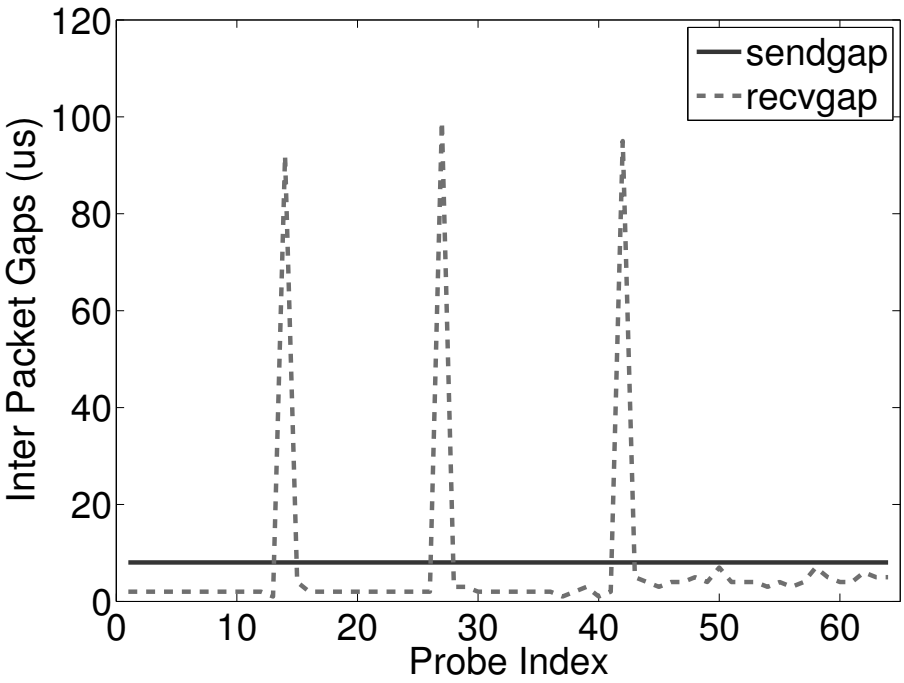
5.2 Reducing p-stream length

Probe streams at a length of 320 packets span a notably large timescale (at least 2.1 ms on 10 Gbps paths with $MTU = 9000\ B$), especially for high-speed congestion control, which is the domain we consider here. The following section evaluates the performance of PRC-MT on p-streams of different lengths N, ranging from 320 to 32, and investigates how estimation accuracy decreases with shorter probing timescales.

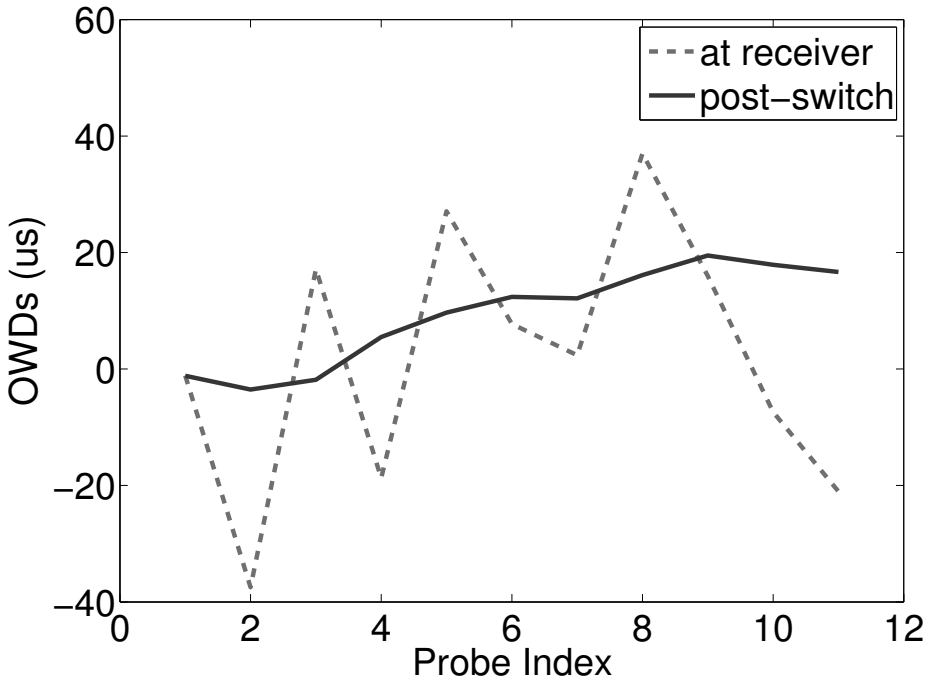
In Fig. 5.3 we plot the inaccuracy ratio for all three denoising techniques with four different N values ($N = 32, 64, 128, 320$). We find that the performance of all three denoising techniques degrades with smaller Ns. IMR-avg and IMR-wavelet both fail to produce accurate estimations for any N value, and in all cases, these techniques yield a high ambiguity ratio.

Fig. 5.3(c) plots the inaccuracy ratio of PRC-MT with $N = 32, 64, 128, 320$, respectively. Results show that PRC-MT performs terribly with smaller N; when $N = 128$, it can give wrong judgments about probing rates for more than 70% of p-streams—even worse than IMR-avg and IMR-wavelet.

In conclusion, all three state-of-the-art denoising mechanisms fail to estimate robustly at short probing timescales. IMR-avg and IMR-wavelet are unable to reveal the trend in one-way delays from denoising on

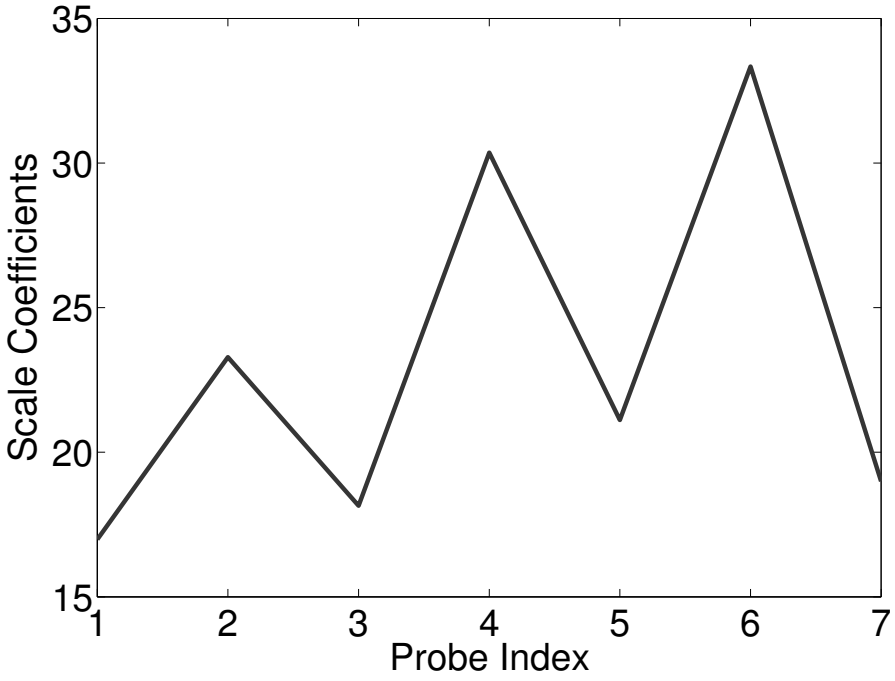


(a) Probing at 9 Gbps, AB: 8.4 Gbps



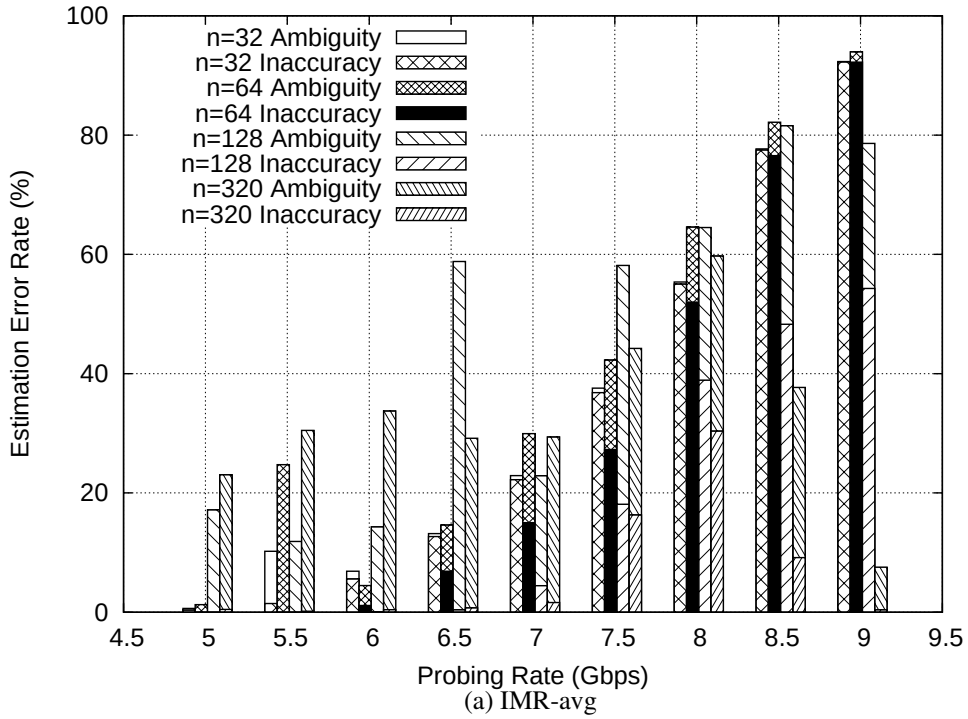
(b) IMR-avg

Figure 5.2: A P-stream Smoothed by IMR-Pathload

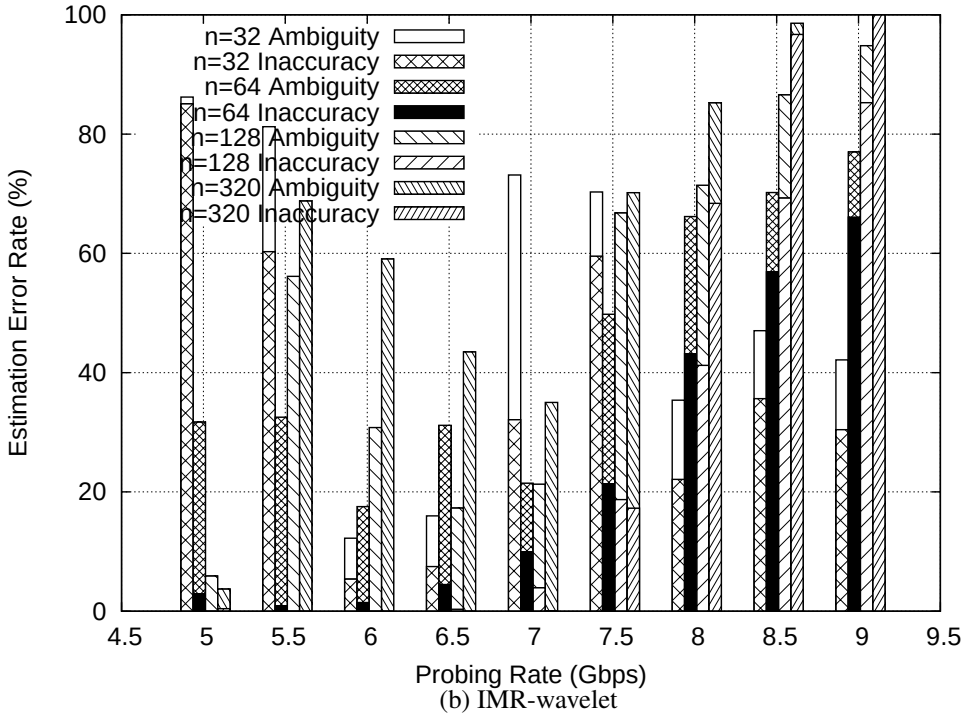


(c) IMR-wavelet

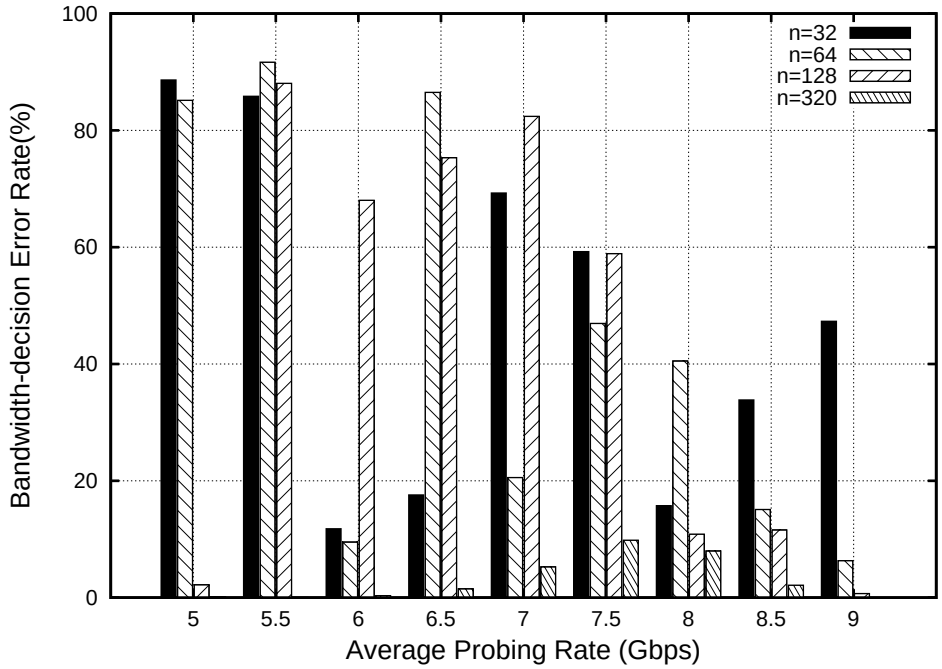
Figure 5.2: A P-stream Smoothed by IMR-Pathload (cont.)



(a) IMR-avg



(b) IMR-wavelet



(c) PRC-MT

Figure 5.3: State-of-the-Art: Impact of N

high-speed paths, and PRC-MT yields robust estimation only when extremely long p-streams are used. In the following sections of this chapter, we work diligently on techniques that meet both requirements .

5.3 BASS

To ensure that short p-streams accurately estimate avail-bw, we develop a novel denoising technique, Buffering-Aware Spike Smoothing (BASS), to overcome the noise in bandwidth estimation at ultra-high speeds. In this section, we first describe the denoising algorithm, then apply it to both single-rate and multi-rate probing frameworks, showing that it meets our goal: it makes feasible p-streams as short as $N = 64$ while achieving robust estimation on 10 Gbps networks.

5.3 BASS algorithm

Buffering Event

We define a "buffering event" as any event that causes packets to get queued up. For example, when a system resource is temporarily unavailable because of competing traffic or processes, packets wait for access to necessary resources in a system queue. These buffering events can also happen at bottleneck switches or routers (when packet arrival rate exceeds the device transmission rate, a queue builds up at the input port) and at the receiving NIC when interrupt coalescence is used (packets buffer at the NIC incoming queue, waiting for the NIC to generate an interrupt to the operating system). Consider what the "spikes" and "dips" in Fig. 1.3 represent. Buffering events caused by interrupt coalescence wait until an interrupt is generated, and the buffered packets are then timestamped by the operating system in a batch. There is a long queuing delay between the first packet in a queue and the final packet in the previous batch; in other words, there is a large inter-packet gap (the "spike") between the two. The remaining packets in the queue get processed fairly quickly, with only fairly small gaps ("dips") between them. It is clear that this kind of bursty buffering event (a "spike" and all the "dips" immediately following it) destroys the inter-packet gap patterns that were present before the spike.

Buffering-aware Spike Smoothing

The previous section described existing smoothing techniques IMR-avg and PRC-MT, which attempt to recover patterns in noise-impacted packet delays by averaging across a large fixed window. This section considers how the size and the boundaries of the averaging window impact the effectiveness of denoising.

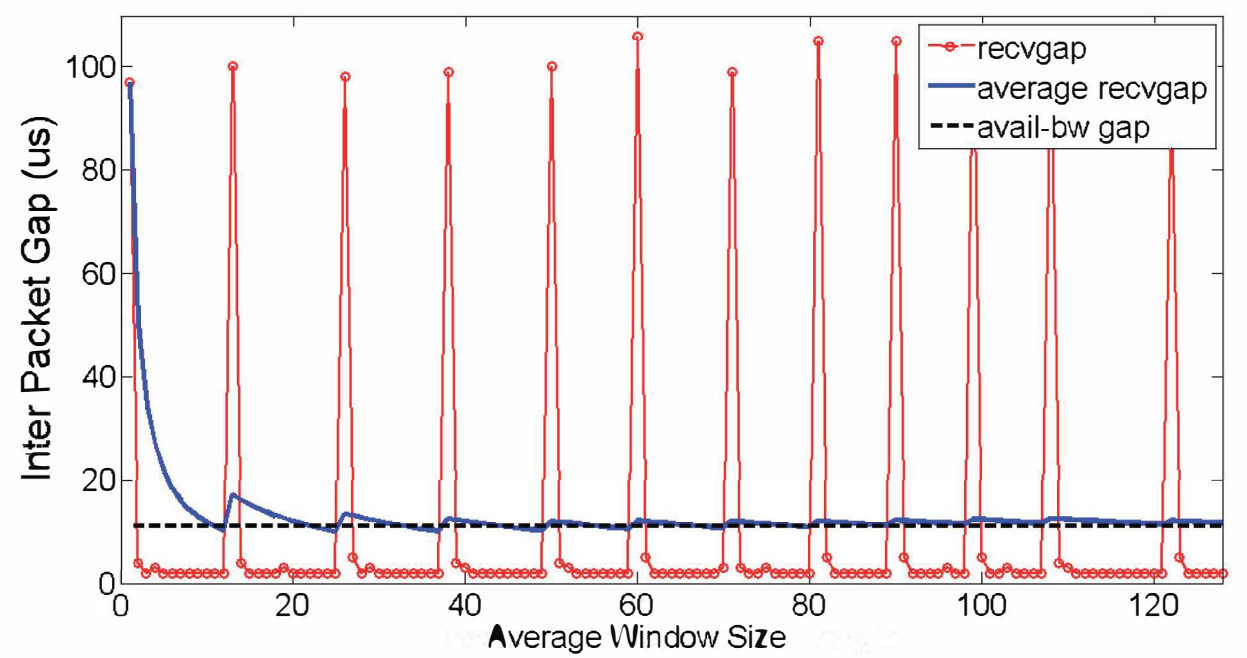


Figure 5.4: Averaging with Various Window Sizes

For a single-rate p-stream, collected in the previous evaluation (Section 5.2.1), we apply smoothing on the received stream with different averaging window sizes. Fig. 5.4 plots the inter-packet gaps observed at the receiver with the red line, and the expected receiver-side gaps in a noise-free path with the black dashed line. The expected gap is computed by analyzing the actual avail-bw on the path using the DAG trace. Ideally, the red line should overlap with the black line. The blue line depicts the denoised gaps—for each point (x, y) on the line, y is the mean gap with averaging window size x, covering packets from index 1 to x. We observe the following:

- A larger averaging window helps to recover gaps from noise, as the blue line approaches the black dashed line for larger values of x. This supports the argument, made by previous researchers [35, 34], that a larger averaging/probing timescale reduces the impact of noise.

- Even with small average windows, there exist specific averaging window sizes at which the denoised gap matches the ideal gap (e.g., $x = 12, 26$). Note that these averaging windows overlap completely with the beginning and end of buffering events. This suggests that if inter-packet gaps are smoothed strictly and completely within buffering event boundaries, the underlying signature of one-way delays may be recovered even within a very small averaging timescale.

Algorithm 1 Buffering Aware Spike Smoothing

```
1: function AverageProbeStream(spike_begin, spike_end)
2:     sum_sendgap ← 0, sum_recvgap ← 0
3:     if spike_begin < spike_end then
4:         for i = spike_begin → spike_end do
5:             sum_sendgap+ = send_gap[i]
6:             sum_recvgap+ = recv_gap[i]
7:         for i = spike_begin → spike_end do
8:             send_gap[i] = sum_sendgap ÷ (spike_end-spike_begin+1)
9:             recv_gap[i] = sum_recvgap ÷ (spike_end-spike_begin+1)
10: function SpikeRemoval
11:     i ← 0, spike_state ← NONE
12:     if recv_gap[0] > recv_gap[1] + SPIKE_DOWN then
13:         spike_begin ← 0
14:         spike_max ← recv_gap[0]
15:         spike_state ← SPIKE_VALID
16:         i ← 1
17:     for i → recv_gap.size()-1 do
18:         switch spike_state do
19:             case NONE
20:                 if recv_gap[i] + SPIKE_UP < recv_gap[i+1] then
21:                     spike_end ← i
22:                     AverageProbeStream(spike_begin,spike_end)
23:                     spike_state ← SPIKE_PENDING
24:                     spike_begin ← i+1
25:                     spike_max ← recv_gap[spike_begin]
26:                 break
27:             case SPIKE_PENDING
28:                 spike_max = max{spike_max, recv_gap[i]}
29:                 if recv_gap[i] + SPIKE_DOWN < spike_max then
30:                     spike_state ← SPIKE_VALID
31:                 else
32:                     break
33:             case SPIKE_VALID
34:                 if recv_gap[i] + SPIKE_UP < recv_gap[i+1] then
35:                     spike_end ← i
36:                     spike_state ← SPIKE_PENDING
37:                     spike_max ← recv_gap[i+1]
38:                 else
39:                     if recv_gap[i]=recv_gap.back() then
40:                         spike_end ← i
41:                     break
42:                 AverageProbeStream(spike_begin, spike_end)
43:                 spike_begin ← i+1
44:     AverageProbeStream(spike_begin, spike_end)
```

Figure 5.5: BASS: Pseudo-code

The key observation is that within each buffering event, the average sending rate and receiving rate can be maintained. This also suggests that if a smoothing strategy smooths within the boundaries of individual buffering events, rather than at a fixed timescale that is agnostic of buffering-event boundaries, it may better recover the underlying signatures in one-way delays. Only buffering events in which all "spikes" and "dips" occur completely within a probe-stream should be considered for averaging to recover signatures in the receiver-side gaps.

Based on the findings above, the following smoothing strategy is considered to reduce the impact of noise on a probe-stream:

1. Explicitly identify the boundaries of each buffering event (starts with a spike and ends with the last dip before the next spike);

2. Eliminate data related to incomplete spikes at either end of the probe-stream; and

3. Average out the gaps/one-way delays within each buffering event.

We refer to this smoothing strategy as *Buffering-aware Spike Smoothing (BASS)*.

Fig. 5.5 provides the pseudo-code for BASS. The parameter $SPIKE\_UP$ helps us to determine the boundaries of buffering events. A buffering event starts at index i if the g_r^i is determined to be larger than g_r^{i-1} by $SPIKE\_UP$, and it ends before the start of the next buffering event. In a buffering event, "dips" follow the "spike"— packets are queued up in a batch after the first packet in the queue, and the gaps between them are smaller than the one between the first packet of the batch and the last packet of the previous batch. Without the dips, it is not considered a valid buffering event, even if the first packet observes a gap larger by $SPIKE\_UP$ than its preceding packet. To make sure a buffering event is valid, we introduce another parameter, "$SPIKE\_DOWN$," to examine whether "dips" are observed. Once a buffering event starts, the algorithm keeps track of the difference between the "spike" and the smallest "dip"—if they differ by $SPIKE\_DOWN$, then this buffering event is marked as "$SPIKE\_VALID$," and averaging is conducted for observations in this buffering event. The more packets queued up in a single buffering event, the higher the "spike" expected. Here, we mainly deal with receiver-NIC buffering due to interrupt coalescence; in this type of buffering, with 10 Gbps NICs, packets can be queued up by the hundreds of μs, creating very large spikes. $SPIKE\_UP = 20\mu s$ and $SPIKE\_DOWN = 10\ \mu s$ are adopted to capture these buffering events.

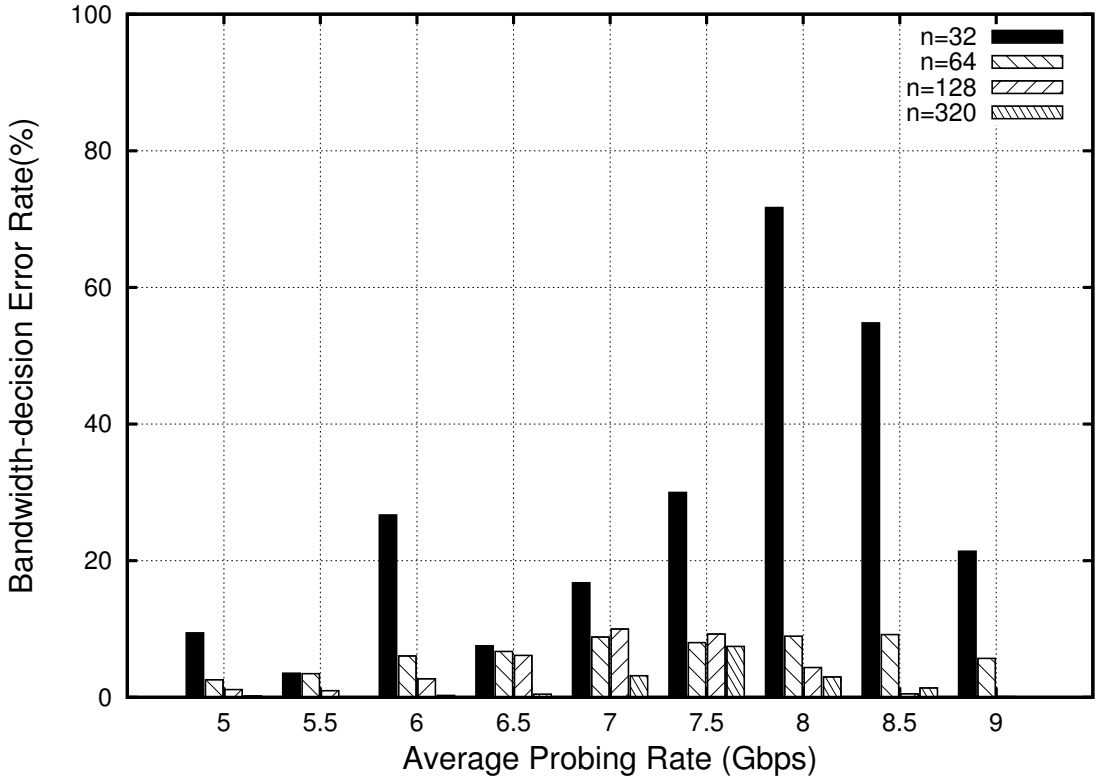


Figure 5.6: Bandwidth Estimation Error: PRC-MT + BASS

5.3 Does BASS Outperform PRC-MT?

We evaluate whether the signature recovered after BASS effectively improves the estimation accuracy of single-rate probing and compare the performance of BASS with that of the best-performing existing noise-reduction strategy, PRC-MT. We use the same sets of p-streams collected for Fig. 5.3(c) and apply BASS to denoise. Then, with the smoothed p-stream (g_s, g_r), we use the BASS algorithm to evaluate whether each probing rate exceeds the avail-bw or not: if the average sending rate r_{in} is no larger than the receiving rate r_{out}, then $R \leq avail - bw$; otherwise, $R > avail - bw$.[1]

Fig. 5.6 plots the estimation inaccuracy with different p-stream lengths N. When we compare this data with that in Fig. 5.3(c), we find that BASS significantly improves estimation accuracy at smaller N—-even with p-streams of just 64 packets, the inaccuracy is limited to within 10%. However, p-streams with just 32 packets continue to yield high errors.

[1]We find that the trend-detection algorithm used by Pathload and IMR-Pathload fails to scale to 10 Gbps, so we do not consider it.

5.3 BASS for Multi-rate Probing

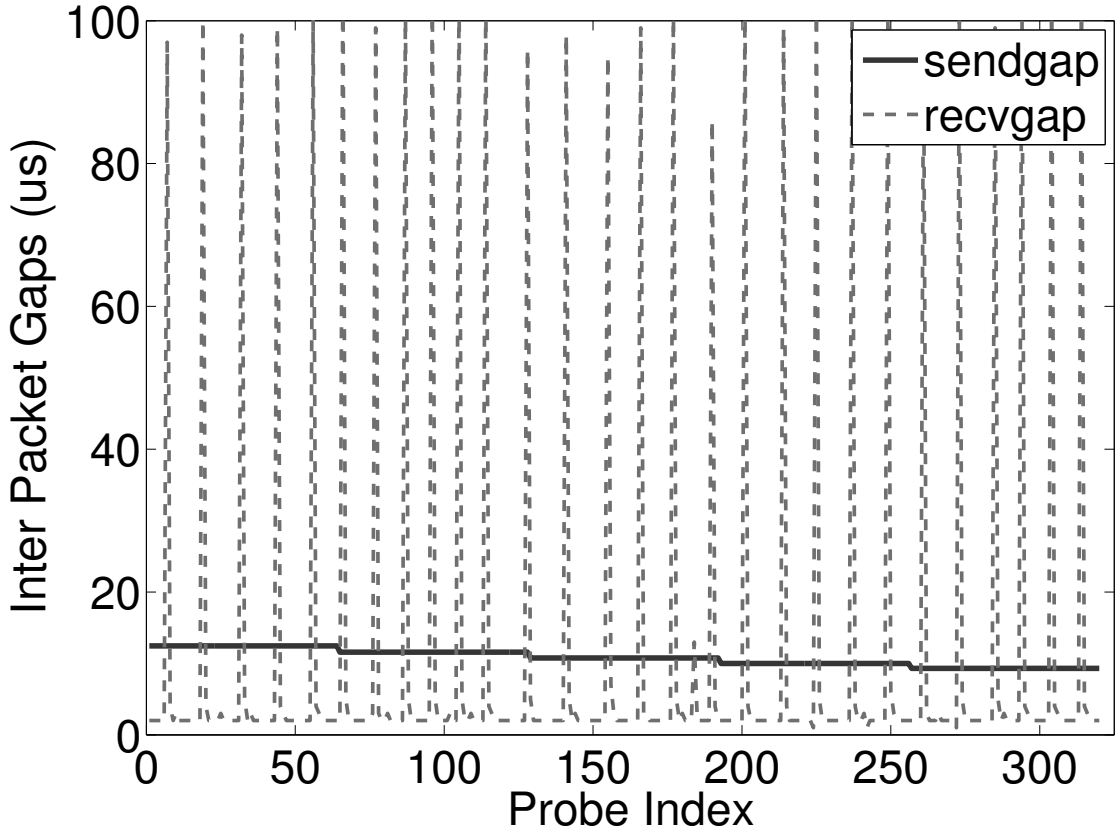
The previous section has shown that BASS successfully enables short p-streams of $N = 64$ while ensuring high estimation accuracy (with less than 10% error) for the single-rate probing framework. Recall that TCP RAPID relies on a multi-rate probing framework—with the average sending rate specified by the rate-adapter, the p-stream generator transmits multi-rate p-streams and probes for multiple rates within each. To estimate bandwidth, it finds the highest probing rate at which no self-congestion is experienced; it relies compares g_r to g_s and finds the largest probing rate beyond which g_r consistently exceeds g_s. [44] shows that the multi-rate probing framework sends fewer p-streams to accomplish the estimation task than single-rate probing does, reducing the probing time and incurring less bandwidth overhead from the probing traffic. In this section, we evaluate our smoothing technique within a multi-rate probing framework.

5.3 BASS on Multi-rate p-streams

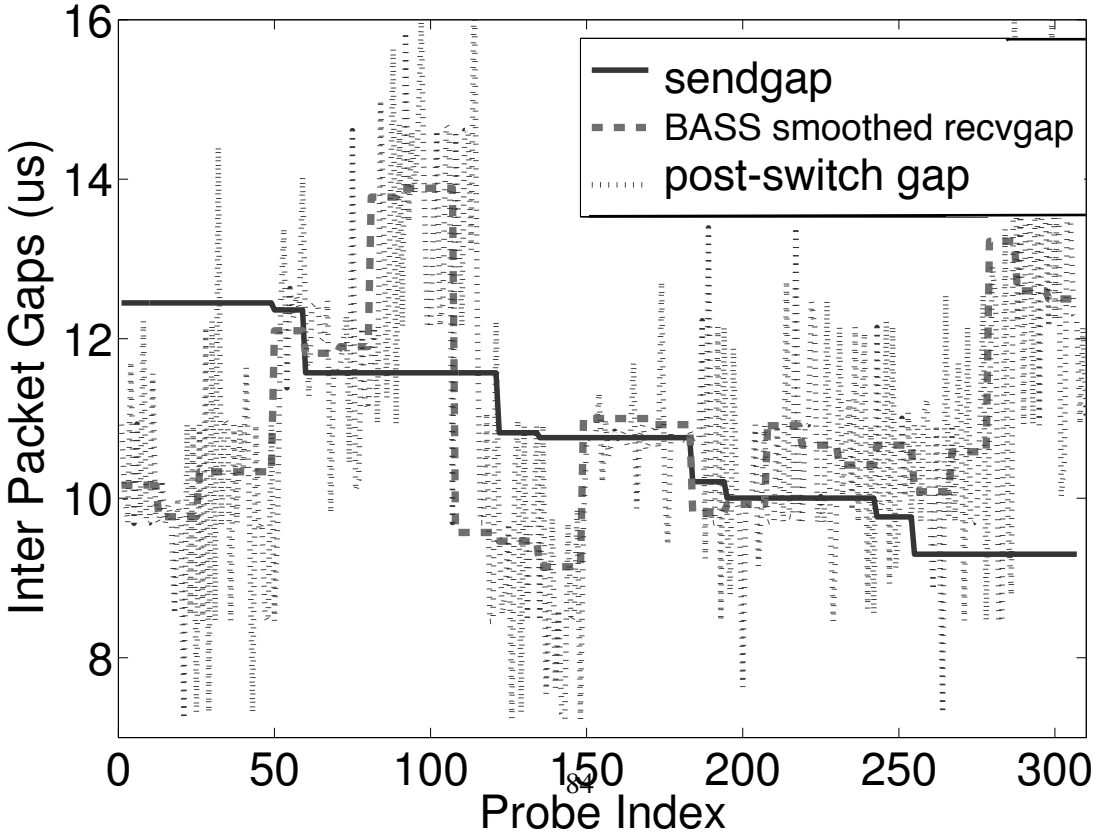
We pick a p-stream of $Range = 30\%$, $N_r = 5$, and $N_p = 64$ to illustrate the effects of *BASS* on typical multi-rate streams. Fig. 5.7(a) plots an example multi-rate probe-stream with typical distortions in g_r caused by interrupt coalescence—with the "spike-dips" pattern, every p-stream estimates avail-bw as the highest probing rate. Fig. 5.7(b) plots the smoothed g_r against the inter-packet gaps observed after the bottleneck switch, before the packets arrive at the receiver. Results suggest that the BASS-denoised g_r matches closely with the trend of noisy gaps observed after the bottleneck. This indicates that our smoothing technique effectively mitigates the impact of interrupt coalescence and reveals the time-varying throughput signature of the cross-traffic.

Multi-pass BASS

The smoothed stream after BASS in Fig. 5.7(b) still fails to reveal a robust signature of persistent queuing delays because the cross-traffic itself is bursty. The remaining (shorter) spikes and dips will lead to considerable over-estimation. Will further buffering-aware smoothing of this stream help?

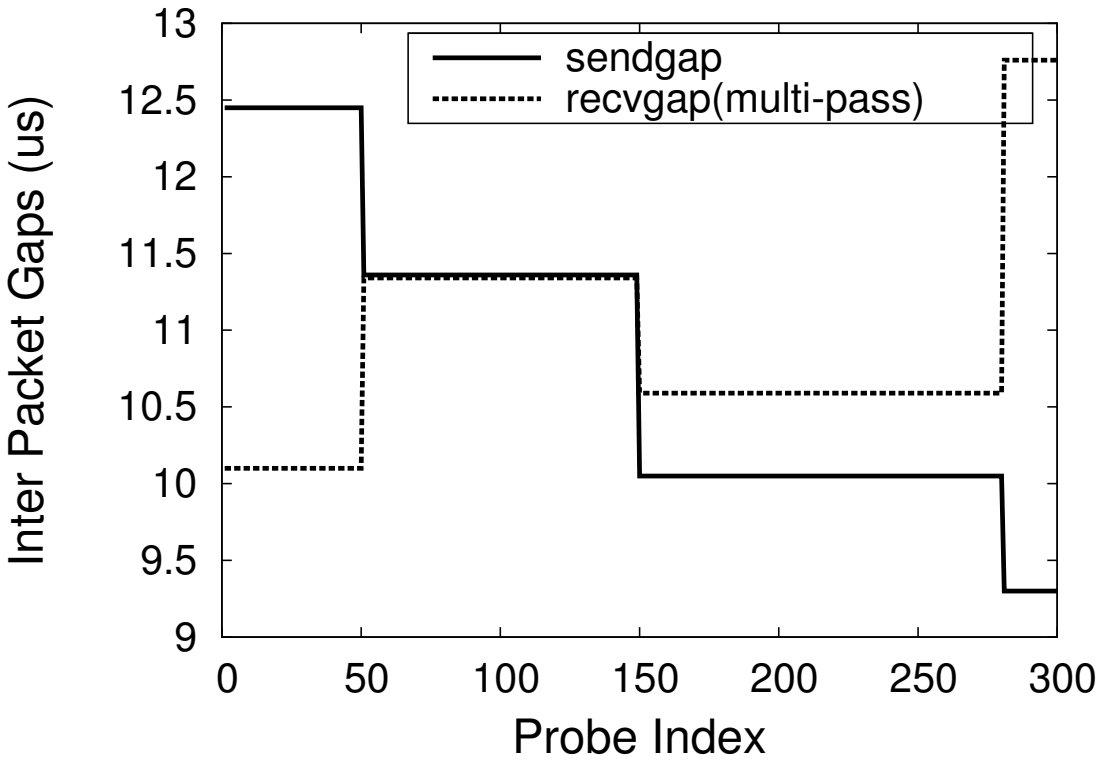


(a) Streams Observed at Receiver

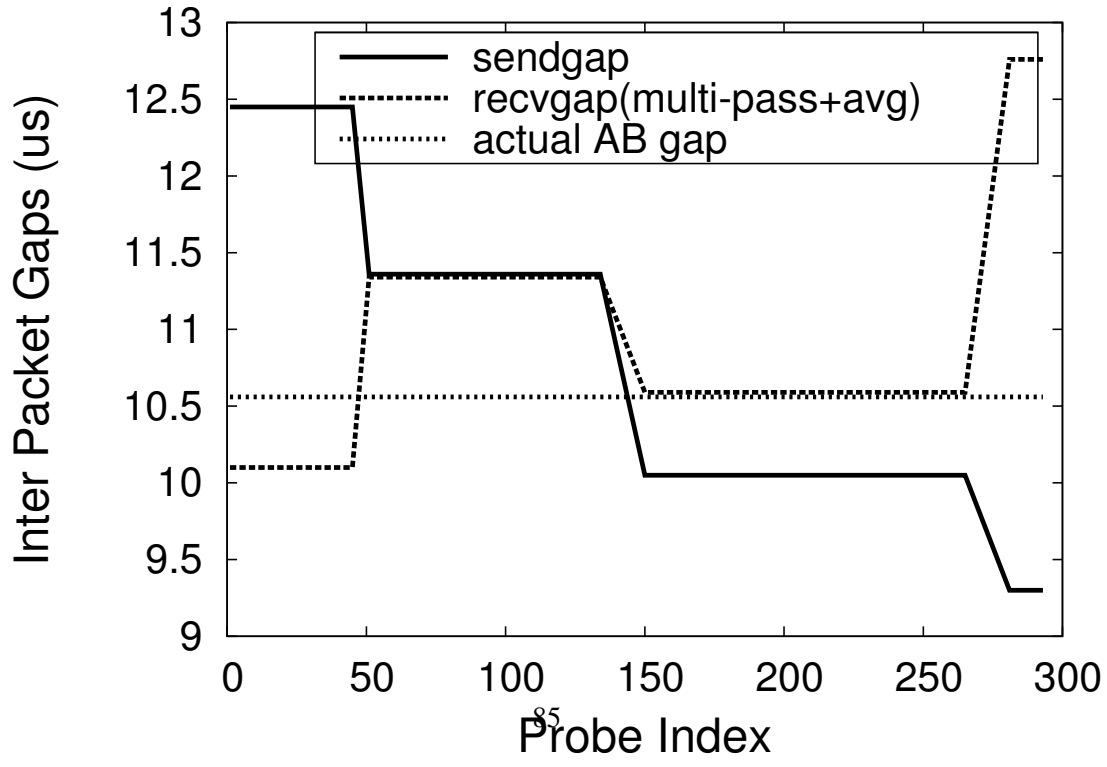


(b) BASS

Figure 5.7: BASS for a Multi-rate Probe Stream (avail-bw =6.8 Gbps)



(c) Multi-pass BASS



Algorithm 5 Multipass-BASS: Pseudo-code

1: $SPIKE\_UP = 20$

2: $SPIKE\_DOWN = 10$

3: SPIKEREMOVAL

4: $SPIKE\_UP = 5$

5: $SPIKE\_DOWN = 4$

6: SPIKEREMOVAL

7: $SPIKE\_UP = 2$

8: $SPIKE\_DOWN = 1$

9: **while** spike exists

10: SPIKEREMOVAL

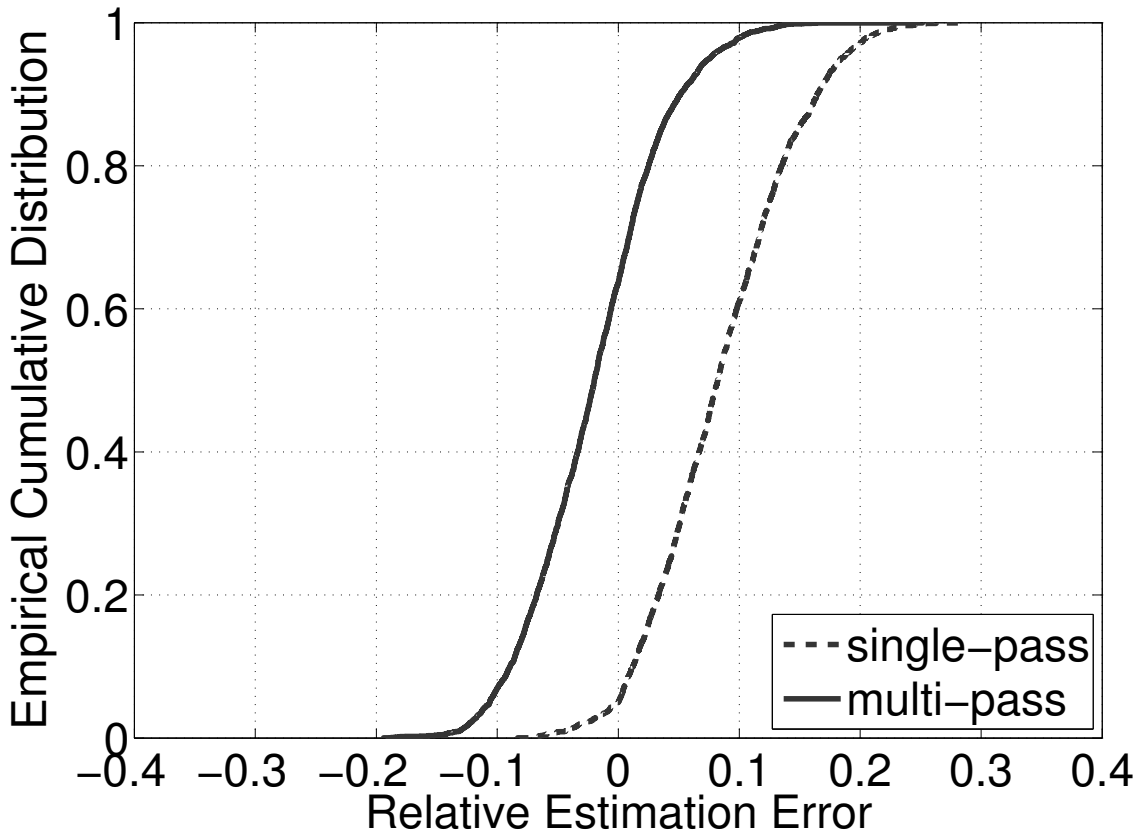


Figure 5.8: Multi-pass Spike Removal Reduces Over-estimation

Based on the observation in Fig. 5.7(b), we develop a *multi-pass spike smoothing* routine that continues to smooth the p-stream until a robust signature of queuing delays is observed; we refer to this routine as "multi-pass BASS." Fig. 5 illustrates the process of multi-pass BASS. It calls the BASS function $SPIKEREMOVAL$ multiple times, each time with different $SPIKE\_UP$ and $SPIKE\_DOWN$ parameters. In the first pass, the parameters $SPIKE\_UP = 20\ \mu s$, $SPIKE\_DOWN = 10\ \mu s$ (the same parameters as those used by BASS) are used to capture buffering events caused by interrupt coalescence. The next pass aims to identify buffering events caused by bursty cross-traffic, where packets experience less severe buffering than that caused by interrupt coalescence. Accordingly, the parameters are reduced to $SPIKE\_UP = 5\ \mu s, SPIKE\_DOWN = 4\ \mu s$. After this pass, all spikes higher than 5 μs (if there are any) are smoothed. Cross-traffic with different burstiness levels leads to different spikes-dips signatures. To capture buffering events caused by smoother traffic, we apply BASS with even smaller parameters: $SPIKE\_UP = 2\ \mu s, SPIKE\_DOWN = 1\ \mu s$, until no spikes higher than 2 μ exist. Finally, a sliding window that averages across the smoothed observations is applied, with a window of size AVG_WINDOW, to further smooth the denoised gaps.

Fig. 5.7(c) and Fig. 5.7(d) demonstrate the impact of multi-pass BASS on the same p-stream depicted in Fig. 5.7(b). The AVG_WINDOW used in Fig. 5.7(d) is 16. The smoothed receive gaps become consistently larger than send gaps after index 145, yielding a clear (and correct) available bandwidth estimate of 6.8 Gbps.

Necessity of Multi-pass BASS

To show the effectiveness of multi-pass BASS, we generate multi-rate p-streams of $N_r = 5$, $N_p = 64$, and $range = 30\%$. Thousands of p-streams are transmitted with controlled average sending rates in fixed increments of 20 Mbps between 1 Gbps and 9 Gbp. These p-streams share the 10 Gbps bottleneck link with the highly bursty cross-traffic. BASS is applied to these p-streams before performing bandwidth estimation. The actual available bandwidth on the path while each p-stream is in flight is computed from the DAG trace taken after the bottleneck. The estimated avail-bw is denoted as B_e and the actual avail-bw as B_g. For those p-streams whose probing range covers B_g, we compute *relative estimation error* as $\frac{B_e - B_g}{B_g}$.

Fig. 5.8 depicts the cumulative distribution of the metric and compares the results when using multi-pass BASS to the results from applying BASS only once. It shows that multi-pass BASS effectively eliminates the over-estimation that results from using only single-pass BASS.

5.4 Denoising for Bandwidth Estimation in TCP RAPID

In Fig. 5.8, we showed that BASS eliminates over-estimation, achieving less than 10% relative error for over 85% of p-streams. However, Fig. 5.8 evaluates only p-streams of probing $range$ $30\%, N_r = 5$, and $N_p = 64$. Several configuration parameters are likely to impact the performance of BASS, including: {p-stream length N, probing $range$, probing granularity, $GAP\_NS\_EPSILON$, and averaging window size AVG_WINDOW}.

- Probing $range$: For TCP RAPID, p-streams with a wider range of probing rates help to locate avail-bw more quickly. However, this wide range also means that TCP RAPID can probe for rates that deviate further than the actual avail-bw , and it thus may potentially yield higher estimation errors. We vary the probing $range$ and study how it affects the performance of BASS.

- *Probing granularity N_r:* With a fixed probing range, a finer granularity helps the estimation logic effectively find a probing rate closer to the actual avail-bw. Therefore, we vary the probing range and the number of probing rates N_r to study how they impact the accuracy of BASS.

- *Probe stream length N:* In Fig. 5.8, p-streams of 320 packets may last for several milliseconds on 10 Gbps path. This is a large timescale in the context of RAPID congestion control. However, shorter p-streams are considered more vulnerable to the impact of noise [35]. In this section, we reduce the p-stream timescale by using a smaller N_p and investigate whether multi-pass BASS can maintain its estimation accuracy.

- $GAP\_NS\_EPSILON$: Notice that in Section 2.3, RAPID bandwidth estimation logic relies on another parameter, $GAP\_NS\_EPSILON$, as a threshold for determining whether self-induced congestion occurs in that p-stream. If this parameter is set too small, the bandwidth estimator may categorize insignificant noise in gaps as a strong indicator of congestion. If it is set too large, the estimator may neglect actual congestion, causing over-estimation. In this section, we vary the parameter from 0 to 1 ms in order to determine an appropriate value for our RAPID implementation.

- AVG_WINDOW: In the multi-pass BASS denoising algorithm, we apply sliding-window smoothing after averaging out gaps within each of the buffering events. A very small AVG_WINDOW fails to

smooth the small-scale noise after BASS denoising; we therefore examine how AVG_WINDOW size affects BASS performance.

5.4 Impact of Probing Range

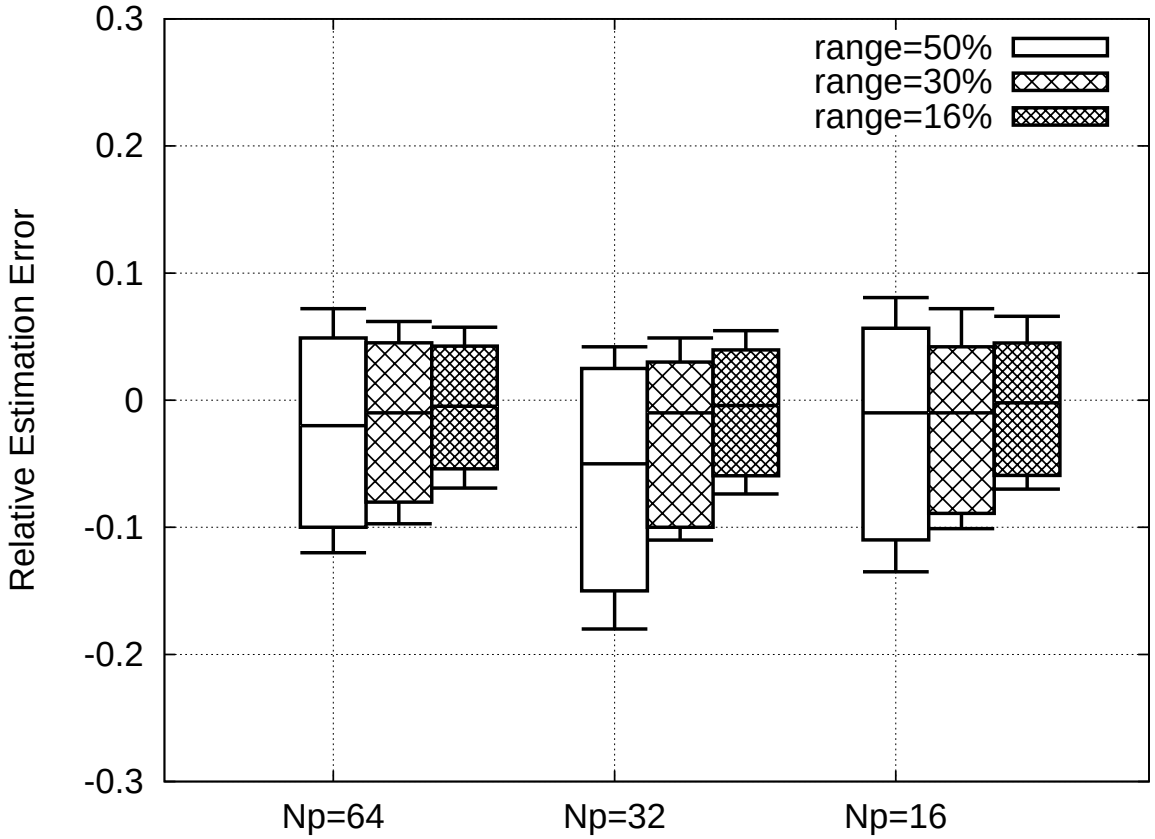


Figure 5.9: Impact of Probing Range

We use the same settings—$N_r = 5, GAP\_NS\_EPSILON = 100\ ns, AVG\_WINDOW = N_p$—as in Fig. 5.8, but we vary N_p from {64,32,16} and $range$ from {16%, 30%, 50%}. P-streams are generated using the *Dummy-frame* mechanism, sharing the 10 Gbps bottleneck link with BCT. Fig. 5.9 depicts the median and two distributions of relative estimation errors, $10 - 90\%$ and $5 - 95\%$. The height of the bar increases with $range$ in Fig. 5.9, indicating a higher degree of estimation error. We find that this observation holds for all different combinations of $N_r, N_p, GAP\_NS\_EPSILON$, and AVG_WINDOW. However, even with probing $range$ 50% and $N_p = 16$, BASS limits estimation error within 15% for over 85% of p-streams. Therefore, our implementation of RAPID can safely adopt 50% as the probing $range$ for each p-stream.

5.4 Impact of p-stream Length N

Next, we keep a fixed N_r and reduce N_p to examine whether shorter p-streams lead to more significant estimation errors. Fig. 5.9 shows that when the probing range is 30% and 16%, N_p offers comparable accuracy within the range from 64 to 16. For the 50% probing range, shorter p-streams do reduce accuracy—in Fig. 5.9, the height of the error bar increases with reducing length. However, the estimation is still accurate; relative estimation error is within 10% for over 85% of p-streams.

For each of the three probing ranges, we fix N_r at 4,[2] but vary N_p from 8 to 64 to examine to what extent we can reduce N_p while still achieving robust bandwidth estimation. Fig. 5.10 plots the cumulative distribution of relative estimation errors with different N_p values. We fail to observe significant decline in estimation accuracy with shorter p-streams as long as $N_p > 8$. As shown in Fig. 1.3, a p-stream of only 32 packets may only include packets from one complete buffering event—the multi-rate p-stream will become a single-rate p-stream after BASS. In order to cover at least two complete buffering events, we make sure that $N_r \times N_p$ is at least 64 and adopt $N = 64$ for RAPID implementation.

5.4 Impact of number of rates N_r

Finally, we study how the granularity of probing rates within each p-stream affects estimation accuracy. The following parameters are fixed in this set of experiments: $range = 50\%, N_r \times N_p = \{320, 96\}, GAP\_NS\_EPSILON = 100ns$, and $AVG\_WINDOW = N_p$, while N_r is varied from {2,3,4,6,8}.[3] In Fig. 5.11, we plot the distribution of relative estimation error with different N_r. With 320 packets in each p-stream, a larger N_r does yield less estimation error. With 96 packets in each p-stream, however, estimation accuracy is comparable once $N_r \geq 3$. This is because short p-streams of 96 length cover three to four complete buffering events; thus, after BASS, the denoised p-streams contain three to four probing rates, even if they actually probe at far higher rates. For RAPID, we set $N_r = 4$.

To sum up: we choose the following heuristics in our RAPID implementation: $range = 50\%, N_r = 4, N_p = 16, AVG\_SMOOTH = 16$, and $GAP\_NS\_EPSILON = 100ns$.

[2]We make N_r an even number for the convenience of Section 5.4.3, where we fix $N_r \times N_p$ while varying N_r; an even N_r helps to give more denominators.

[3]Since 320 is not dividable by 3 or 6, we use $N_p = 106, 53$ respectively in order to achieve a p-stream length as close to 320 as possible.

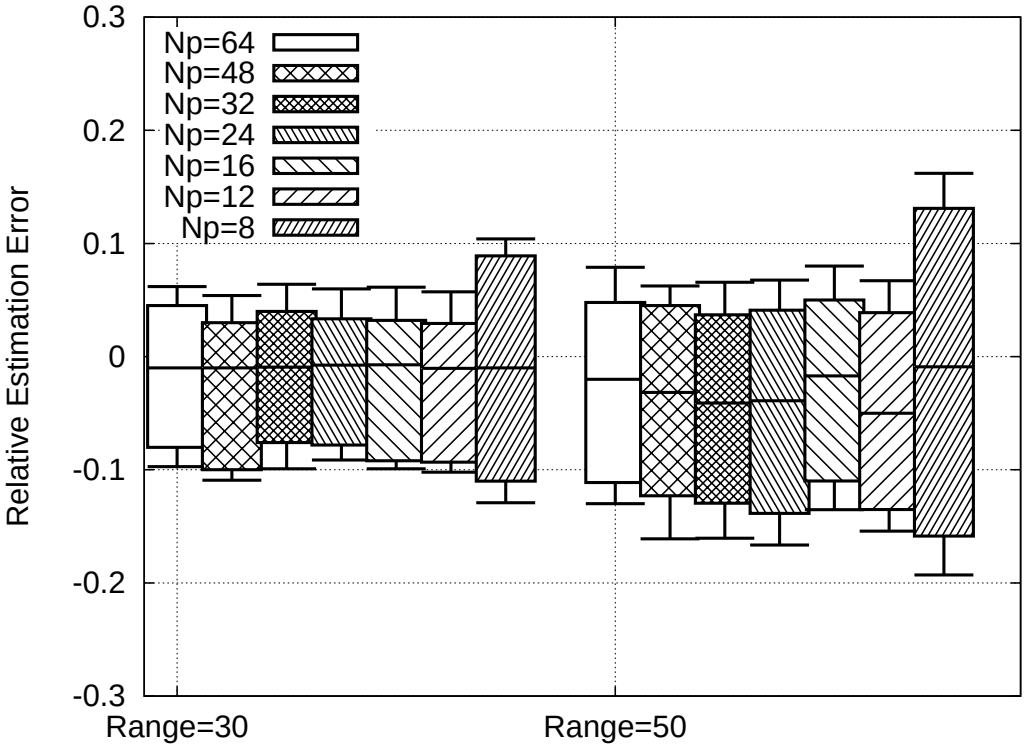


Figure 5.10: Impact of P-stream Length

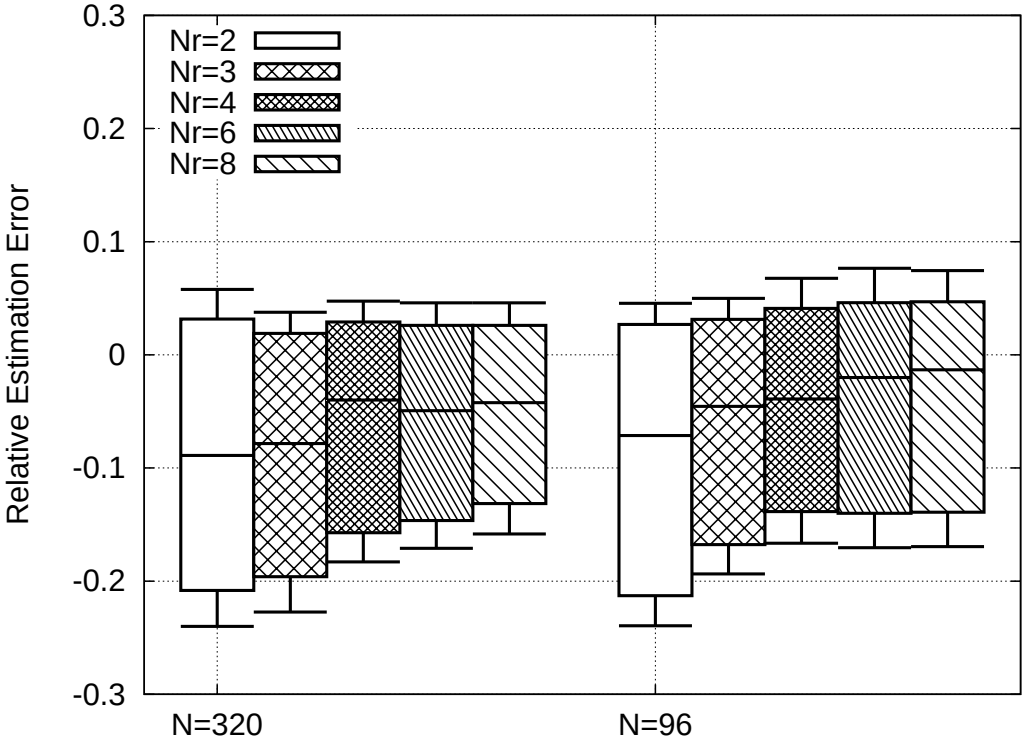


Figure 5.11: Impact of N_r

5.4 Impact of $GAP\_NS\_EPSILON$

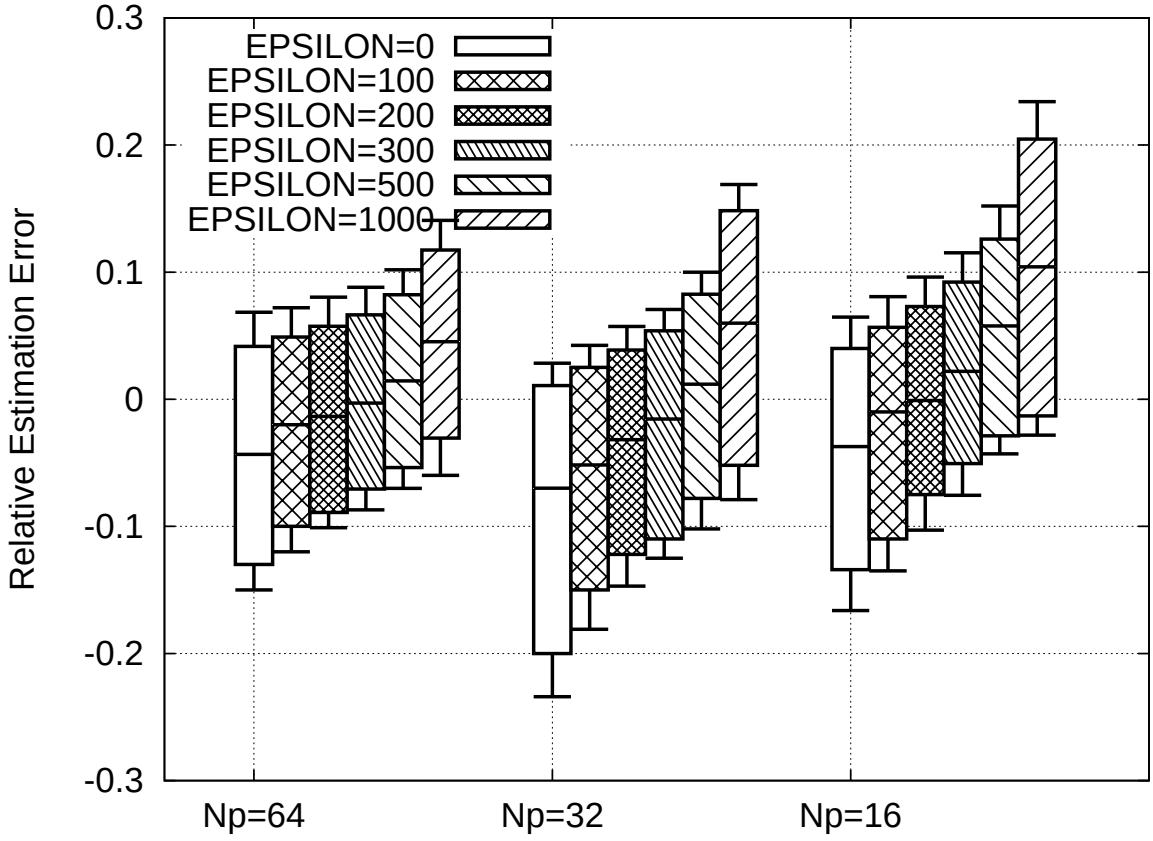


Figure 5.12: Impact of $GAP\_NS\_EPSILON$

We fix $N_r = 5, range = 50\%$, and $AVG\_WINDOW = N_p$ and highlight the impact of different $GAP\_NS\_EPSILON$ values from {0, 100 ns, 200 ns, 300 ns, 500 ns,1 ms}. We experiment with three different N_p {64,32,16}. Fig. 5.12 shows the distribution of relative estimation error; $GAP\_NS\_EPSILON$ does not change the range of errors, but a larger parameter consistently shifts the bars more to the over-estimation side.

In TCP RAPID, the estimated avail-bw is used to calculate the average sending rate for future p-streams, and an estimate higher than avail-bw causes the following p-streams to overload the path. To avoid persistent overloading, we prefer a parameter that tends to under-estimate, rather than over-estimate, avail-bw. Thus, we use $GAP\_NS\_EPSILON = 100\ ns$, which produces a median of estimation error below 0.

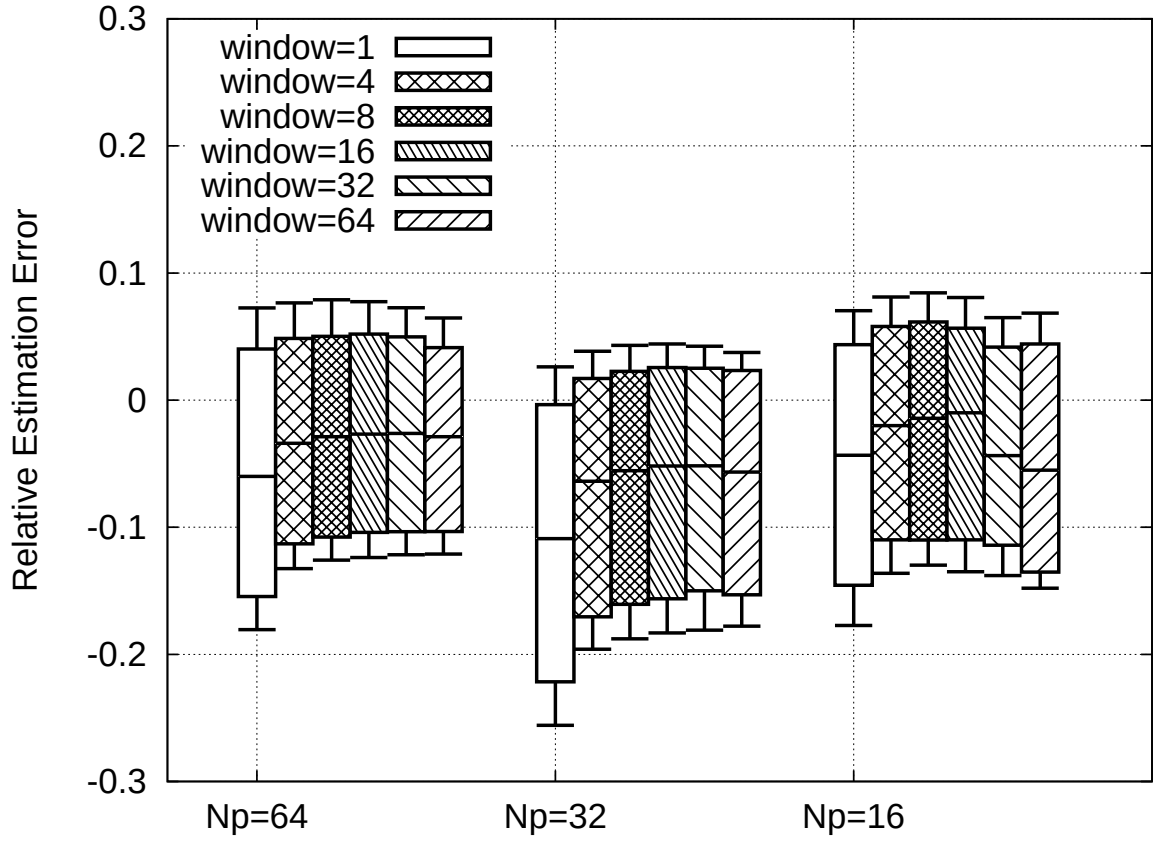


Figure 5.13: Impact of AVG_SMOOTH

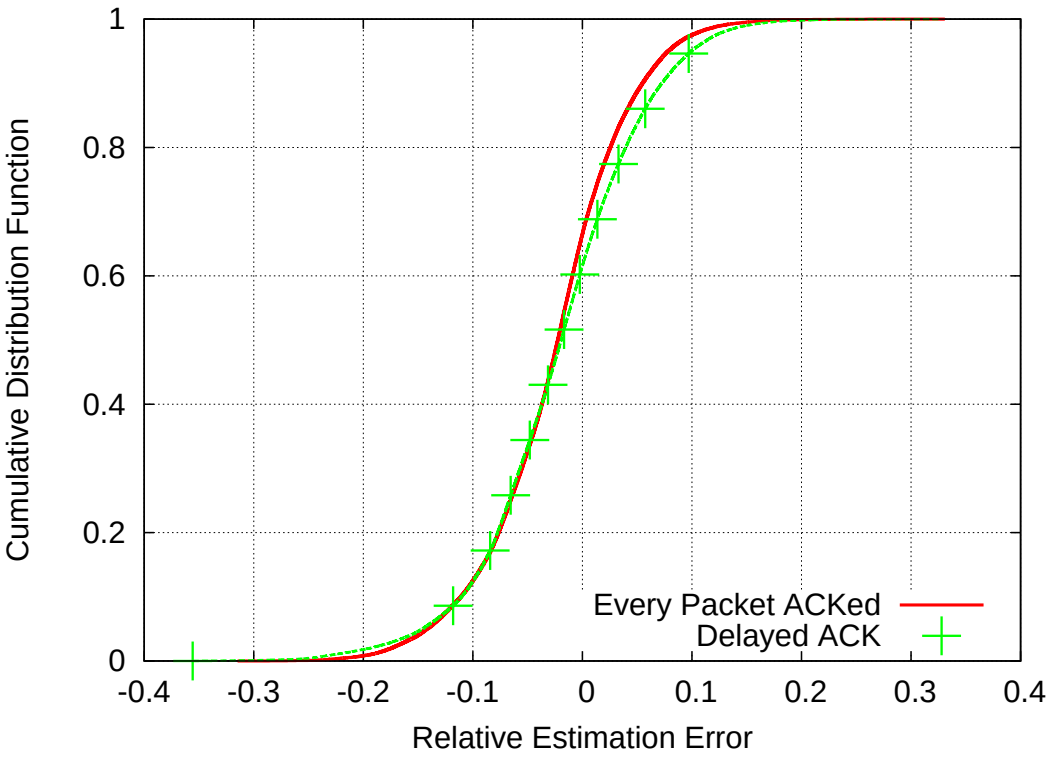


Figure 5.14: Impact of Delayed ACK

5.4 Impact of Smoothing Window

Next we study the impact of the averaging window AVG_WINDOW. Different window sizes, ranging from 4 to 64, are applied to the p-streams collected from the experiments, with $N_r = 5$ and $range = 50\%$. Fig. 5.13 shows how estimation accuracy is affected by different window sizes (avail-bw is calculated with $GAP\_NS\_EPSILON = 100\ ns$). When there is no window-based smoothing ($AVG\_WINDOW = 1$), the 5% to 95% error range may span 16%. AVG_WINDOW values of > 1 achieve highly comparable results. This suggests that AVG_WINDOW has very limited impact on bandwidth estimation accuracy, because the p-streams denoised by BASS are already highly smoothed. For the following experiments, as well as RAPID implementation, we use $AVG\_WINDOW = N_p$.

5.4 Multi-pass BASS with Delayed ACK

For the previous evaluation of BASS on multi-rate p-streams, we observed inter-packet arrival gaps using the Qdisc I_r at the receiver machine—the inter-packet gap of every individual packet was used for bandwidth estimation. However, when we send p-streams in the context of TCP RAPID and collect gap observations at the sender, there is a key difference: the mechanism of delayed acknowledgment is adopted by default, meaning that the receiver generates an ACK covering multiple packets. For kernel 2.6.32 in our testbed, the TCP receiver typically sends an acknowledgment for every other incoming packet. As a result, when the sender calculates inter-packet arrival gaps from timestamps in successive ACKs, it obtains only $\frac{N}{2}$ observations.

Algorithm 6 normalize_pstream(pkts[],N_r,N_p, last_acked)

```
1: N = N_r × N_p
2: sum_sendgap=sum_recvgap=0
3: norm_streamcnt=cnt=0
4: for i in 0:N-1 then
5:   if pkts[i] is acked then
6:     cnt++
7:     sum_sendgap+=pkts[i].sendgap
8:     sum_recvgap = pkts[i].seqno - last_acked
9:     for j in 0:cnt-1 then
10:      norm_sendgap[norm_streamcnt]=sum_sendgap/cnt
11:      norm_recvgap[norm_streamcnt]=sum_recvgap/cnt
12:      norm_streamcnt++
13:    sum_sendgap=sum_recvgap=0
14:    cnt=0
15:    last_acked = pkts[i].seqno
16:  else
17:    sum_sendgap+=pkts[i].sendgap
```

To address this difference, we extend the $\frac{N}{2}$ observations to the original p-stream lengths N using the logic in Algorithm 6. The function $normalize\_pstream$ is called once all packets in a p-stream are acknowledged. It counts the number of packets cnt covered by each ACK, calculates gaps between successive ACKs, and divides them by cnt. The normalized observation is then copied for cnt times. Eventually, the normalized p-streams contain N packets and are fed to the bandwidth estimation logic. Fig. 5.14 demonstrates the impact of delayed ACK using the chosen parameter set; p-streams with delayed ACK achieve comparable accuracy with those that have every packet acknowledged.

5.5 Summary

In this chapter, we aimed to achieve robust bandwidth estimation for TCP RAPID on ultra-high speed links, using the shortest p-streams possible. We first evaluated state-of-the-art mechanisms to deal

with noise in bandwidth estimation: IMR-avg and IMR-wavelet yielded considerable error, and PRC-MT performed well only with extremely long p-streams, of 320 packets. Thus, none of the existing mechanisms met the requirements for RAPID.

We developed a novel denoising technique, BASS, which identifies those packets that belong to one buffering event and smooths out delay observations within each. BASS was then applied to single-rate probing, and we found that it enables short p-streams of $N = 64$ to correctly estimate 90% of p-streams. For multi-rate p-streams, we developed multi-pass BASS and examined the impact of several parameters on its estimation accuracy. We concluded that multi-pass BASS addresses the dilemma described earlier in this chapter: it yields robust estimation while simultaneously limiting the timescale during which each p-stream overloads the path by using short p-streams of $N = 64$. We also evaluated BASS in the context of TCP RAPID, which uses delayed ACK. Based on these evaluations, we employ the following parameters for p-streams in the implementation of RAPID: ($range = 50\%, N_r = 4, N_p = 16, AVG\_WINDOW = 16, GAP\_NS\_EPSILON = 100$).

CHAPTER 6: A MACHINE-LEARNING SOLUTION FOR BANDWIDTH ESTIMATION

In Chapter 5, we evaluated existing algorithms designed to deal with noise on ultra-high speed links. We also successfully developed the BASS algorithm, which has been shown to achieve robust bandwidth with p-streams as short as 64 packets. It is important to note that noise can distort gaps within a p-stream that has several different signatures, each with its own magnitude of gap-distortion; as shown in Fig. 1.2, different levels of burstiness in cross-traffic yield different heights of "spikes" in the distorted gaps. And each source of noise manifests itself at its own frequency. As shown in Fig. 1.2 and Fig. 1.3, packets in a p-stream may consistently encounter cross-traffic noise in short timescales, and they may be distorted more drastically by interrupt coalescence at larger timescales, depending on the interrupt delay setting of the receiving NIC. When simple denoising heuristics are used by state-of-the-art techniques for dealing with such diversity in noise, they result in an *underfit* model—these techniques either lead to severe estimation errors (e.g. IMR-Pathload), or must smooth over a *large* number of probe packets in order to be robust (e.g. PRC-MT). Even BASS may cause up to 15% error and is sensitive to the parameters used by the denoising algorithms.

In recent years, machine learning has been shown to outperform human-crafted algorithms in networking research fields including traffic classification [80], intrusion detection [81], and congestion control [82]. The main hypothesis of this chapter is that machine learning can understand the noise signature in gaps better than human-designed denoising algorithms, using even shorter probe streams than the current state of the art.

Goal In this chapter, we propose to use supervised learning to automatically derive an algorithm that estimates avail-bw from the inter-packet send and receive gaps of each p-stream. Such an algorithm is referred to as a learned "model." We envision that the model be learned *offline* and then incorporated into the avail-bw estimation processes.

Section 6.1 describes how we obtain models for both single-rate and multi-rate probing, and in Section 6.2 we compare the performance of these models with the best performing human-designed algorithm, BASS.

6.1 A Learning Framework

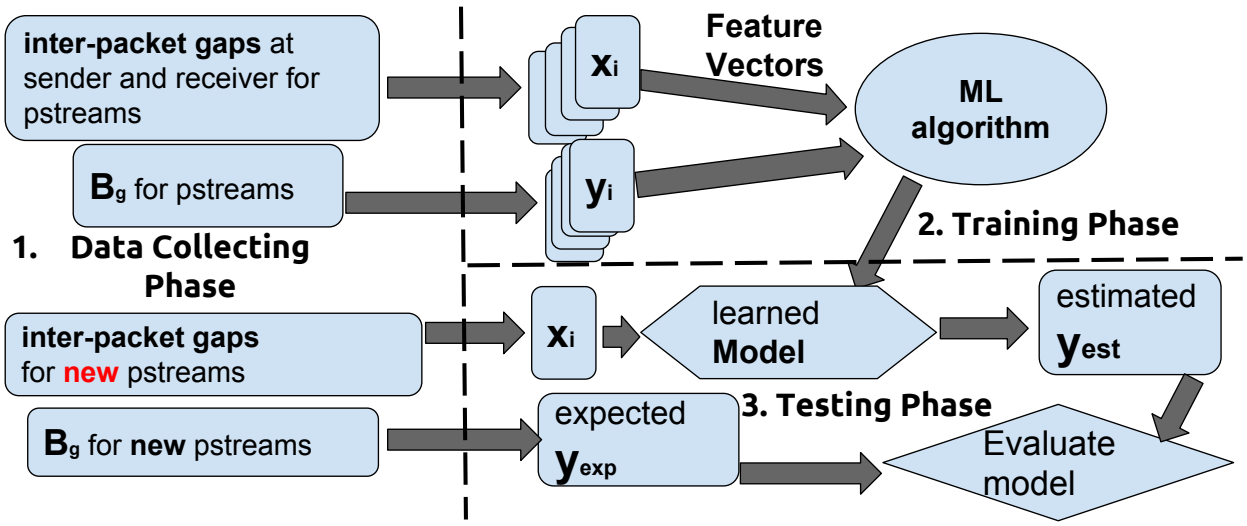


Figure 6.1: Machine Learning Framework For Bandwidth Estimation

Below, we describe the key components of this machine-learning-based framework for bandwidth estimation. As plotted in Fig. 6.1, a typical machine learning process consists of three phases. In the data collecting phase, we conduct experiments and produce multiple data instances (X, Y), which become the input of the next training phase. The input set is referred to as a "training set." This training set will be fed into a machine learning algorithm that will automatically investigate the relationship of X to Y and generate a learned model. Finally, in the testing phase, we evaluate the quality of the learned model using another set of data, different from the training set; this is referred to as a "testing set." We feed its X vector to the resultant model and compare the model output with that of the Y vector in the testing set.

6.1 Input Feature Vector

The input feature vector for a p-stream is constructed from the set of send gaps and receive gaps, $\{g_i^s\}$ and $\{g_i^r\}$. Fourier transforms are commonly used in machine learning applications when the input may contain information at multiple frequencies [83, 80], and this condition certainly holds for g_r, which are distorted by different sources of noise on a network path. Therefore, we use as a feature vector the Fourier-transformed sequence of send and receive gaps for a p-stream of length N: $x = FFT(g_1^s, ..., g_N^s, g_1^r, ..., g_N^r)$.

6.1 Output

The output y of the machine learning framework is the result of bandwidth estimation for each p-stream. For *single-rate* p-streams, bandwidth estimation can be formulated as a classification problem, which maps the output to a set of discrete values. For bandwidth estimation in our case, $y = 1$ if the probing rate exceeds avail-bw; otherwise, $y = 0$. For *multi-rate* p-streams, bandwidth estimation can be formulated as a regression problem, which maps the output to non-discrete values. In our case, $y = B_g$.

6.1 Machine-learning Algorithms

We consider several common machine-learning algorithms: ElasticNet [55], RandomForest [84], AdaBoost [57], and GradientBoost [58]. Below we briefly describe each of these algorithms.

- ElasticNet: ElasticNet is in the family of linear learning methods, which assume a linear relationship between the input feature vectors x and the output vector y. In other words, y can be expressed as the sum of polynomial terms of x. The training task of the ElasticNet algorithm is to figure out the polynomial coefficients that best characterize the relationship between x and y. Other algorithms in the linear family include Lasso [56] and Ridge [85]. We chose to apply ElasticNet because it has been claimed to overcome the shortcomings of the other two and to generally outperform the others [55].

- RandomForest: RandomForest, AdaBoost, and GradientBoost all follow the principle of ensembling simple, relatively inaccurate models (usually decision-tree based [86]) into a more robust model. But RandomForest differs from the other two in how it ensembles the weak models: it separates the training set into multiple non-overlapping subsets by random, and a weak model is trained on each subset. For each input x, the ensembled RandomForest model then produces y as some weighted mean of the results from all weak models.[1]

- AdaBoost: As stated before, AdaBoost is also based on multiple weak models. It constructs the combined model by perfecting weak models in an iterative and incremental fashion. It begins with a simple decision-tree model, and in each subsequent iteration another model is trained with the goal of compensating for the shortcomings of previous models. To achieve this goal, each new model

[1]The computation of these weights, which is conducted by the internal algorithm of each learning algorithm, depends on the correctness of each weak model.

emphasizes the training data on which previous models fell short, instead of using a random subset of data, as RandomForest does. Consequently, after each iteration, the combined model becomes more robust. Finally, the ensembled model computes as the output the weighted average of the results of all ensembled weak models.

- GradientBoost: GradientBoost follows the same methods of ensembling multiple weak models as AdaBoost, but it differs in how it measures the accuracy of a model and how it computes weight coefficients when optimizing the robustness of the combined model at each iteration. AdaBoost relies on exponential loss function and GradientBoost on gradient descent. Please refer to [87, 57, 58] for more detail.

6.1 Data Collection

The success of any machine-learning framework depends heavily on good data collection; the framework must collect data that is both accurate and representative. The knowledge of AB_{gt} allows us to compute an expected value y_{exp} of the output of the machine-learning framework for both single-rate and multi-rate p-streams.

P-streams are generated with the "Dummy-frame" mechanism described in Chapter 4, which shares the 10 Gbps bottleneck link with cross-traffic of different burstiness levels. Some of these p-streams are used as the "training set" to train each of the learning techniques. The p-streams excluded from the training set serve as the "testing set" on which the quality of learned models will be evaluated. We re-use the p-stream observations from Section 5.3, which were obtained to evaluate BASS, as the source of p-streams for both our training and testing data sets. In each experiment below, we use more than 10 000 p-streams, among which 5 000 are used for training and the rest for testing.

6.1 Training

We implement the training phase with the Python scikit-learn [88] library, which offers abundant interfaces for different machine learning algorithms as well as parameter tuning. We use its automatic parameter tuning feature for all machine learning methods and use 5-fold cross-validation to validate our results.

6.1 Metrics

Each test that is run on a p-stream yields an estimate of the output y. For a single-rate p-stream, the accuracy of the model is quantified by the *decision error rate*, which is the percentage of p-streams for which $y \neq y_{exp}$. For multi-rate p-streams, we quantify *relative estimation error* as $e = \frac{y - B_g}{B_g}$.

6.2 Evaluation: Single-rate Probing

As stated in Section 1.2.2.2, the two major sources of noise considered are cross-traffic burstiness and receiver-side interrupt coalescence. In this section, we first present experiments conducted under conditions similar to those used to evaluate BASS: BCT cross-traffic and default configuration of interrupt coalescence on NIC1. Later, in the multi-rate probing framework adopted by TCP RAPID, we explicitly control for and consider the impact of cross-traffic burstiness and interrupt coalescence.

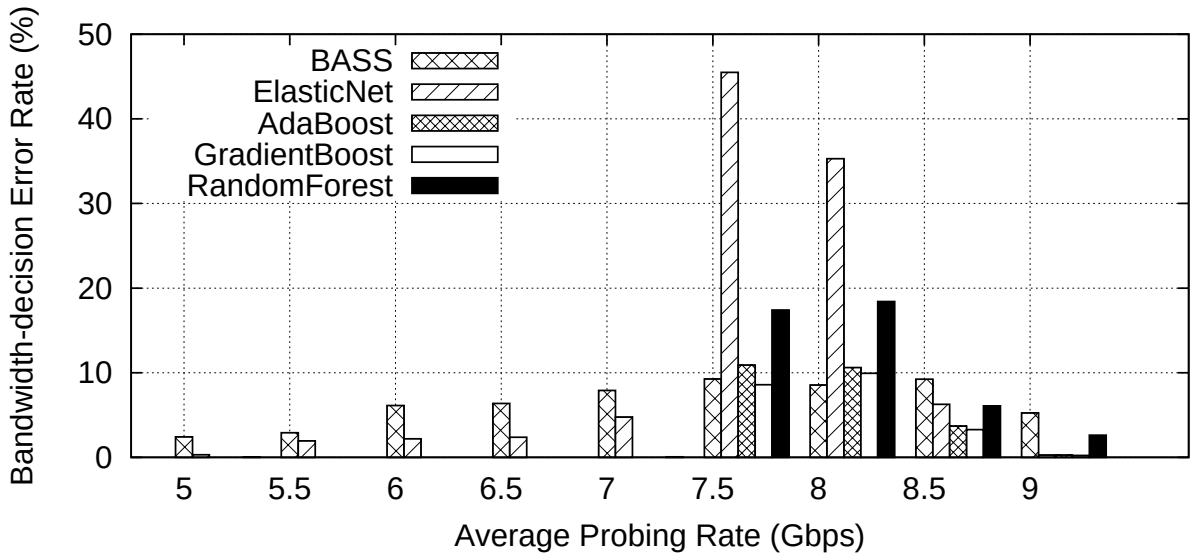


Figure 6.2: Estimation Error of Single-Rate Probe Streams (N=64)
For each probing rate, plot the percentage of probe streams yielding wrong decision of whether the probing rate exceeds avail-bw. Machine-learning approaches lead to more accurate estimation than BASS.

6.2 N = 64

Recall that BASS is able to yield an estimation accuracy of over 90% with $N = 64$. We first use the same N value to train models of different machine learning algorithms; we then test them on p-streams probing at 9 discrete rates, ranging from $5 - 9$ Gbps. Probe-streams in both training and testing sets share the 10 Gbps path with BCT. The bandwidth-decision errors observed at each rate are plotted in Fig. 6.2. We find that, unlike BASS, each of the three ensemble methods (AdaBoost, GradientBoost, and RandomForest) leads to negligible error when the probing rate is far below or above avail-bw. When probing rates are close to the AB, both BASS and the machine learning models yield more ambiguity. AdaBoost and GradientBoost perform comparably to BASS; RandomForest performs worse than the two boosting methods, a result that agrees with the findings in [89]. This may be because each weak model in RandomForest is learned on a different subset of training data and the final prediction is the average result of all weak models; AdaBoost and GradientBoost follow a boosting approach, in which each model is built to emphasize the training instances that previous models did not handle well. The boosting methods are known to be more robust than RandomForest [89] when the data has few outliers. Among all machine learning algorithms, ElasticNet performs worst, giving extreme high inaccuracy over 40%. It tries to formulate a polynomial relationship between X and Y, but however intuitive the idea may be, there exists no such linear relationship with a fixed set of co-efficients that can be applied to universal p-streams. The position and the height of each spike, as well as the spaces between spikes, are unpredictable, depending completely on the characteristics of the noise encountered by each individual p-stream.

6.2 N = 48,32

We then consider shorter p-streams by reducing *N* to 48 and 32, respectively; we compare the accuracy using BASS and the machine-learning approach in Fig. 6.3 and Fig. 6.4. In Fig. 6.3 the performance of BASS degrades drastically with reduced N: for $N = 32$, the error rate can exceed 50% when the probing rate is higher than 8 Gbps. Although the three ensemble methods (AdaBoost, GradientBoost, and RandomForest) also yield more errors with smaller N, their error rates are limited to within 20% even with $N = 32$. Their performances are very similar.

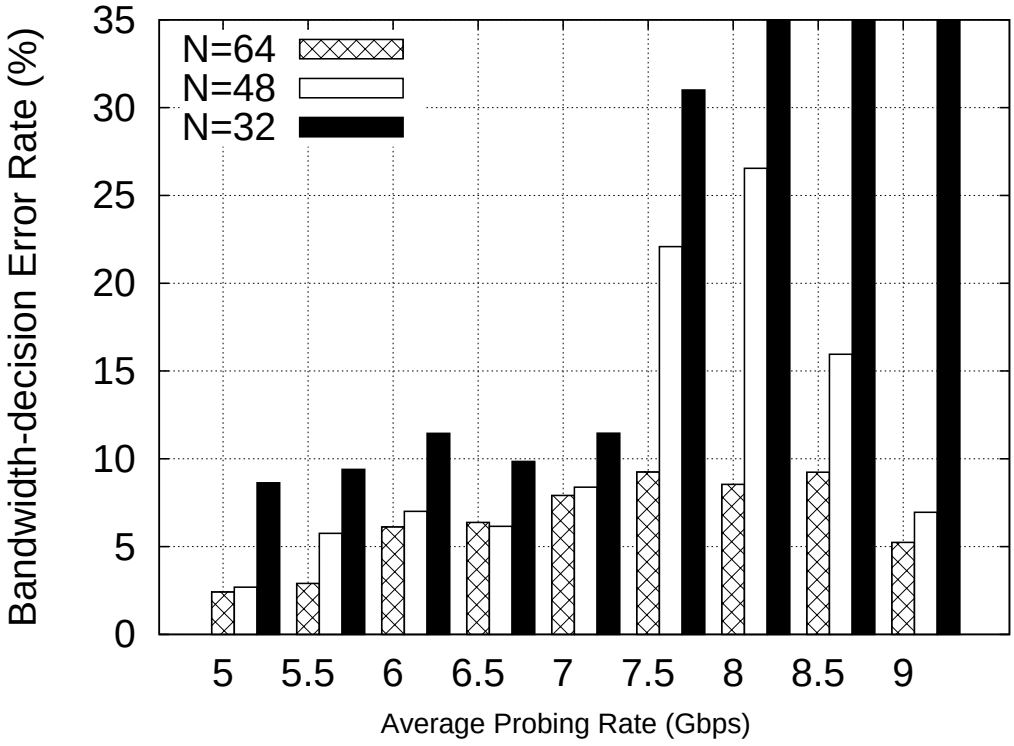


Figure 6.3: Single-Rate Estimation Error Using BASS (Smaller N)

6.3 Evaluation: Multi-rate Probing

Next we consider applying machine learning techniques to the multi-rate probing framework that is used in the TCP RAPID. We train models with multi-rate p-streams of $N = 64, N_r = 4$, and probing $range$ 50%. P-stream observations for both the training and testing data sets are obtained experimentally by sending p-streams with BCT, with the default interrupt coalescence on NIC1 at the receiver. Fig. 6.5(a) plots the distributions of relative estimation error when BASS is used with different p-stream lengths ($N = 96, 48, 32$), and Fig. 6.5(b) to Fig. 6.5(e) plot the relative estimation errors of the machine-learned models. Machine learning significantly outperforms BASS, limiting error to within 10% for over 95% p-streams.

We further reduce N to 48 and 32 and find that, while a smaller N produces a higher degree of inaccuracy for BASS, it has limited impact on the machine learning-based approach. The learned models maintain similar accuracies at $N = 48$ and at $N = 64$; they are slightly more error-prone at $N = 32$.

Based on our experiments so far, we conclude that *our machine learning framework is capable of estimating bandwidth with higher accuracy using shorter p-streams than the current state-of-the-art approaches, both with single-rate and multi-rate probing techniques*.

6.3 Impact of Cross-traffic Burstiness

We next consider the impact of prominent sources of noise, namely cross-traffic burstiness and receiver-side interrupt coalescence. In what follows, we focus on multi-rate probing with $N = 48$ and $N_r = 4$, and use only GradientBoost, the best-performing machine learning algorithm.

We repeat the experiments in Fig. 6.5, with BCT replaced by each of the other five models of cross-traffic, each with different burstiness levels. In other words, the machine learning model is trained and tested using p-streams that encounter the *same* type of cross-traffic model. Fig. 6.6(b) plots the results; the boxes plot the $10 - 90\%$ range of the relative estimation error, and the extended bars plot the $5 - 95\%$ ranges. For each cross-traffic type, the left two bars compare the performance of BASS with our machine learning model. We find that the performances of both BASS and our machine learning model are relatively insensitive to the level of burstiness in cross-traffic. However, in each case, machine learning consistently outperforms BASS.

In practice, it is not possible to always predict how bursty the cross-traffic on a given network path is. The more bursty the traffic, the more likely it is to distort gaps to different degrees. Intuitively, a model learned from bursty cross-traffic seems more likely to efficiently handle real-world cases where traffic is bursty; however, models trained on bursty data may be more subject to overfitting—the model may try to "memorize" the noisy training data, leading to poor performance in conditions with smoother traffic. On the other hand, a model learned from smooth cross-traffic may perform terribly for p-streams encountering bursty traffic, because the training set excludes signatures of severe noise. We next ask the following question: *how does our machine learning framework perform when it encounters different types of burstiness in the training and testing phases?*

Training with Smoother Traffic

First, we train models on the several types of cross-traffic at various levels of burstiness: BCT, SCT, CBR, and UNC1-3, with BCT the most bursty and CBR the smoothest. Then we apply the learned models, using them to test p-streams that encounter the more bursty BCT; see Fig. 6.6(a). We find that machine learning outperforms BASS in all cases. However, models learned with smoother traffic lead to slightly more errors than the model learned with BCT. This is to be expected, because bursty traffic introduces a higher degree of noise. We conclude that it is *preferable to train a machine learning model with highly bursty cross-traffic in order to prepare it for traffic occurring in the wild.*

Training with Bursty Traffic

We use the model trained with BCT to predict avail-bw for p-streams that encounter other types of cross-traffic. In Fig. 6.6(b), we see that the BCT-derived model is as accurate for each type of cross-traffic as the model trained on that type of cross-traffic; the model trained on BCT does as well, for example, on the SCT testing set as the model trained on SCT, and as well on the CBR testing set as the model trained on CBR, and so on. Thus, *a model learned from bursty cross-traffic is robust* to testing cases of less bursty cross-traffic.

6.3 Impact of Interrupt Coalescence Parameter

Interrupt coalescence by a NIC platform is typically configured using two types of parameters (*ICparam*): "rx-usecs," the minimum time delay between interrupts, and/or "rx-frames," the number of packets coalesced before the NIC generates an interrupt. By default, NICs are configured to use some combination of both of these parameters—our experiments presented so far use the default configuration on NIC1, which roughly boils down to a typical interrupt delay of about 120 μs. Different $ICparam$s lead to different "spike-dips" patterns in the receive gaps; these patterns differ both in the heights of the spikes and in the distances between the neighboring spikes. For instance, Fig. 6.7 plots three p-streams with identical inter-packet gaps at the sender side that were received with different $ICparam$ on NIC1. $ICparam = 200\mu s$ leads to fewer, taller spikes than $ICparam = Default$, while $ICparam = 10\ \mu s$ yields no significant spikes. Because a model learned with one parameter may fail on p-streams that encounter another, we next study the impact of having different $ICparam$s in the training and testing data sets on the two NIC (labeled as NIC1 and NIC2) platforms available in our testbed.

6.3 NIC1

With NIC1 coalescence parameters set to 2 μs to 300 μs at the receiver side, respectively, we generate p-streams sharing the path with BCT and collect the corresponding g_s and g_r as testing sets. The previously learned GradientBoost model, using *ICparam* = *default*, is applied to each of these testing sets for bandwidth estimation. Fig. 6.8 compares the estimation accuracy of BASS (the first bar) and the machine learning model (the second bar, marked as "Default"). The box plots the $10\% - 90\%$ relative error and the extended bar plots the $5\% - 95\%$ error. We find that BASS severely over-estimates avail-bw when interrupt delay is significant (rx-usecs $\geq$ 200 μs); the model learned with default $ICparam$ yields much better accuracy. However, we find that the "Default" mode l consistently under-estimates avail-bw when rx-usecs = 300 μs.

Machine learning performs best when the training set is similar to the testing set. To achieve this, we create a training set that includes 5 000 p-stream samples for each *ICparam* value and denote this training set as "ALL-set." As shown in Fig. 6.8, the model learned from "ALL-set" reduces error to within 10% for most p-streams that encounter extreme rx-usecs values.

In practice, however, *all* possible configurations of $ICparam$ at a receiver NIC may not be known. We next ask: does a model exist that can be trained with only a limited set of $ICparam$s and that is generalizable to all configurations? To study this, we minimize the training set to only include two extreme values (2 us and 300 us) in addition to the default setting. We refer to this set as "3sets." Fig. 6.8 shows that "3sets" is sufficient to train an accurate machine learning model that provides similar accuracy to "ALL-set."

6.3 NIC2

Different NIC platforms may interpret and implement interrupt coalescence differently. For instance, NIC2 relies on an adaptive interrupt behavior even though it allows us to specify "rx-usecs" and "rx-frames." Fig. 6.9 illustrates this by plotting the distribution of the number of frames coalesced per interrupt. We find that "rx-frames" has no effect when rx-usecs $\leq 12\ \mu s$. But "rx-usecs" is not respected once it exceeds 12 μs; the distribution mainly depends on rx-frames, which presents a much more diverse and unpredictable interrupting behavior. We next examine whether our machine learning framework will also work on such a NIC.

We repeat the experiments of Section 6.3.2.1, but we use NIC2 instead of NIC1 for collecting both the training and testing data. We consider the following $ICparam$s for NIC2: rx-usecs from 2 to 10 μs and rx-frames from 2 to 20 (rx-usecs = 100). Models are learned from training sets consisting of different combinations of $ICparam$s in training scenarios: the "Default," "ALL-set," and the "3sets"(default, rx-frames = 2, and rx-frames = 20). Fig. 6.10 plots the estimation errors for these three environments. Compared with Fig. 6.8, the estimation error is generally higher on NIC2 than NIC1, presumably due to greater unpredictability in the interrupting behavior of NIC2. As before, the "3sets" on NIC2 significantly outperforms BASS and gives comparable accuracy to "ALL-set," agreeing with our observation about NIC1.

6.3 Portability

To investigate the portability of a learned model across NICs, we next perform a cross-NIC validation: the model trained with data collected using one NIC is tested on data collected with the other NIC. We use

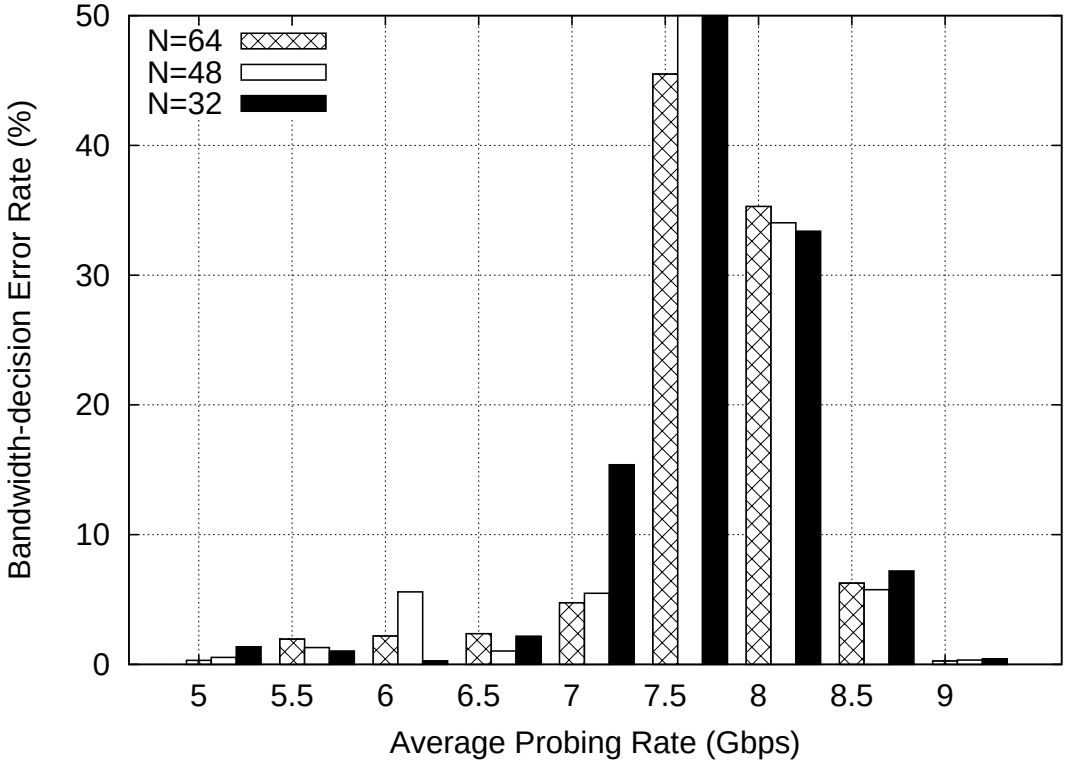
only *ICparam = 3sets*, and we plot the results for NIC1 in Fig. 6.11(a) and and NIC2 in Fig. 6.11(b). The black bar represents the estimation error for the model trained with "3sets" on the same NIC; the white bar represents the estimation error for the model trained with "3sets" using the other NIC. For both NICs, these two bars have comparable heights, indicating the models have comparable accuracy across both NICs, regardless of what data they were trained on. The notable exceptions occur for extreme values of *ICparam* of *rx-usecs = 300* μs on NIC1 and *rx-frames = 20* in NIC2. In general, the machine-learned models on one NIC are portable and yield robust estimation on different NIC platforms if the interrupt coalescence parameter is set within a moderate range.

6.4 Summary

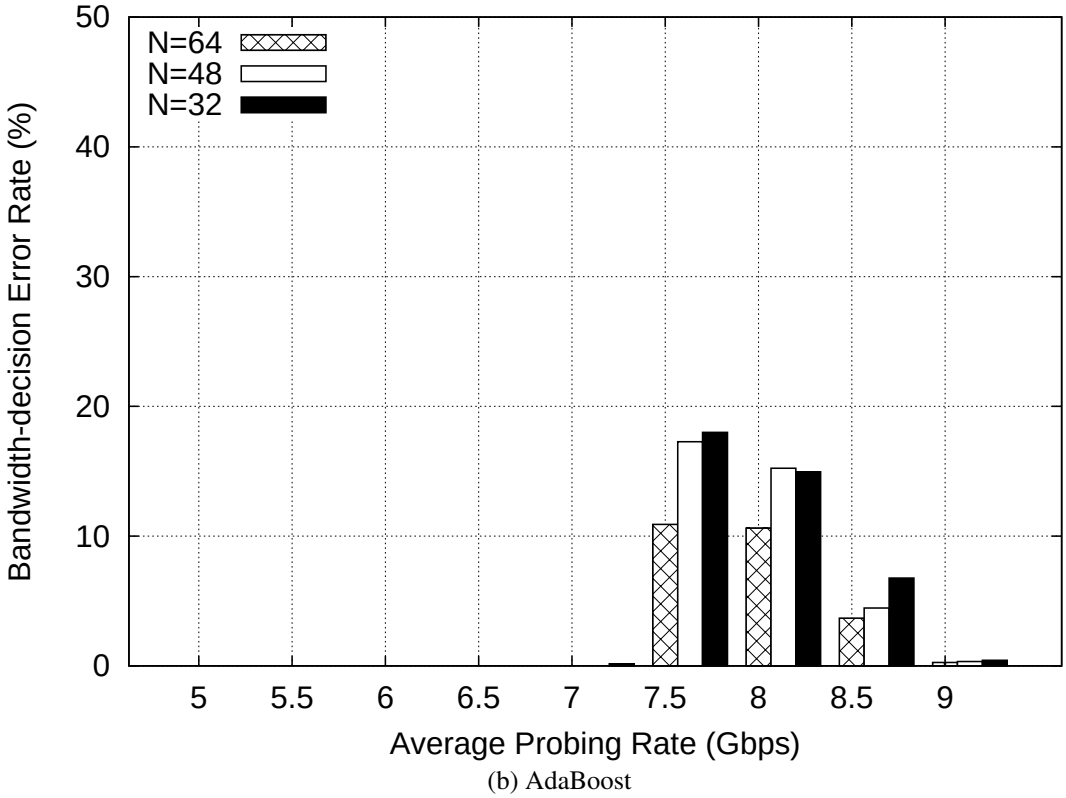
In this chapter, we borrowed ideas from the area of machine learning and developed a machine learning framework to automatically generate models for bandwidth estimation. Compared with BASS in Chapter 5, supervised learning helps to improve estimation accuracy for both single-rate and multi-rate probing frameworks, and is useful at shorter p-streams with $N = 48$ than BASS. Further experiments showed that:

- a model trained with more bursty cross-traffic is robust to traffic burstiness;
- the machine learning approach is robust to interrupt coalescence parameters, if default and extreme configurations are included in training; and
- the machine learning framework is portable across different NIC platforms.

However, as far as we know, there is no kernel-friendly implementation of such a machine learning framework. Therefore, we do not presently consider it in this implementation of RAPID , but we do encourage future researchers to consider the approach if any kernel-friendly implementation of a machine learning library is available in the future.

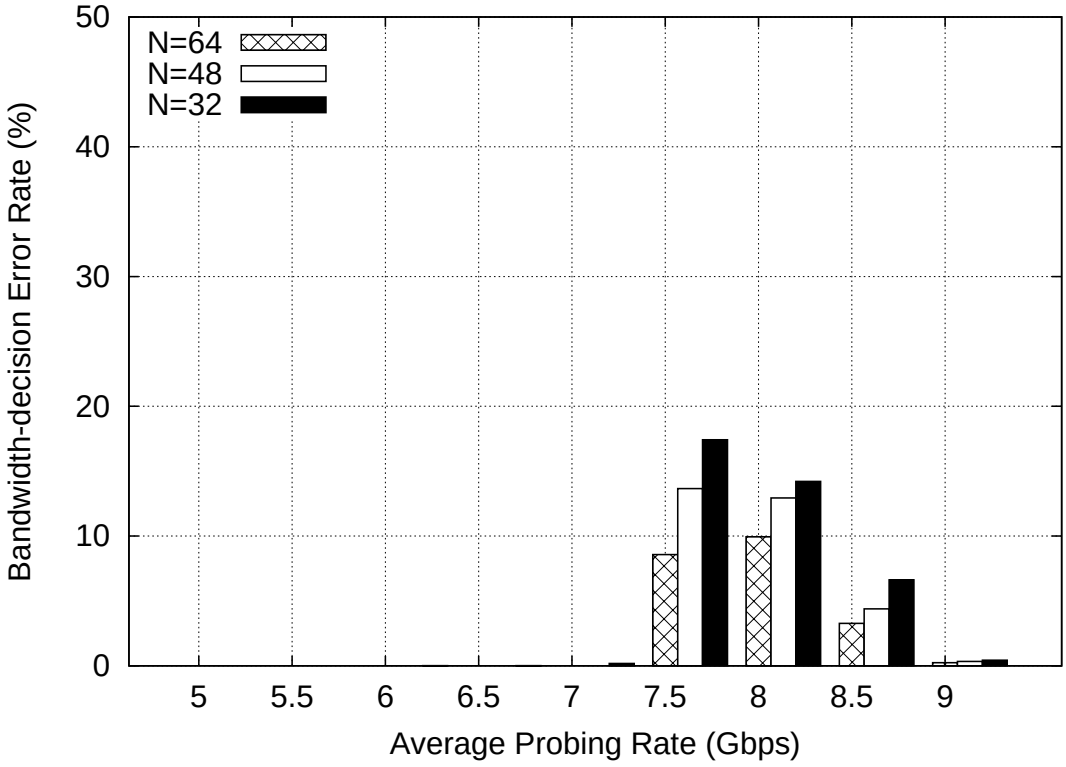


(a) ElasticNet

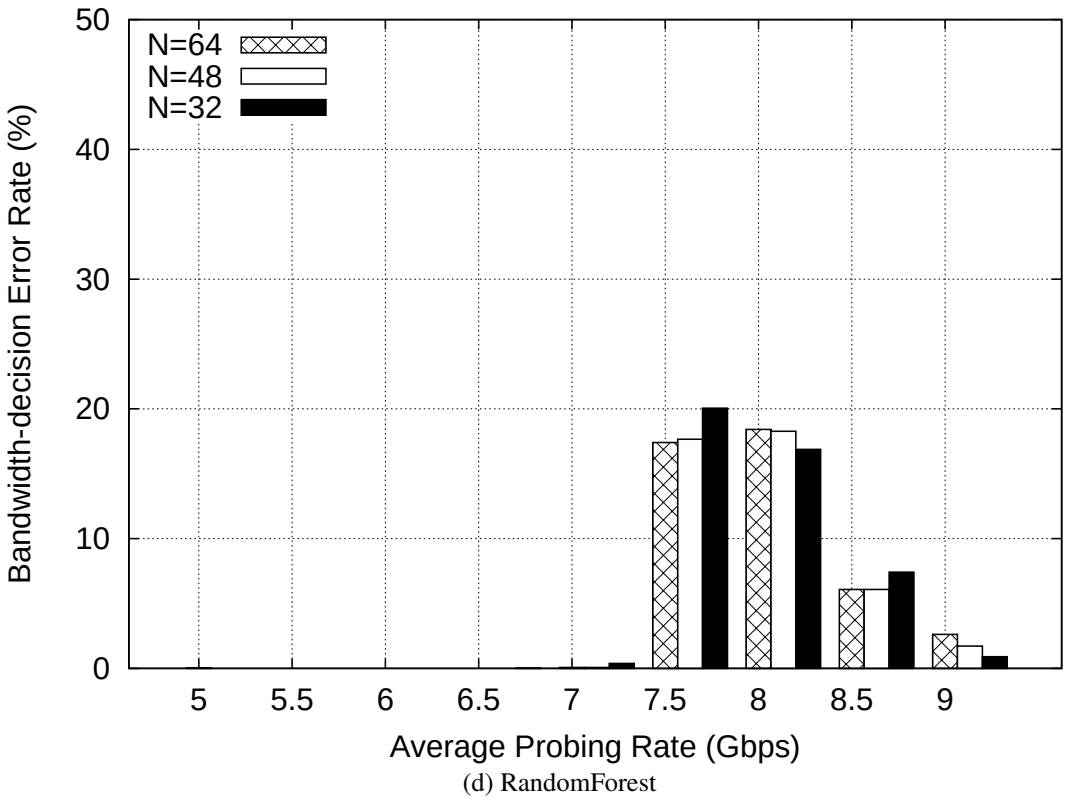


(b) AdaBoost

Figure 6.4: Single-Rate Estimation Error Using Machine-Leaning (Smaller N)

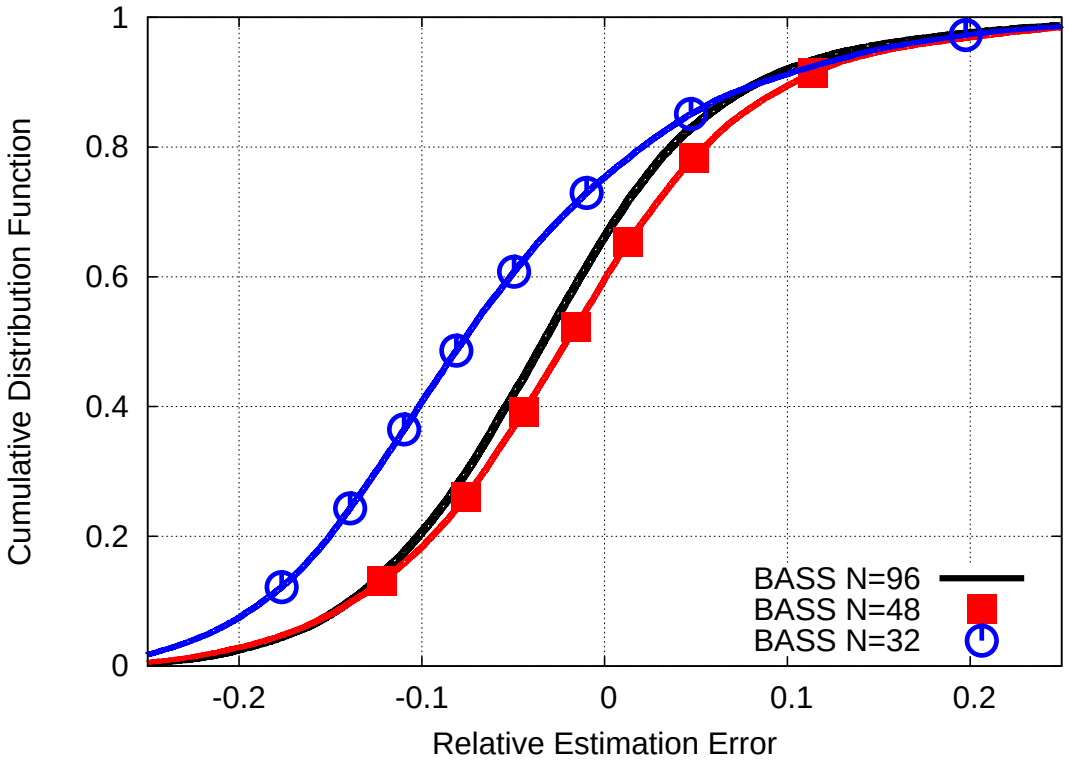


(c) GradientBoost

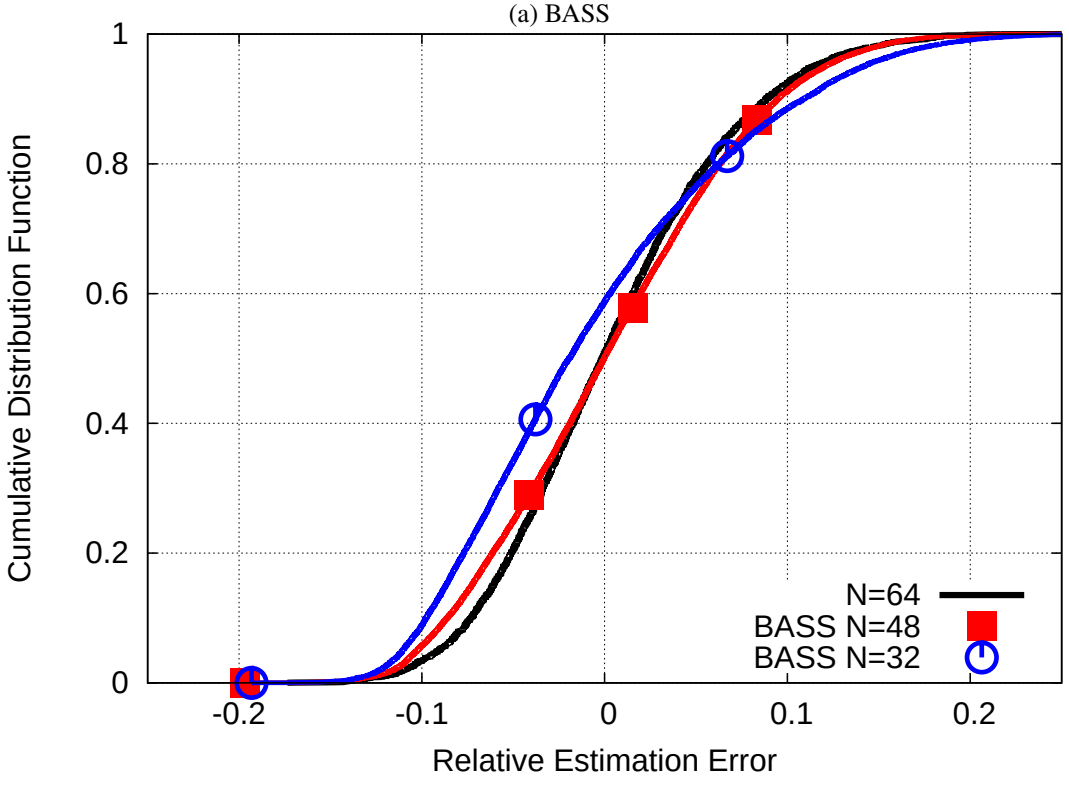


(d) RandomForest

Figure 6.4: Single-Rate Estimation Error Using Machine-Leaning (Cont.)

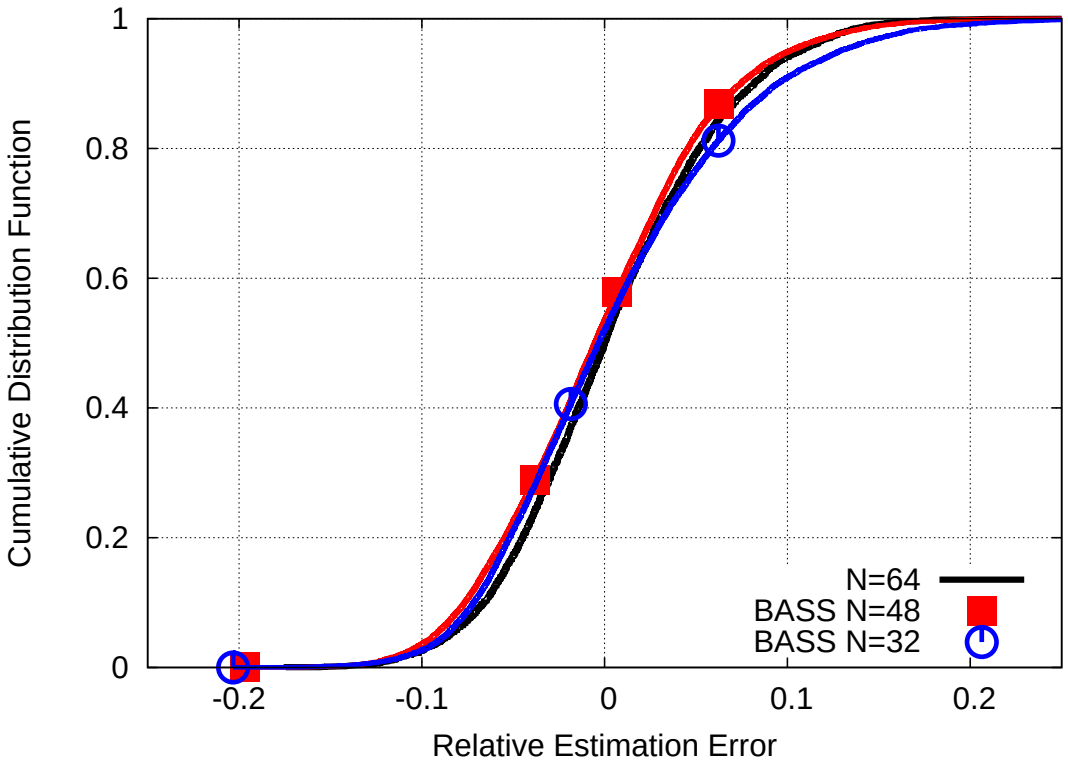


(a) BASS

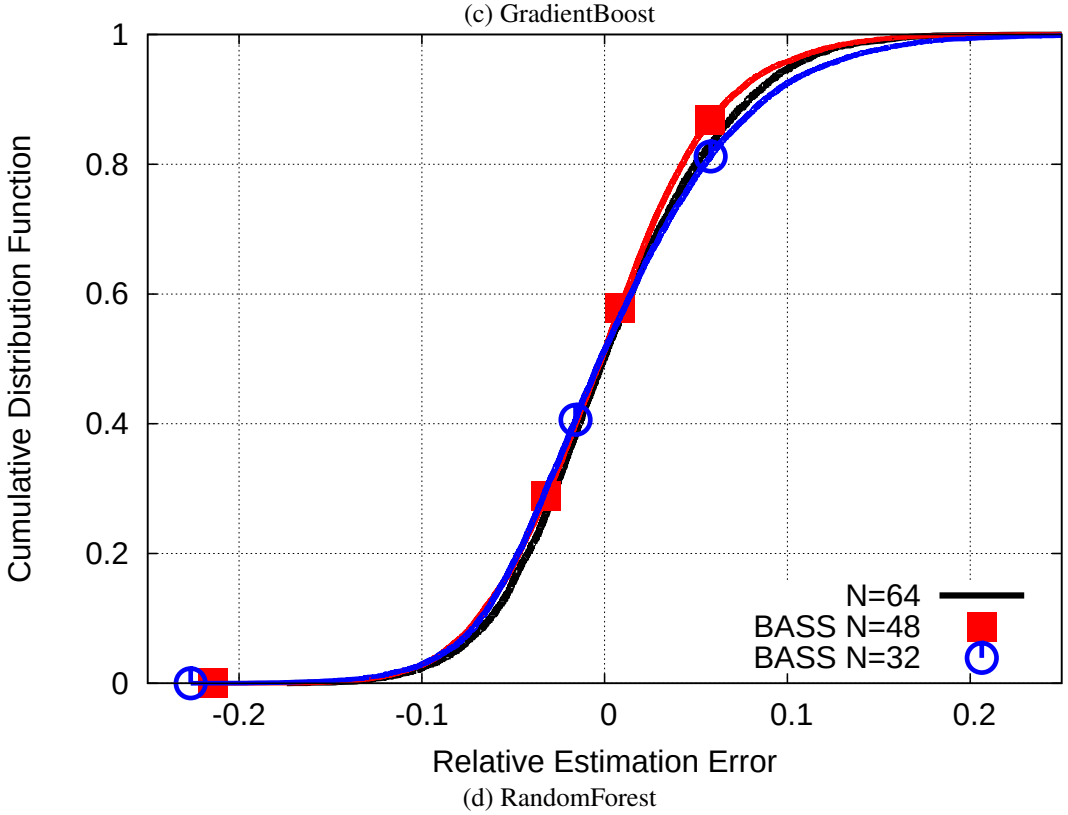


(b) AdaBoostl

Figure 6.5: Multi-rate Estimation Error with BASS and Machine-Learning Models
Cumulative distribution function of relative-estimation error rate for multi-rate probe streams is plotted. The machine-learning approach yield more accurate estimation than BASS, showing more vertical curves.

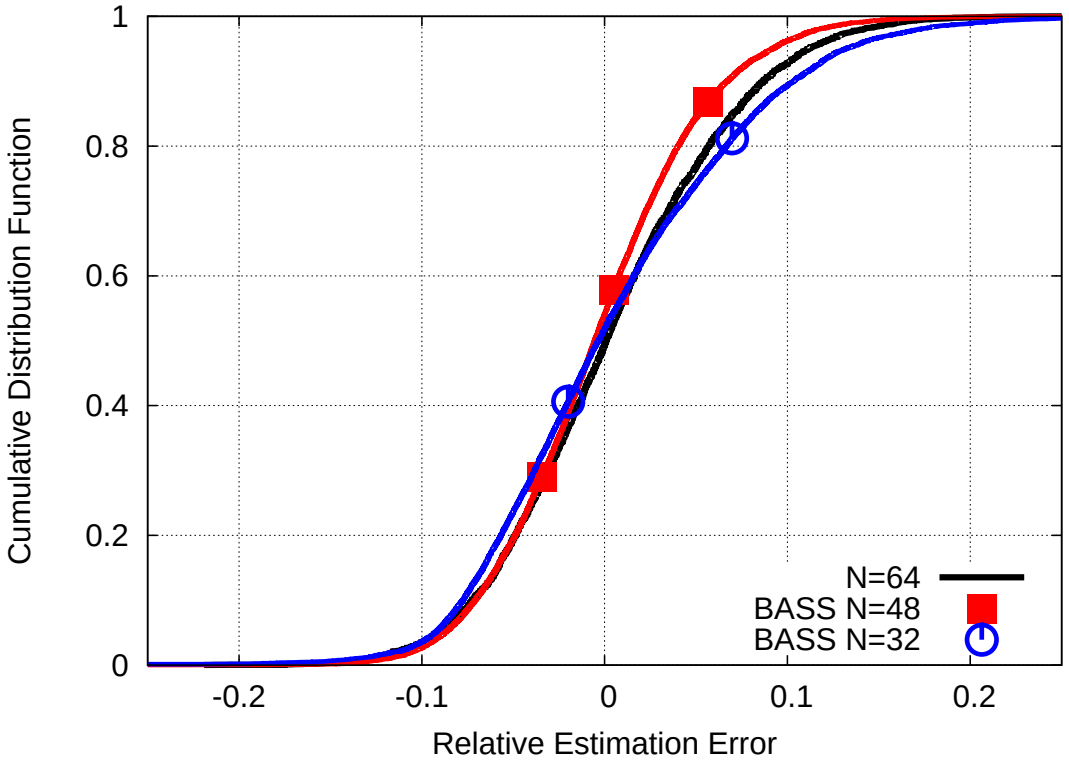


(c) GradientBoost



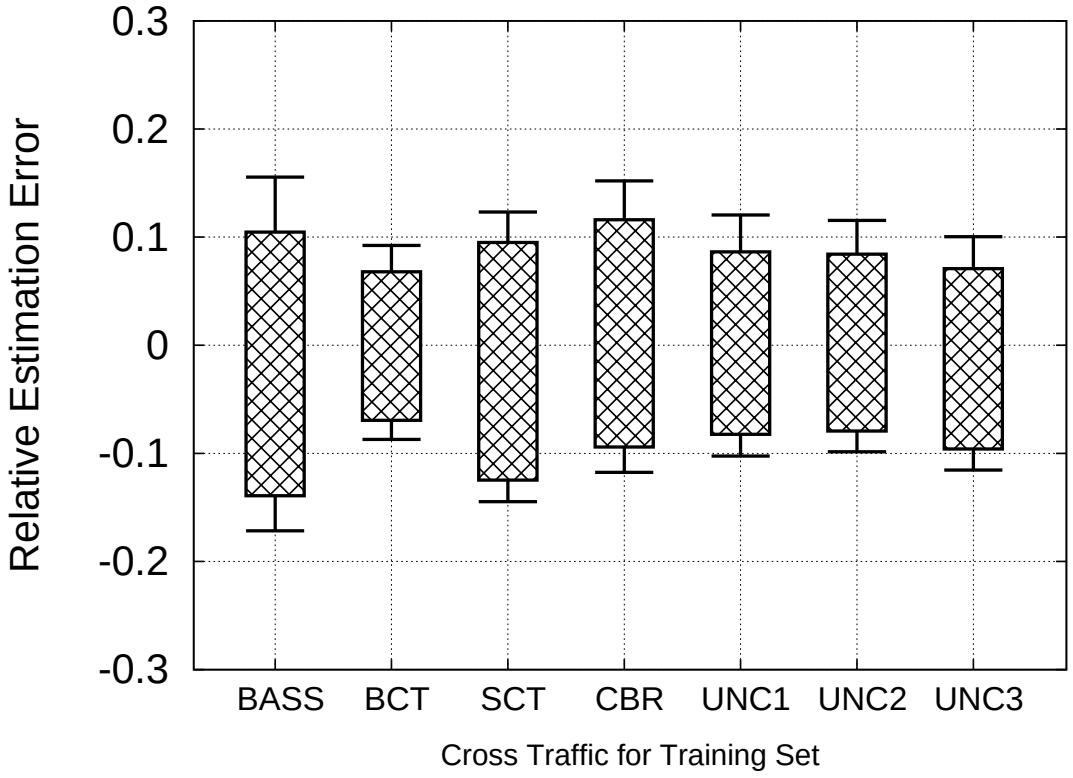
(d) RandomForest

Figure 6.5: Multi-rate Estimation Error with BASS and Machine-Learning Models(Cont.)

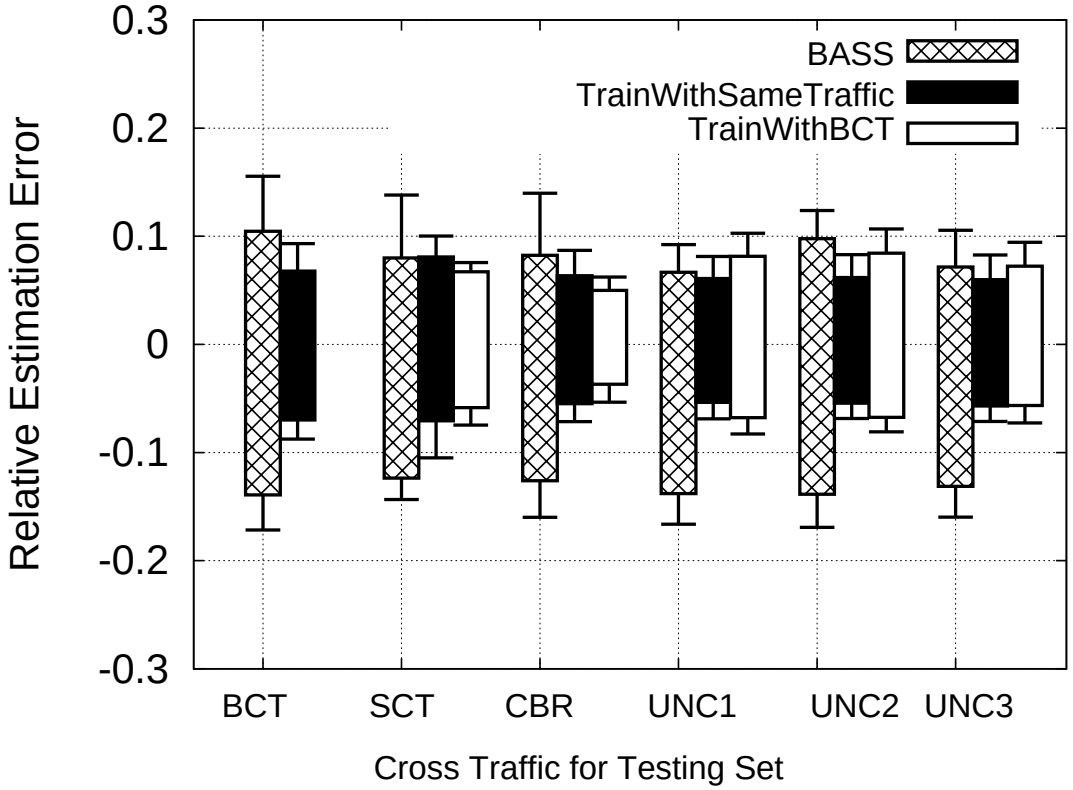


(e) ElasticNet

Figure 6.5: Multi-rate Probing with BASS and machine learning Models (Cont.)

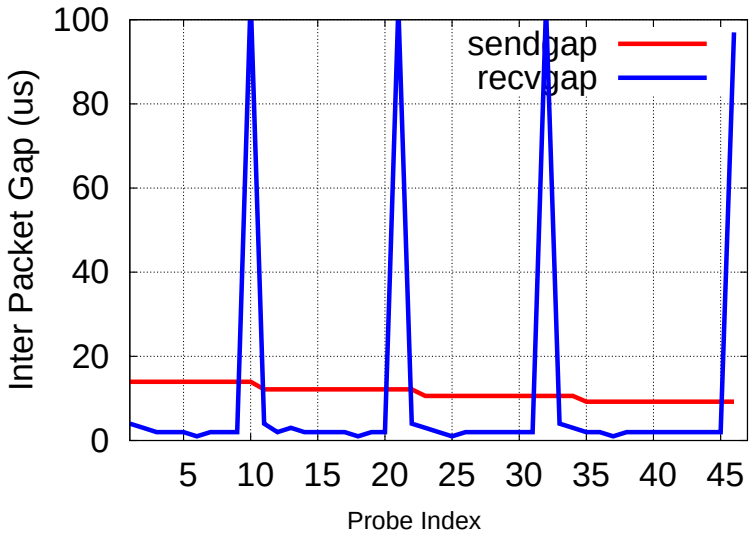


(a) Train with Smoother Traffic (test on the most bursty traffic)

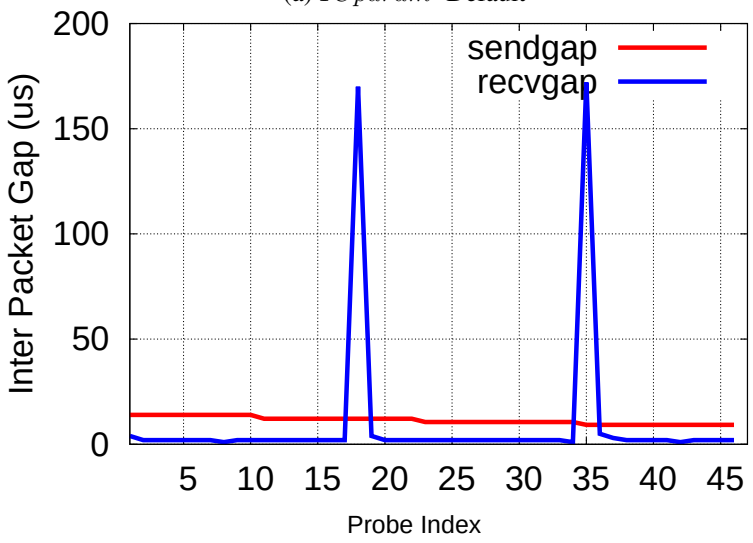


(b) Test with Same/Smoother Traffic

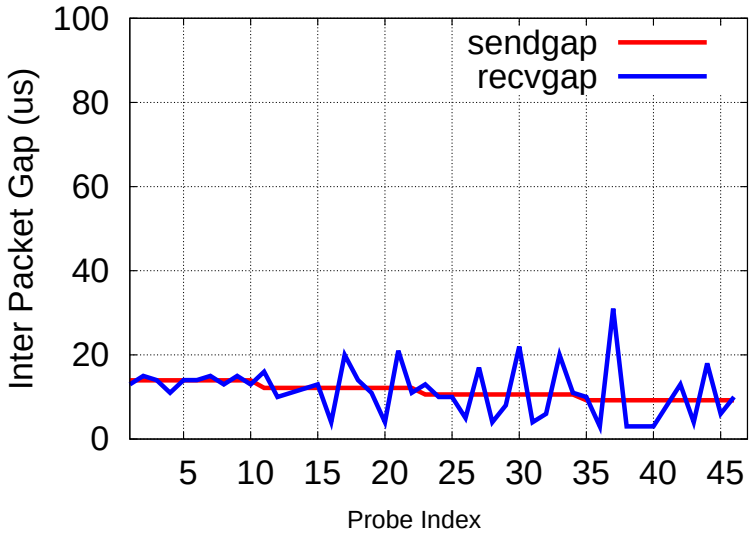
Figure 6.6: Impact of Cross-traffic Burstiness In Training and Testing, Respectively
The 5%-95& relative estimation error is plotted as candle sticks, and 10%-90% relative error is plotted as bars.



(a) $ICparam$=Default



(b) $ICparam$=200μs



(c) $ICparam$=10μs

Figure 6.7: Probe Streams with Different $ICparam$

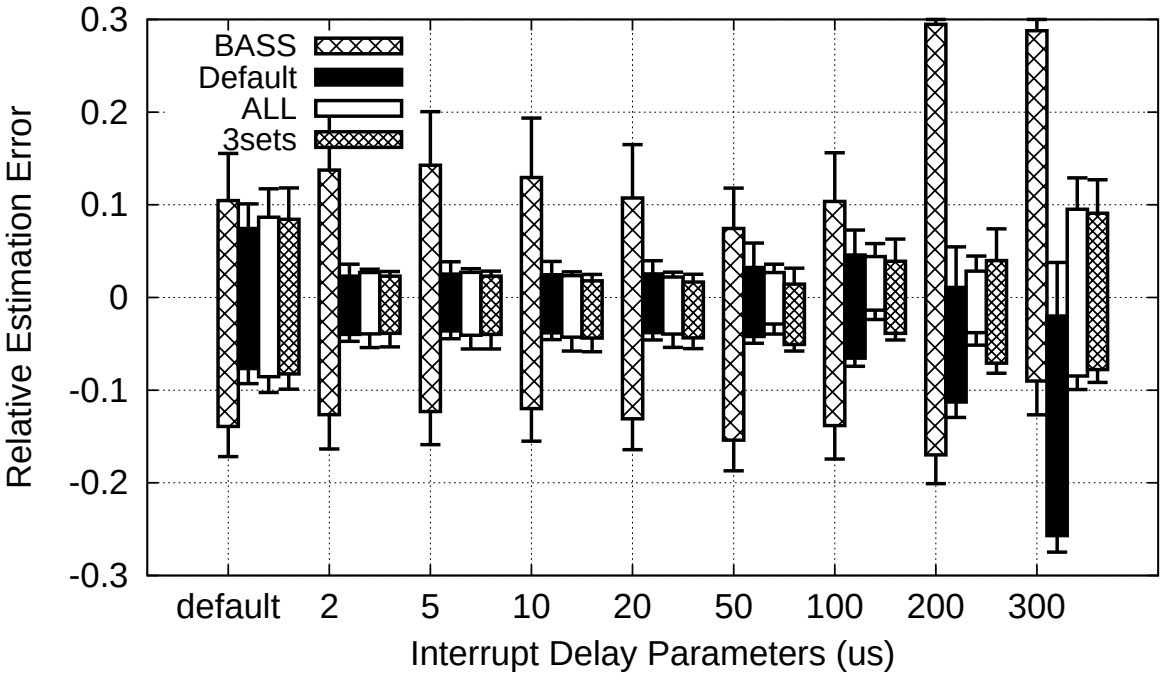


Figure 6.8: Impact of ICparams on NIC1
Train with different sets of ICparams. "3-sets" achieves comparable accuracy with "ALL" set.

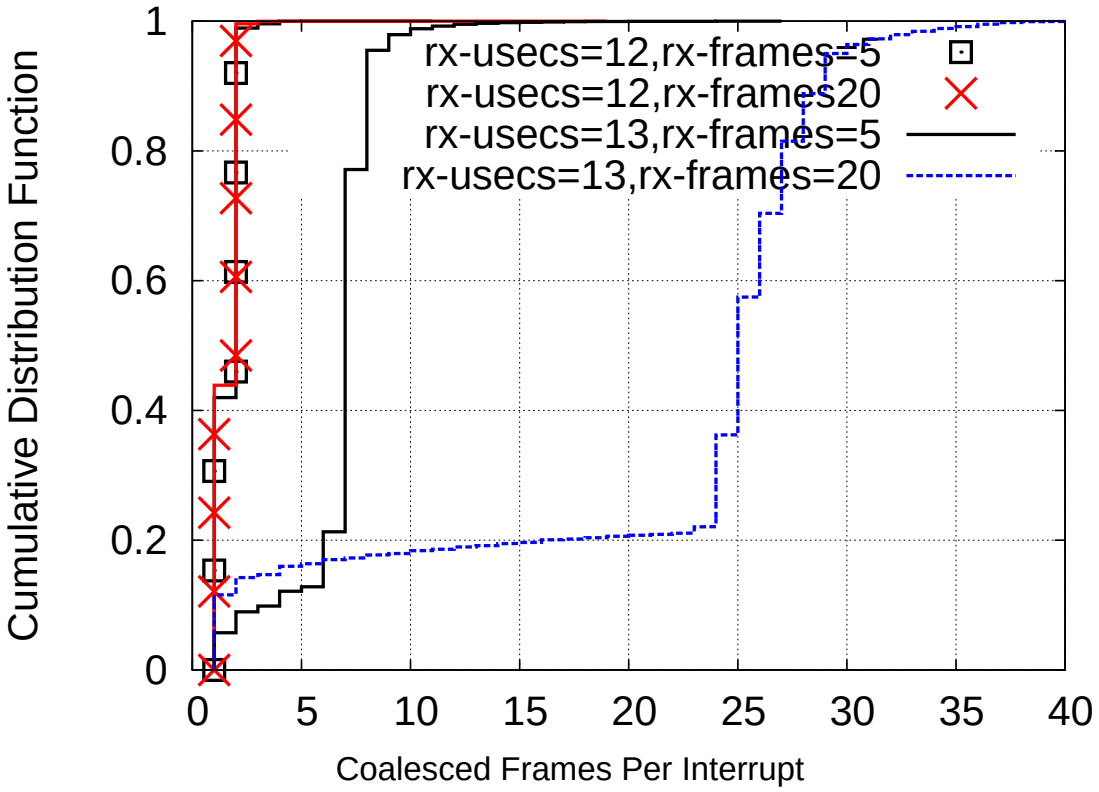


Figure 6.9: Interrupting Behavior on NIC2

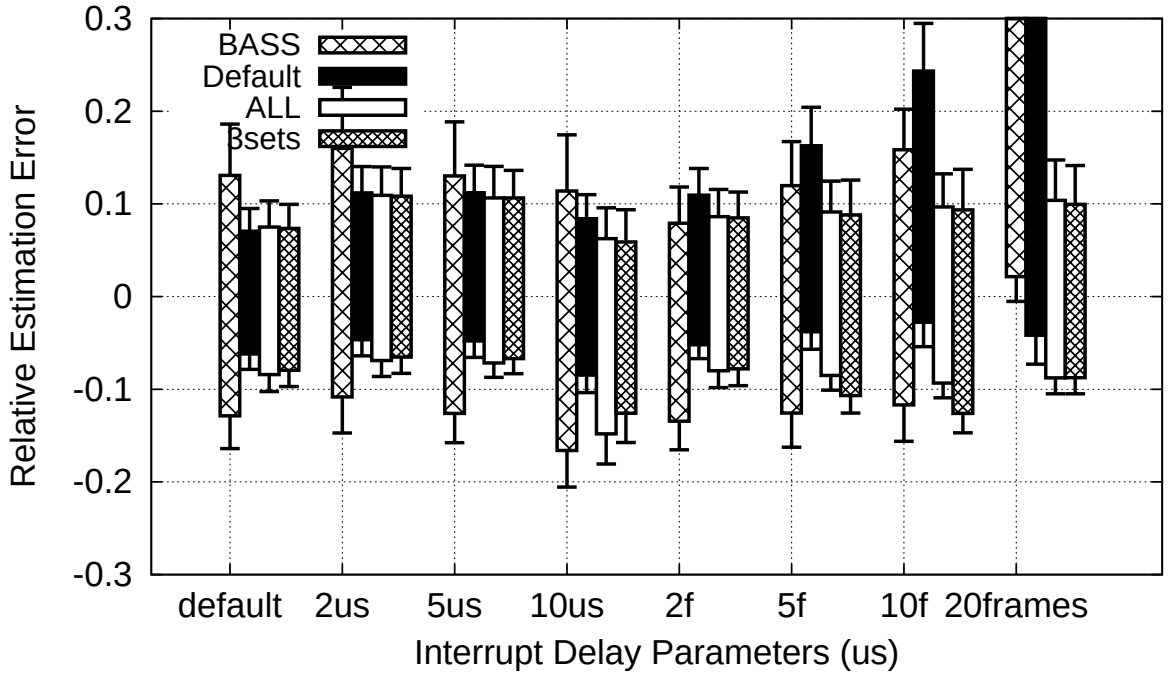
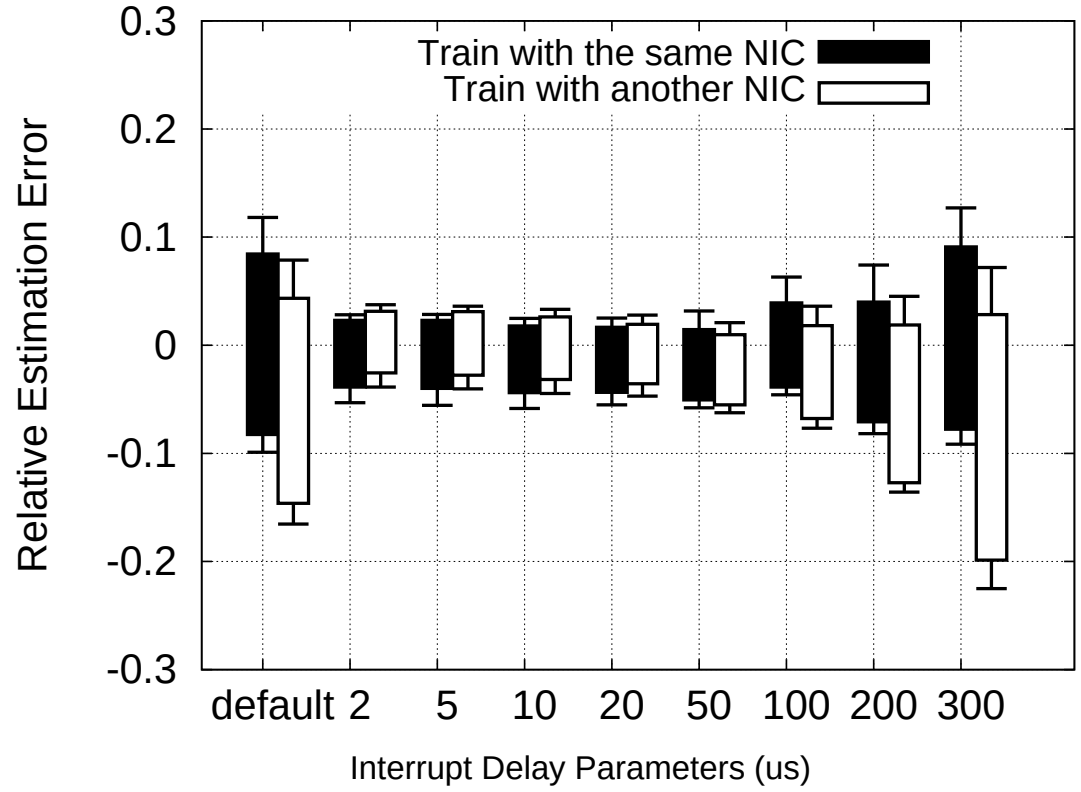
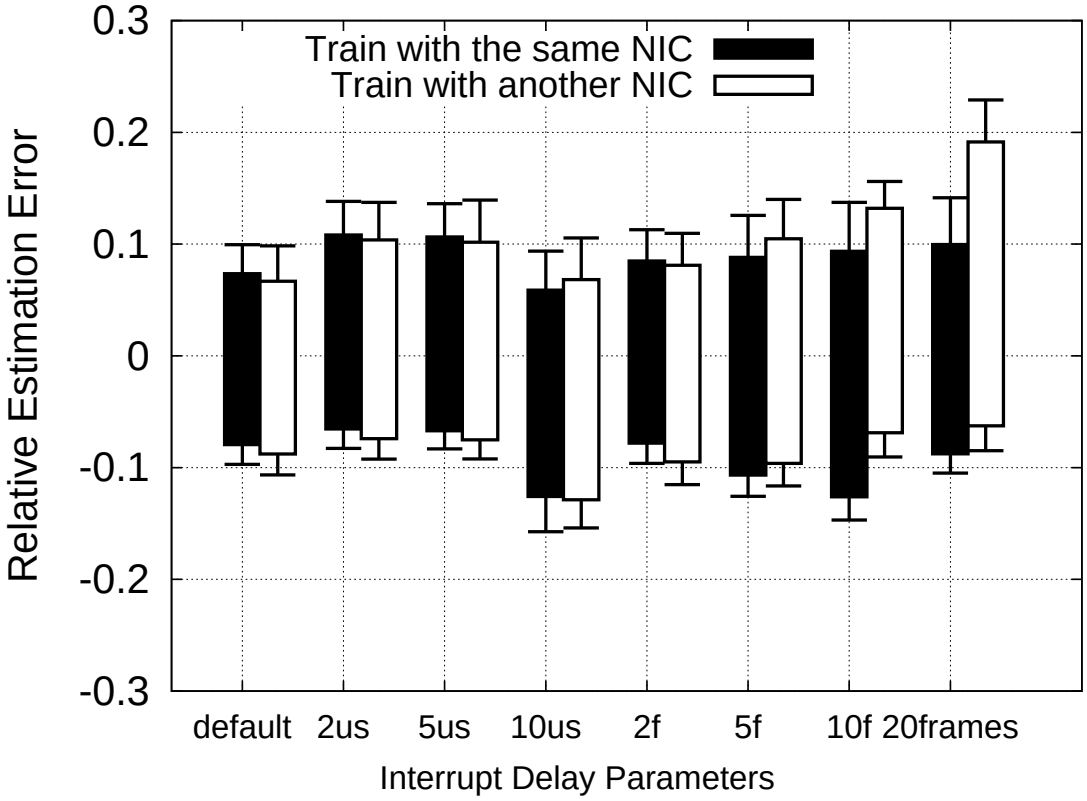


Figure 6.10: Impact of ICparams on NIC2
The same conclusion as on NIC1 is obtained — "3-sets" is able to yield considerable accuracy.



(a) Testing with NIC1



(b) Testing with NIC2

Figure 6.11: Cross-Nic Validation

The models trained on one NIC is used to estimate bandwidth with probe streams received on another NIC. Cross-NIC models are portable, yielding comparable $ABest$ accuracy with those trained on the same NIC.

CHAPTER 7: IMPLEMENTING RAPID IN TCP STACK

From the previous two chapters, we have addressed the challenges for achieving accurate inter-packet gap creation, and robust bandwidth estimation. For each of them we have decided mechanisms that are suitable for RAPID Linux implementation. For wide deployment of the protocol, the implementation should be compatible with the state-of-the-art protocol headers and common protocol stack implementations. Recall that in Section 1.2.4, we point out two aspects of RAPID which are intrinsically incompatible with existing Linux protocol stack: firstly, RAPID packet transmission is "gap-clocked", but Linux provides no interface to realize this; secondly, RAPID requires μs resolution timestamping at the receiver, but Linux provides only milli-second resolution. Besides these two major obstacles, RAPID implementation also face other challenges such as TCP offloading options which can potentially destroy the created gaps, the constraint of no floating point computation in kernel modules and concurrent access of shared variables in multi-core systems.

Goal In this chapter, we address these implementation-wise challenges. We aim to deploy RAPID as pluggable kernel modules with existing Linux kernel interfaces, without modifying the kernel source code.

In Fig. 7.1 we plot the kernel modules in our implementation prototype. Currently, we deploy RAPID as a one-way protocol, with different sets of modules on the sender and receiver side. On each side, it includes a TCP module and two Qdisc modules. In Section 7.1 and Section 7.2, we describe how these modules jointly serve the purpose of "gap-clocking" and fine-grain receiver timestamping. Section 7.3 addresses other minor challenges mentioned above.

7.1 Realizing "gap-clocking"

As described in Section 1.2.4, main-stream congestion control protocols have a notion of "congestion-window", which controls the number of unacknowledged packets allowed at any time. For them, packet-transmission is "ack-clocked": the arrival of acknowledgments triggers transmission of new data packets. However, TCP RAPID keeps no notion of "congestion-window", and it is designed to send continuous

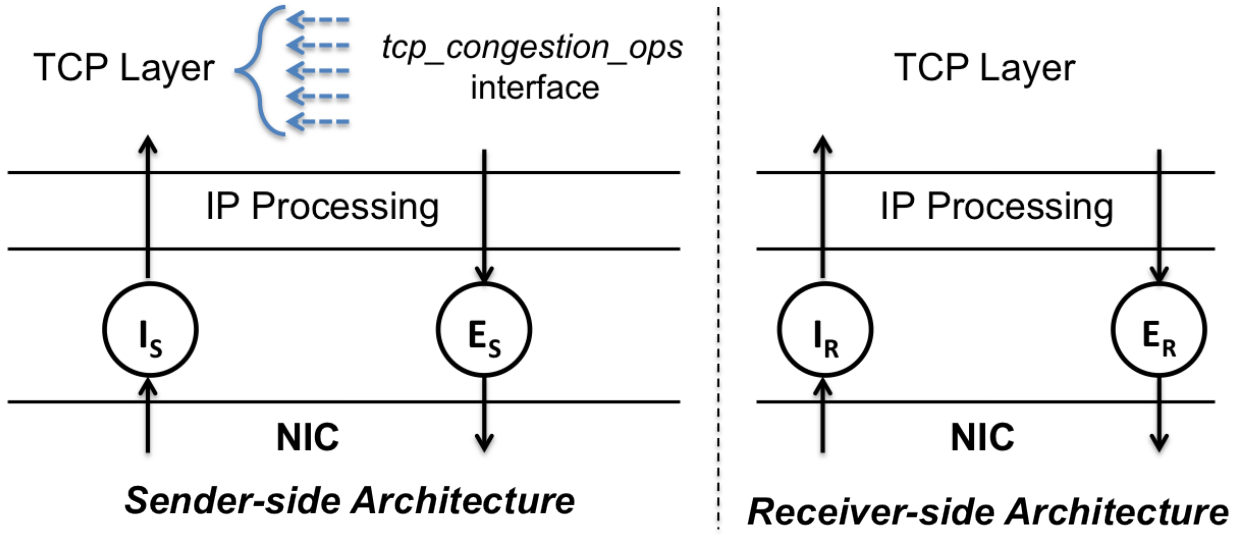


Figure 7.1: Architecture of RAPID Implementation

p-streams asynchronous with ACK arrivals — each packet transmission is "gap-clocked". Next, we describe how we achieve "gap-clocking" with existing Linux kernel interfaces.

7.1 Removing "ack-clocking"

We decouple packet-transmission with ACK arrivals with relative ease by simply setting the *cwnd* to INF (a very large value, we set it to $2 \times BDP$). As a result of doing this, the TCP layer would send segments down for lower layer processing as soon as data is made available by the application. In our implementation, we use the `init` and `cong_avoid` interfaces provided by $tcp\_congestion\_opts$ for doing this.

7.1 Incorporating "gap-clocking"

The packet scheduler E_S illustrated in Fig. 4.4 is deployed in our RAPID implementation to realize "gap-clocking". The packet scheduler is implemented using Linux Qdisc interface, handing packets from the protocol stack to the NIC driver. It maintains a queue for each TCP flow using a given NIC interface, and a scheduling heap facilitating gap maintenance across multiple flows. E_S responses to $dequeue()$ and $enqueue()$ system calls. As described in Section 4.5, the *Dummy-packet* approach is adopted to enforce precise creation of inter-packet gaps.

For each active flow using a given NIC interface, the Qdisc fulfills the following tasks:

- Upon each $enqueue()$ call, the scheduler receives a packet from TCP/IP stack and places it in the internal queue for this specific flow.

- Once a per-flow queue has enough packets in it, E_S groups them into units of p-streams. It computes per-packet gaps based on the average sending rate of each p-stream (which is obtained from the TCP module), and assigns the scheduled transmission time t_s for each packet;

- For each packet, it records its sequence number, t_s and the index in its p-stream in a circuar table T_S shared with the TCP layer — the TCP congestion control module will extract per-p-stream information from the table for bandwidth estimating.

- Each entry in the scheduling heap is the packet at the head of each per-flow queue. E_S. Thus, the heap-top packet has the nearest dequeue time across all flows. Upon each $dequeue()$, E_S checks whether the transmission of the heap top is due. If true, E_S removes the packet from its per-flow queue, transmits it to the NIC, and update the scheduling heap with the current queue head. Otherwise, E_S inserts a proper-sized PAUSE frame to occupy gap-time.

7.2 Fine-scale Arrival Timestamping

As stated in Section 1.2.4, RAPID relies on TCP timestamps to communicate packet arrival time at the receiver back to the sender. However, widely-deployed protocol stacks only produce and recognize timestamps with milli-second accuracy, which fails to satisfy the accuracy requirement on ultra-high speed networks.

In order to record inter-packet receive gaps with μs precision and accuracy, we create two Qdiscs E_R and I_R at the receiver, and one I_S ingress Qdisc at the sender. We plot the four Qdiscs including E_S in the RAPID implementation in Fig. 7.1. I_R receives packets as they are delivered by the NIC to the kernel and uses $ktime\_to\_ns$ to timestamp packet arrivals with μs precision. We implement a circular table to store the sequence number and the arrival timestamp for each incoming packet. This table data structure is shared with I_R and E_R. Once an ACK is generated and sent by the protocol stack to E_R, it looks up the table for the arrival time of the corresponding packet that triggered this ACK, and substitutes it for the TCP timestamp value in header timestamp(TSval) field—TCP checksum is recomputed and updated accordingly.

The sender side also maintains a circular table to store the sequence number of each packet, as well as its index in a p-stream. The table is shared among I_S, E_S and the tcp module. When the ACK segment reaches the sender, the ingress Qdisc I_S extracts the μs timestamp contained in the TSval field for each incoming ACK, finds the corresponding packet in the circular buffer this ACK is acknowledged for, and record extracted timestamp in the table in order to be accessed by the TCP layer and E_S. To ensure correct TCP processing (which expects monotonically increasing millisecond timestamps), I_S restores TSval field in the ACK header with the local *jiffies* timestamp before handing the packet for TCP-layer processing. The local millisecond timestamp is aligned with the offset between the sender and receiver clocks obtained from SYN/ACK packet to approximate the corresponding *jiffies* clock on the receiver. Once the TCP layer receives an ACK of a packet, it finds its corresponding entry in the circular table, and checks whether the acknowledged packet is the last packet in a p-stream. If it is, the TCP module computes (g_s, g_r) for the whole p-stream and then conduct bandwidth estimation.

7.3 Additional challenges in the Linux kernel implementation

TCP RAPID adopts the cross-layer design — both the sender and receiver implementation consists of a congestion-control module using *tcp_congestion_ops* interface in TCP layer, and two link layer Qdisc modules. In this section we describe three additional challenges in designing an implementation of TCP Rapid as dynamically loadable kernel modules. These challenges are independent of the challenges in realizing "gap-clocking" and microsecond timestamping because they deal with conditions specific to implementing functions used by TCP Rapid inside the kernel [90].

Fast Per-frame Execution

One goal of the implementation design was to keep CPU utilization as close as possible to that of widely used TCP variants such as Cubic. Some additional utilization is unavoidable in implementing functions such as BASS de-noising and computing available bandwidth. However, these functions are executed on a per-probe-stream basis and their CPU utilizations are amortized over a number of TCP segments (64 or more). It is also important to control CPU utilization required for processing each segment. The data structures and algorithms that need to run for each segment sent or received were carefully chosen to minimize per-frame execution path lengths. Some examples are: the heap used for scheduling frame transmissions, the per-TCP-connection send queues, and the circular table holding p-stream frame times.

No Floating-point Hardware

Using the system floating-point hardware is permitted in kernel code, but its use is strongly discouraged and is very rare, confined to special heavy-computation cases such as checksums and cryptography. The reason is that the kernel does not save floating-point context when entered by system calls from user space where floating point may be in use. If a kernel module were to use the floating-point hardware, it would have to save and restore the registers and other state. The overhead of handling such a large amount of context usually negates any gain from using the hardware unless there is a large amount of computation to be done. While TCP Rapid could conveniently use floating-point in some cases, the overheads make its use undesirable.

Instead TCP Rapid uses several strategies to handle these computations. In many cases it is more efficient to use 64-bit integers for precision (e.g., nanosecond counters) and integer operations with appropriate scaling of the operands. In other cases, single-variable functions can be pre-computed and the results stored in an array with linear interpolation using integers applied to find values between two stored results. Another approach is to approximate some non-integer values as a ratio of two integers that can be easily computed with arithmetic shifts (e.g., 256). For example, 1000/1.15 (869.565) is adequately approximated for use in Rapid by (1000 * 223) $>>$ 8 (871).

Implementation for Multi-core

Obviously implementing kernel functions on multicore systems requires careful attention to protecting critical sections in the code from concurrent execution on independent CPUs. In the TCP Rapid implementation, this issue is made more critical because of the data structures shared between the TCP module and the Qdisc module. As mentioned in Section 7.1, the tcp module and the Qdisc modules share a circular table which records for each packet its index in the p-stream, intended dequeue time, and the time it is acknowledged. The table is accessible from the socket private data structure *inet_ca_priv*. The egress Qdisc module adds entries to the table upon packet enqueuing; the ingress Qdisc module searches in the table for the packet corresponding to the latest ACK, and updates its acknowledgement time; the congestion control module checks whether an entire p-stream is acknowledged, and computes inter-packet arrival gaps for bandwidth estimation for that p-stream. On multicore systems, these modules can be executed concurrently, and either may be executed concurrently on a code path initiated by a system call from user space or an interrupt in kernel context.

To ensure that at any moment there is only one module accessing the circular table, the critical sections in the shared data structures are protected by kernel read/write spin-locks that also disable interrupt "bottom half" (softirq and tasklet) processing (e.g., write_lock_bh()). Obtaining the lock is required to access *inet_ca_priv* from any module. Twenty-one lock/unlock pairs are used to synchronize access to shared data. Primitive data types (e.g., integers) are protected by declaring them as "atomic" types and using kernel guaranteed atomic operations (e.g., atomic_set()).

TCP Offloading

TCP offloading is the technique to offload some functionalities in TCP/IP stack processing to the NIC. The most important offloading functions are TCP segmentation offloadint (TSO) and large receiving offloading (LRO). For outbound traffic, TSO allows the CPU to send large chunks of data to the NIC, which will then divide packets into MTU size and encapsulate them. For inbound traffic, LRO merges multiple packets from a specific stream into a larger packet to reduce the number of NIC interrupt. Such offloading is widely used with gigabit NIC interfaces in order to free up CPU cycles from high-rate packet processing. But it deprives the operation system the luxury to manipulate in per-packet granularity, which is necessary for gap-creation in RAPID implementation for both sender and receiver side. On the sender side, the egress scheduler controls each individual packet transmission before they are handed to the NIC, and the ingress scheduler has to replace the timestamps for each ACK packet. Similarly on the receiver side, the ingress scheduler requires to precisely record arrival time for each ACK packet after they are handed from the NIC to the operation system, and the egress scheduler writes μs-resolution timestamps in the header of each returning ACK. As a result, TSO and LRO are turned off on both the sender and receiver side to make RAPID function properly. To observe the effects of turning off offloading, we run an *iperf* flow transmitting packets at 10Gbps, and monitor the CPU utilization on both the sender and the receiver. Without TCP offloading, the CPU utilization increases from 21.6% to 30.6% at the sender, and the from 34.2% to 45.2% at the receiver.

7.4 Line of Count

We list the line count for critical sections in our implementation code in Table 7.1.

Table 7.1: Cound of Line For RAPID Implementation

Code Functionality	Sender	Receiver
Header Files, where data structures used in the modules are declared	126 for tcp module 169 for Qdisc modules	72 for tcp module 94 for Qdisc modules
Data Structure and Variable Initialization	87 for tcp module 197 for Qdisc modules	33 for tcp module 87 for Qdisc modules
Probe Stream Generation and Gap Updates	254	-
Probe Stream Scheduling	206	-
Scheduling Heap Library	131	-
Bandwidth Estimation and Rate Adaptation	630	-
μs Timestamping	-	209
ms Timestamp Substitution	169	-

7.5 Summary

This chapter describes how we implement TCP RAPID as kernel modules, solely with Linux kernel interfaces. Specially, we decouple packet transmission from ACK arrivals by setting congestion window to a large enough value, and schedule packets according to inter-packet gaps with an egress Qdisc at the sender. We implement a set of Qdiscs on both the sender and receiver sides to timestamp packet arrival with microsecond accuracy, as well as communicate the timestamps with counterparts. We also addresses the challenges of kernel floating point calculation, multi-core processing and TCP offloading.

CHAPTER 8: CLOSE-LOOP EVALUATION OF TCP RAPID

In previous chapters, we focused on the two modules in Fig. 2.1—the p-stream generator module and the bandwidth estimator module. Specifically, we (i) implemented a set of Qdiscs to timestamp packet arrivals with μs precision; (ii) developed and evaluated the gap-packet mechanism to create precise inter-packet gaps ; and (iii) developed the BASS denoising mechanism to improve bandwidth estimation accuracy for multi-rate p-streams, In this chapter, we feed the estimated avail-bw to the rate-adapter module and conduct close-loop evaluation of RAPID congestion control. The ***goals*** of this chapter are as follows:

- We focus on the rate-adapter module, which addresses the stability/adaptability trade-off discussed in Section 1.2.3. Specially, we decouple the bandwidth-probing timescale from the adaptation timescale by introducing a control parameter γ (Section 8.1).

- We conduct extensive experimental evaluation to study the impact of the rate-adapting parameters τ and η on the performance of RAPID.

- We experimentally study whether our implementation is able to achieve the performance promised in simulation-based evaluations reported in [91]. We test how the implementation scales to high-speed throughput, how quickly it adapts to changes in avail-bw, and whether it interferes with low-speed TCP traffic.

- In our ultra-high-speed 10 Gbps testbed, we compare our implementation of Linux implementations of several protocols: New Reno, Bic, Cubic, Scalable, Highspeed, Hybla, Illinois, Vegas, Westwood, LP, Yeah, Fast, and Compound.[1]

- Finally, we demonstrate how important each of the mechanisms developed in this dissertation is to addressing the practical challenges faced by RAPID. We remove the four mechanisms one by one: first the μs timestamping with Qdiscs, then the dummy-packet method of gap creation, then the BASS

[1]The implementation of Fast is not publicly available. We implement it in Linux based on its Linux-emulating pluggable NS2 simulation code. With default parameters, it performs poorly in our testbed.

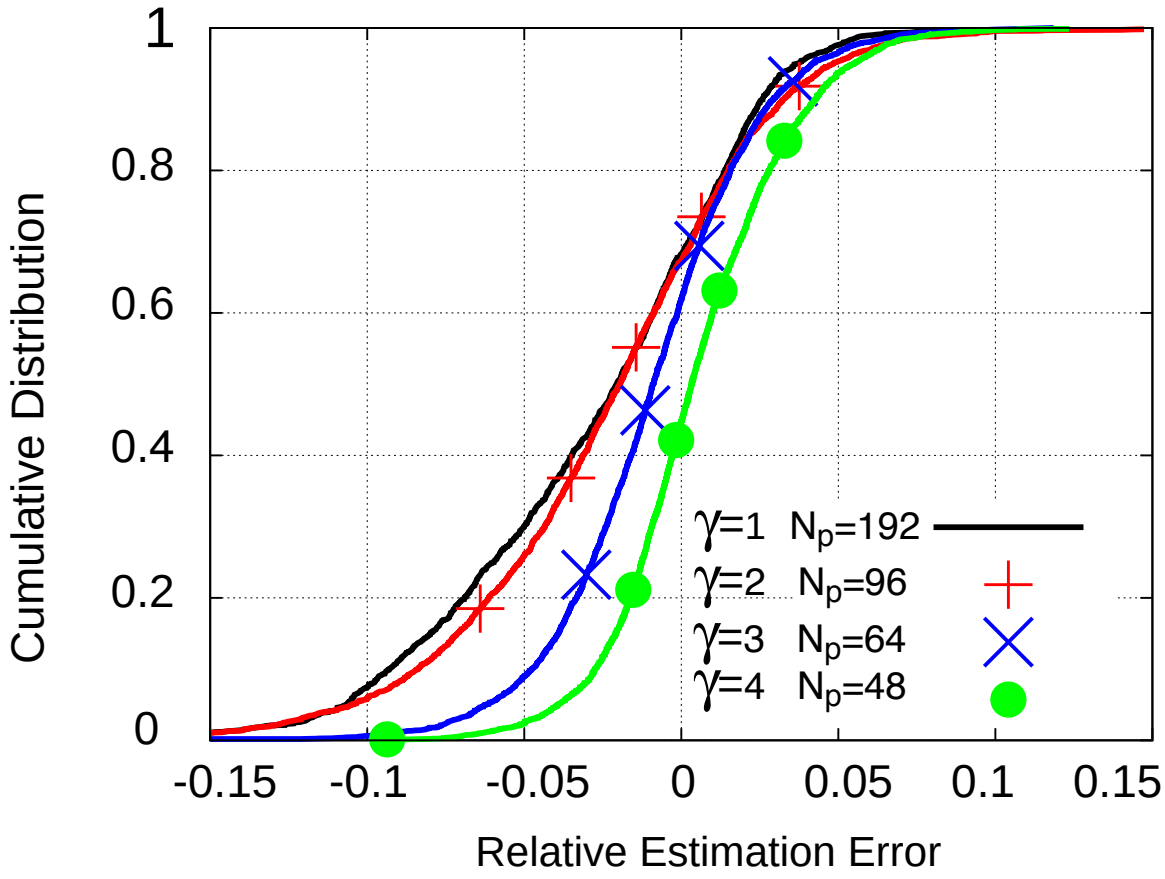


Figure 8.1: Decoupling Probing/Adapting

denoising mechanism, and finally the decoupling of the probing timescale from the adapting timescale. We show how the performance of RAPID degrades without each of them.

8.1 Addressing the Stability/Adaptability Trade-off

RAPID faces the classical dilemma of of bandwidth tracking and adapting: the stability/adaptability trade-off discussed in Section 1.2.3. RAPID needs to track the variation in avail-bw as agilely as possible, but it adapts its sending rate only when the avail-bw becomes stable. The design of RAPID means that it cannot simultaneously satisfy both requirements; the frequency at which it probes bandwidth and adapts its rate is controlled by a single parameter N, the p-stream length ($N = N_r \times N_p$). Using a small N improves the agility of bandwidth tracking because it samples the avail-bw more frequently, but a small N also has a short probing timescale, thus impairing the stability of the estimated avail-bw. Using a larger N reduces sampling rates, but also produces a more stable avail-bw signature since the avail-bw is estimated at larger timescales.

To reduce the impact of this trade-off, we propose to decouple the probing and adapting timescales from one another. Rather than requiring the sender to update R_{avg} upon each $ABest$ computation, we propose to update R_{avg} less frequently—every γ p-streams. Specifically, we compute the mean of avail-bw estimates made by every γ successive p-streams, and we use this mean to adapt the R_{avg} of the transfer only once every γ p-streams.

This decoupling naturally leads to a question: if we fix the rate-adapting timescale ($N \times \gamma$), are bandwidth estimates more accurate when we use longer p-streams (large N, $\gamma = 1$) or when we use the mean bandwidth estimate of several smaller p-streams (small N, $\gamma > 1$)? To study this, we fix the rate-adapting timescale at a heuristic value of 192 packets and vary the probing timescale at $48, 64, 96$, and 192 packets per p-stream ($\gamma = 4, 3, 2, 1$, respectively). Specifically, we generate p-streams of four different lengths: $N = 48, 64, 96, 192$. These p-streams share the 10 Gbps bottleneck link with BCT cross traffic. Each of these p-streams probe at four rates, with a probing range of 50%. These p-streams have controlled average sending rates that range from 1 Gbps to 9 Gbps, with 10 Mbps steps between consecutive p-streams. To emulate real-world situations, we use a delayed acknowledgment mechanism. We compute the mean of the bandwidth estimates yielded by every γ successive p-streams and analyze the actual avail-bw B_g from the trace taken after the bottleneck link at the 192-packet timescale. The relative bandwidth estimation error at this rate-adapting timescale is plotted in Fig. 8.1, where each curve represents a different probing timescale. We find that a larger γ value (shorter p-streams) reduces estimation errors from 15% to 5%. When p-streams are shorter than 64 packets, however, the estimation errors do not reduce with larger γ. In the remaining evaluations, we use this heuristic of $N = 64$ with the rate-adapting timescale of 192 packets ($\gamma = 3$).

8.2 Sustained Error-based Losses

On high-speed paths, congestion can cause packet loss—routers and switches may drop packets when their internal buffers overflow in the presence of exceedingly high incoming traffic volume. Even when the path is congestion-free, packet loss can be induced by bit errors. In bit-error packet losses, ideally an end-to-end congestion control protocol would re-acquire a sending rate equal to the spare bandwidth as quickly as possible after loss recovery.

Our first set of experiments evaluates how RAPID and other TCP protocols adapt to error-based packet losses. We generate a single TCP flow using the iperf program; there is no cross traffic on the 10 Gbps path.

We vary the loss rate from 10^{-2} (very high) to 10^{-8} (very low). The round-trip time of the TCP flows is fixed at 30 ms, the median round trip time (RTT) across the continental US. Each RAPID flow lasts for 5 minutes.

Table 8.1: RAPID with Sustained Error-based Losses

(TAU,ETA)	no loss	10^{-8}	10^{-7}	10^{-6}	10^{-5}	10^{-4}	10^{-3}	10^{-2}
5ms, 1/10	9.45	9.45	9.44	9.41	9.24	6.43	1.32	0.01
5ms, 1/6	9.45	9.45	9.45	9.40	9.21	6.35	1.31	0.01
5ms, 1/4	9.45	9.45	9.44	9.41	9.20	6.35	1.29	0.01
5ms, 1/3	9.45	9.44	9.44	9.42	9.19	6.37	1.30	0.01
5ms, 1/2	9.44	9.44	9.44	9.41	9.19	6.39	1.31	0.01
5ms, 1	9.45	9.44	9.44	9.39	9.20	6.37	1.32	0.01
10ms, 1/10	9.45	9.45	9.44	9.39	9.13	6.21	1.31	0.01
10ms, 1/6	9.45	9.44	9.44	9.40	9.10	6.19	1.32	0.01
10ms, 1/4	9.44	9.44	9.44	9.40	9.09	6.19	1.33	0.01
10ms, 1/3	9.45	9.44	9.44	9.41	9.07	6.17	1.34	0.01
10ms, 1/2	9.44	9.44	9.42	9.37	9.07	6.18	1.31	0.01
10ms, 1	9.45	9.44	9.39	9.33	9.04	6.09	1.30	0.01
20ms, 1/10	9.45	9.44	9.44	9.40	9.09	6.19	1.32	0.01
20ms, 1/6	9.45	9.44	9.44	9.39	9.06	6.17	1.33	0.01
20ms, 1/4	9.44	9.44	9.42	9.35	9.04	6.12	1.30	0.01
20ms, 1/3	9.44	9.43	9.43	9.34	9.04	6.07	1.31	0.01
20ms, 1/2	9.44	9.41	9.37	9.32	8.99	6.01	1.29	0.01
20ms, 1	9.43	9.43	9.34	9.23	8.83	5.92	1.29	0.01
30ms, 1/10	9.44	9.42	9.42	9.39	9.08	6.13	1.30	0.01
30ms, 1/6	9.43	9.42	9.41	9.35	8.93	6.08	1.30	0.01
30ms, 1/4	9.42	9.41	9.38	9.33	8.95	5.99	1.30	0.01
30ms, 1/3	9.41	9.30	9.29	9.24	8.91	5.89	1.29	0.01
30ms, 1/2	9.37	9.29	9.21	9.21	8.87	5.81	1.29	0.01
30ms, 1	9.32	9.25	9.17	9.09	8.73	5.69	1.30	0.01
50ms, 1/10	9.40	9.36	9.31	9.23	9.05	4.83	1.31	0.01
50ms, 1/6	9.35	9.33	9.25	9.20	8.88	4.76	1.28	0.01
50ms, 1/4	9.33	9.30	9.24	9.20	8.76	4.51	1.29	0.01
50ms, 1/3	9.32	9.30	9.24	9.17	8.73	4.38	1.28	0.01
50ms, 1/2	9.32	9.31	9.23	9.17	8.73	4.29	1.27	0.01
50ms, 1	9.32	9.29	9.19	9.16	8.66	4.07	1.28	0.01

8.2 Impact of τ and η

We first control τ and η to determine how the aggressiveness of the RAPID rate-adapter affects link utilization in the presence of random packet losses. Table 8.1 lists the throughput with different parameter settings. We find that, for all settings, RAPID estimates avail-bw at 9.6 to 10 Gbps for more than 95% of p-streams. At loss rates lower than 10^{-7}, τ parameters make little difference. At loss rates higher than 10^{-6}, however, a larger τ yields slightly lower throughput; the larger τ slows down the rate at which RAPID

Table 8.2: Non-RAPID Protocols with Sustained Error-based Losses

Throughput (Gbps)	no loss	-8	-7	-6	-5	-4	-3	-2
RAPID (τ=10ms,η=$\frac{1}{3}$)	9.45	9.44	9.44	9.41	9.07	6.17	1.34	0.01
RAPID (τ=50ms,η=$\frac{1}{4}$)	9.33	9.30	9.24	9.20	8.76	4.51	1.29	0.01
Reno	9.85	6.13	4.97	2.54	0.923	0.262	0.076	0.031
Cubic	9.85	9.29	8.68	4.44	1.20	0.243	0.067	0.026
Scalable	9.85	9.61	9.61	9.54	8.85	1.800	0.213	0.054
BIC	9.84	9.76	9.62	9.30	5.36	0.919	0.144	0.039
Fast	9.82	9.10	6.71	6.25	5.32	3.72	1.32	0.174
HighSpeed	9.54	9.54	9.48	8.53	2.43	0.451	0.107	0.035
HTCP	9.61	9.35	9.15	6.76	2.29	0.401	0.979	0.032
Hybla	9.53	9.23	8.02	6.66	1.96	0.411	0.109	0.032
Illinois	9.63	9.62	9.57	9.32	5.67	1.87	0.622	0.165
LP	9.34	7.56	4.97	2.44	0.76	0.256	0.071	0.027
Yeah	9.83	9.83	9.82	9.71	9.03	1.72	0.223	0.074
Westwood	9.73	3.59	3.56	3.00	1.13	0.690	0.252	0.053
Veno	9.80	4.99	4.21	2.32	0.90	0.399	0.119	0.039
Vegas	9.01	5.80	4.89	3.91	3.21	0.966	0.113	0.034
Compound	9.16	9.03	6.92	2.27	0.826	0.227	0.109	0.032

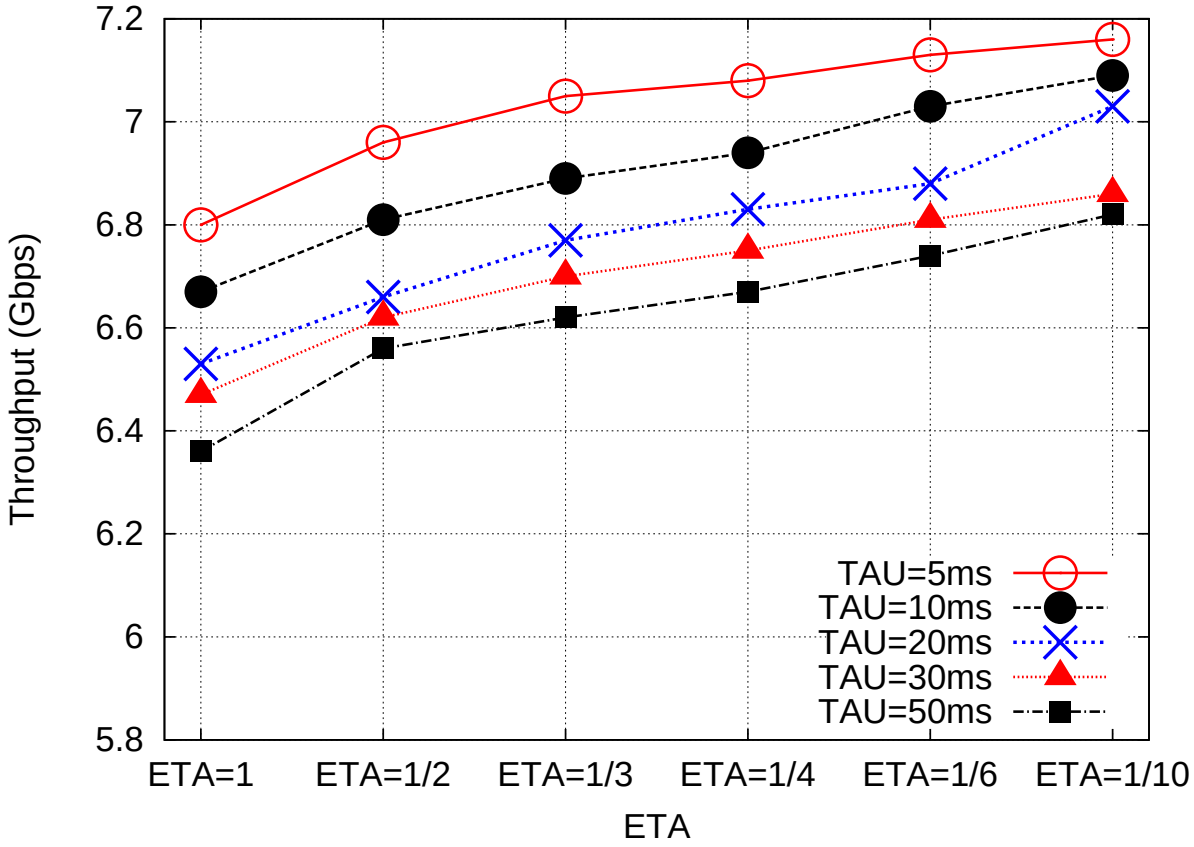
increases to R_{avg}. At loss rates higher than 10^{-4}, all of the parameters yield comparably low throughput. RAPID reduces its sending rate by 25% after each recovery from a packet loss, and when packet loss is frequent, the impact of this rate reduction outweighs the impact of rate-adapting aggressiveness. For all parameter settings, RAPID performs poorly with loss rates of 10^{-2} or more; losses in nearly every other p-stream prevent it from estimating avail-bw at all.

8.2 RAPID with TCP Variants

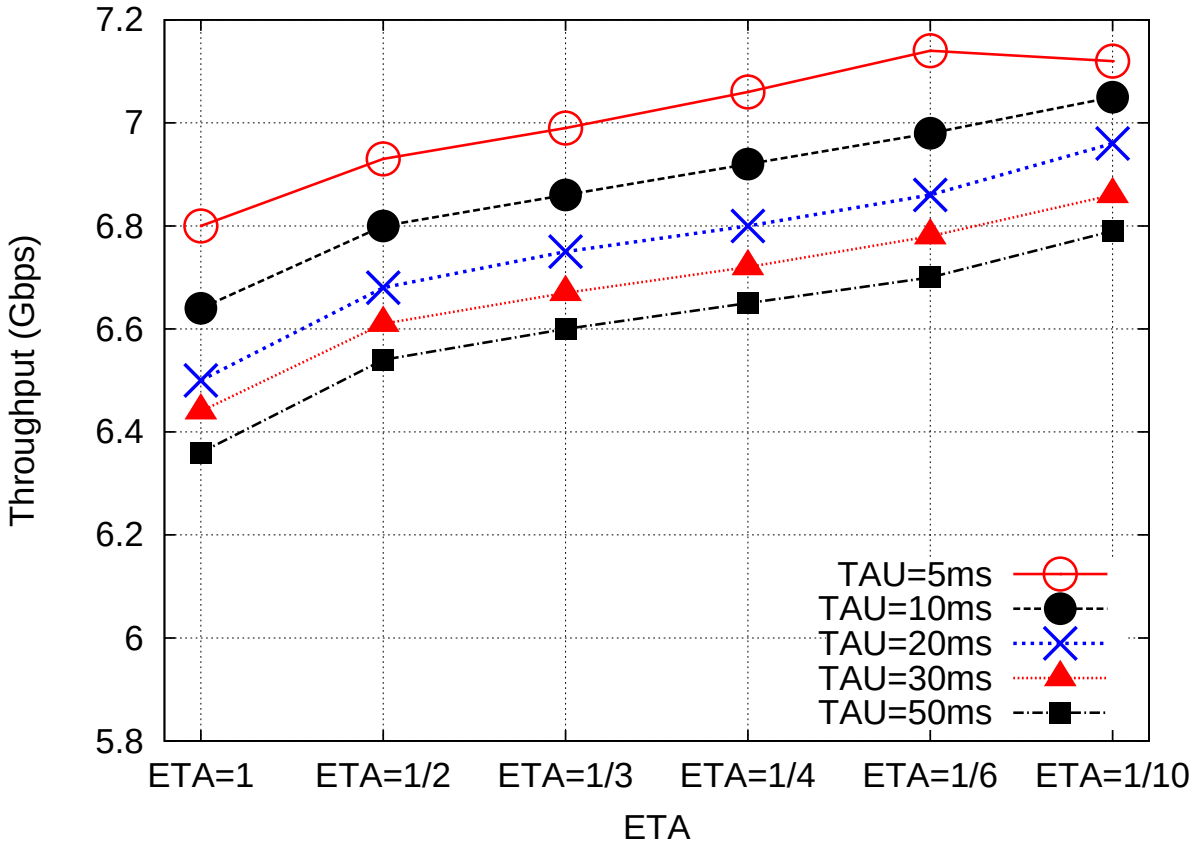
We next run an iperf TCP flow on the 10 Gbps path using other underlying transport protocols. We list their achieved steady-state throughput in Table 8.2; these values are compared with the steady-state throughput achieved by RAPID in Table 8.1. All protocols show significant throughput losses when error rates exceed 10^{-6}. However, at high loss rates, RAPID consistently yields more than double the throughput of other protocols. We conclude that *in the presence of error-based losses, RAPID scales better than the main-stream transport protocols*.

8.3 Adaptability to Bursty Traffic

In this section we evaluate how well the RAPID implementation adapts to non-responsive cross-traffic with a dynamically changing load. We generate and experiment with cross-traffic of three different levels

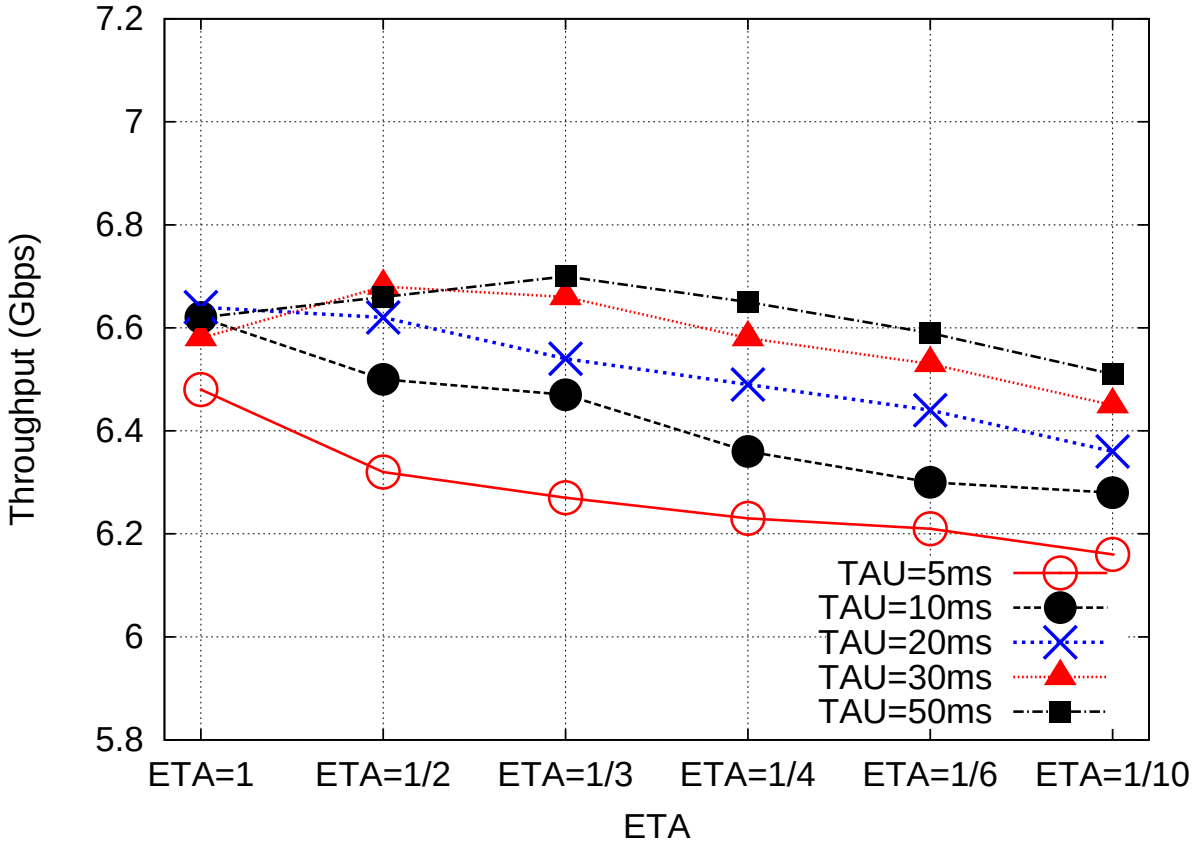


(a) RTT=5ms

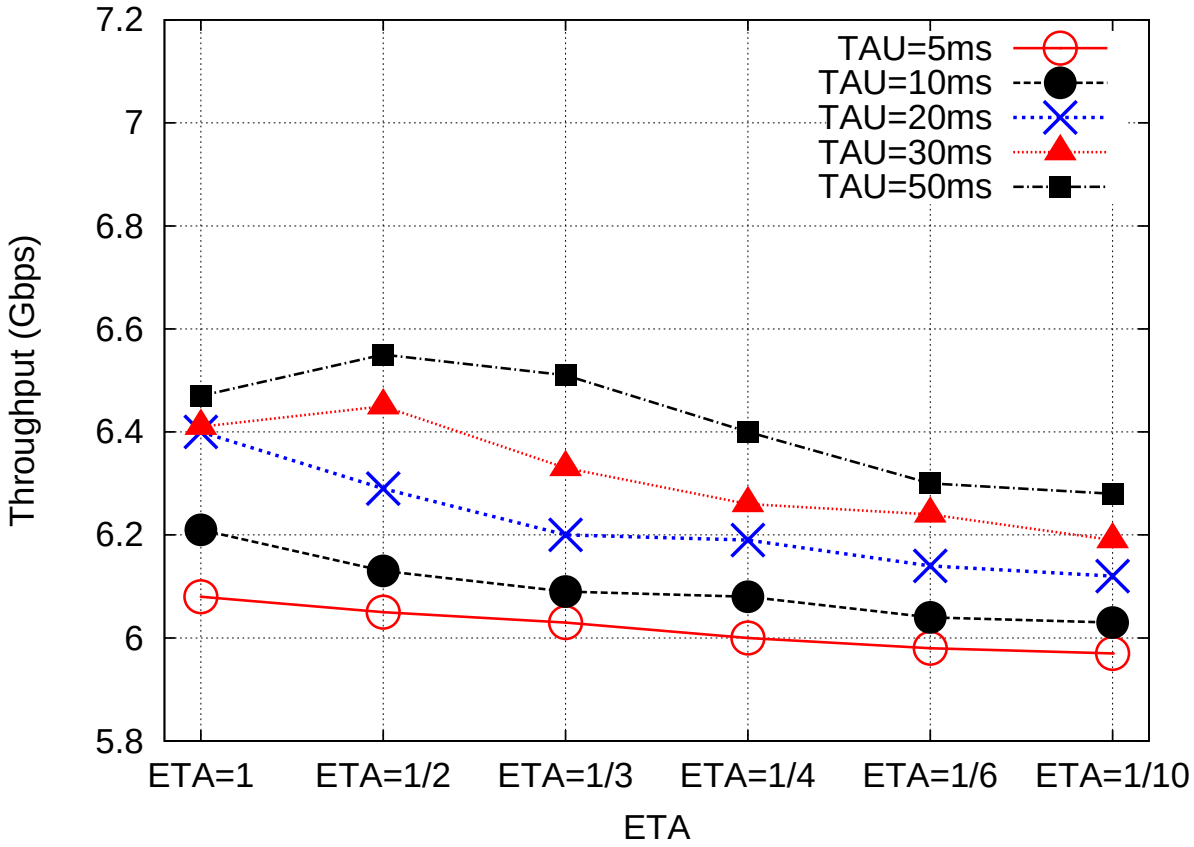


(b) RTT=30ms

Figure 8.2: Impact of Rate-Adapting Parameters with CBR



(a) RTT=5ms



(b) RTT=30ms

Figure 8.3: Impact of Rate-Adapting Parameters with BCT

Table 8.3: Throughput and Loss Ratio with CBR Cross-Traffic(τ,η)

Throughput Loss Rate	η=1	1/2	1/3	1/4	1/6	1/10
RTT=5ms						
τ=5ms	6.80, 0.000	6.96, 0.000	7.05, 0.000	7.08, 0.000	7.13, 0.002	7.16, 0.003
10ms	6.67, 0.000	6.81, 0.000	6.89, 0.000	6.94, 0.000	7.03, 0.000	7.09, 0.001
20ms	6.53, 0.000	6.66 0.000	6.77, 0.000	6.83, 0.000	6.88, 0.000	7.03, 0.000
30ms	6.47, 0.000	6.62, 0.000	6.69, 0.000	6.75, 0.000	6.81, 0.000	6.86, 0.000
50ms	6.36, 0.000	6.56, 0.000	6.62, 0.000	6.67, 0.000	6.74, 0.000	6.82, 0.000
RTT=30ms						
TAU=5ms	6.80, 0.000	6.93, 0.000	6.99, 0.000	7.06, 0.000	7.14, 0.000	7.12, 0.000
10ms	6.64, 0.000	6.80, 0.000	6.86, 0.000	6.92, 0.000	6.98, 0.000	7.05, 0.000
20ms	6.50, 0.000	6.68, 0.000	6.75, 0.000	6.80, 0.000	6.86, 0.000	6.96, 0.000
30ms	6.44, 0.000	6.61, 0.000	6.67, 0.000	6.72, 0.000	6.78, 0.000	6.86, 0.000
50ms	6.36, 0.000	6.54, 0.000	6.60, 0.000	6.65, 0.000	6.70, 0.000	6.79, 0.000

Table 8.4: Throughput and Loss Ratio with BCT Cross Traffic(τ,η)

Throughput Loss Rate	η=1	1/2	1/3	1/4	1/6	1/10
RTT=5ms						
τ=5ms	6.45, 0.046	6.32, 0.089	6.27, 0.115	6.22, 0.137	6.21, 0.162	6.16, 0.181
10ms	6.62, 0.019	6.50, 0.045	6.47, 0.051	6.36, 0.076	6.30, 0.093	6.28, 0.110
20ms	6.64, 0.007	6.62, 0.019	6.54, 0.027	6.49, 0.036	6.44, 0.049	6.36, 0.060
30ms	6.58, 0.005	6.68, 0.010	6.66, 0.015	6.58, 0.023	6.53, 0.027	6.45, 0.036
50ms	6.62, 0.001	6.66, 0.003	6.70, 0.005	6.65, 0.009	6.59, 0.010	6.51, 0.019
RTT=30ms						
TAU=5ms	6.08, 0.042	6.05, 0.063	6.03, 0.095	6.00, 0.118	5.98, 0.145	5.97, 0.175
10ms	6.21, 0.022	6.13, 0.039	6.09, 0.060	6.08, 0.065	6.04, 0.078	6.03, 0.110
20ms	6.40, 0.009	6.29, 0.018	6.20, 0.028	6.19, 0.030	6.14, 0.047	6.12, 0.041
30ms	6.41, 0.004	6.45, 0.008	6.33, 0.014	6.26, 0.019	6.24, 0.029	6.19, 0.053
50ms	6.47, 0.001	6.55, 0.005	6.51, 0.007	6.40, 0.013	6.30, 0.016	6.28, 0.019

Table 8.5: Throughput and Loss Ratio with Non-responsive Cross Traffic

Throughput(Gbps)	RTT=5ms			RTT=30ms			RTT=60ms			RTT=100ms		
Loss Rate (%)	CBR	SCT	BCT	CBR	SCT	BCT	CBR	SCT	BCT	CBR	SCT	BCT
RAPID $\tau = 10ms, \eta = \frac{1}{3}$	6.89, 0.000	6.88, 0.016	6.47, 0.051	6.86, 0.000	6.61, 0.014	6.09, 0.060	6.86, 0.000	6.22, 0.019	5.94, 0.007	6.85, 0.001	6.09, 0.035	5.81, 0.062
RAPID $\tau = 50ms, \eta = \frac{1}{4}$	6.67, 0.000	7.08, 0.000	6.65, 0.009	6.65, 0.000	7.00, 0.000	6.40, 0.013	7.03, 0.000	6.97, 0.000	6.33, 0.009	7.03, 0.000	6.96, 0.000	6.26, 0.005
New Reno	5.80, 0.001	5.56, 0.001	5.00, 0.002	1.83, 0.001	1.73, 0.001	1.37, 0.001	0.43, 0.004	0.911, 0.013	0.504, 0.001	0.35, 0.002	0.257, 0.004	0.264, 0.009
Cubic	6.40, 0.004	6.21, 0.002	5.50, 0.002	3.58, 0.002	3.25, 0.002	2.51, 0.002	2.94, 0.003	2.63, 0.001	1.10, 0.004	2.24, 0.004	1.59, 0.005	0.411, 0.011
Scalable	6.85, 0.089	6.88, 0.072	6.59, 0.069	4.77, 0.072	4.38, 0.046	4.20, 0.040	3.71, 0.032	3.14, 0.031	2.92, 0.028	2.61, 0.025	1.92, 0.043	1.44 0.026
BIC	6.17, 0.093	6.26, 0.076	6.20, 0.056	3.96, 0.021	4.02, 0.012	3.87, 0.010	3.61, 0.000	3.00, 0.009	1.80, 0.009	2.62, 0.005	1.92, 0.006	0.893, 0.005
Fast	6.12, 0.141	5.98, 0.132	5.66, 0.159	3.09, 0.145	2.43, 0.232	2.19 0.375	2.56, 0.071	2.13, 0.073	1.56, 0.035	2.34, 0.014	1.36, 0.013	1.28, 0.013
HighSpeed	6.27, 0.006	6.16, 0.011	5.80, 0.013	3.58, 0.005	3.53, 0.003	3.40, 0.002	3.07, 0.006	2.64, 0.003	1.70, 0.002	2.06, 0.003	1.31, 0.003	0.727, 0.048
HTCP	5.68, 0.003	5.57, 0.003	5.14, 0.003	3.45, 0.008	3.18, 0.006	3.06, 0.002	2.93, 0.010	2.06, 0.003	1.13, 0.006	1.36, 0.013	0.822, 0.011	0.866, 0.014
Hybla	5.71, 0.001	5.50, 0.001	5.03, 0.002	1.90, 0.001	1.74, 0.001	1.57, 0.001	1.43, 0.001	1.24, 0.001	0.994, 0.001	1.38, 0.002	1.28, 0.001	0.852, 0.023
Illinois	6.60, 0.004	6.68, 0.007	6.37, 0.010	3.75, 0.002	3.42, 0.002	3.40, 0.002	3.82, 0.087	3.32, 0.011	2.39, 0.010	2.59, 0.068	2.32, 0.006	1.36, 0.071
LP	5.69, 0.001	5.46, 0.002	4.86, 0.002	1.76, 0.001	1.86, 0.000	1.57, 0.001	0.513, 0.003	0.593, 0.003	0.526, 0.004	0.355, 0.003	0.350, 0.003	0.346, 0.003
Yeah	6.86, 0.097	6.91, 0.076	6.63, 0.069	4.91, 0.064	4.30, 0.051	4.08, 0.058	3.73, 0.030	3.16, 0.039	2.29, 0.029	2.01, 0.032	1.73, 0.026	1.43, 0.026
Westwood	5.11, 0.169	5.21, 0.177	5.37, 0.051	1.66, 0.001	1.66, 0.001	1.66, 0.000	0.480, 0.003	0.477, 0.002	0.450, 0.016	0.327, 0.001	0.268, 0.120	0.254, 0.003
Veno	5.62, 0.001	5.44, 0.001	4.79, 0.001	1.46, 0.001	1.39, 0.001	1.53, 0.000	0.646, 0.002	0.595, 0.001	0.517, 0.004	0.42, 0.003	0.392, 0.002	0.312, 0.040
Vegas	5.31, 0.002	5.32, 0.004	5.27, 0.003	2.98, 0.000	2.95, 0.001	2.41, 0.002	0.939, 0.001	0.863, 0.002	0.694, 0.001	0.627, 0.001	0.392, 0.003	0.386, 0.0298
Compound	7.22, 0.002	6.76, 0.002	4.96, 0.010	3.08, 0.003	2.14, 0.003	1.19, 0.014	0.68, 0.019	0.55, 0.023	0.31, 0.024	0.345, 0.013	0.329, 0.017	0.287, 0.021

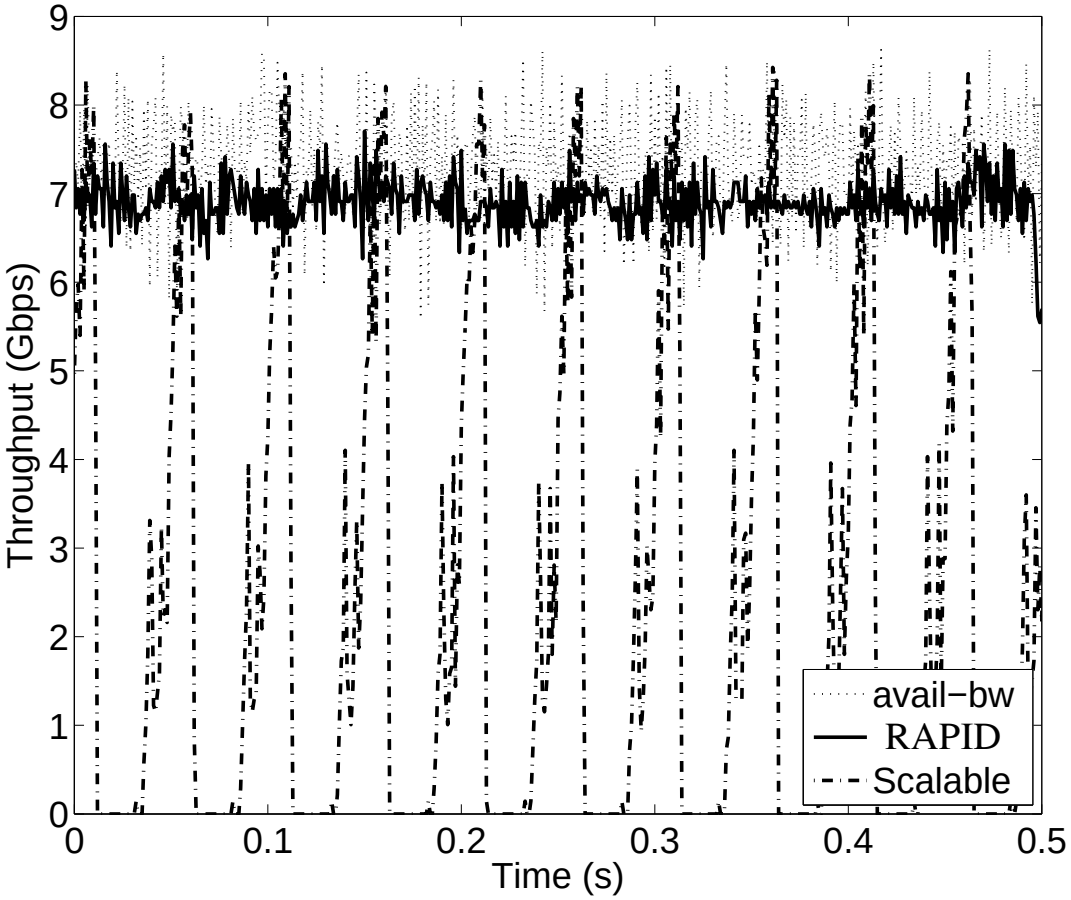


Figure 8.4: Throughput with BCT every 1ms

of burstiness, as described in Table 3.1: the per-millisecond burstiness of CBR, SCT, and BCT ranges from 0.51 Gbps to 2.79 Gbps. A high-speed transfer is initiated to share the bottleneck with the cross-traffic link for 120 seconds, and the average throughput for the TCP flow is reported. We repeat this experiment using RAPID while varying the rate-adapting parameters (τ,η), as well as other TCP variants.

8.3 Impact of τ and η

We first study how τ and η impact RAPID's performance with the smoothest traffic, CBR, and the most bursty traffic, BCT. Fig. 8.3 and Fig. 8.2 show the throughputs achieved with different parameter settings. We find the following:

- RAPID yields higher throughput with the smoother CBR traffic than with the BCT traffic, because the smoother CBT flow has negligible losses; with CBR, RAPID persistently benefits from more aggressive rate-adapting parameters.

- $\tau \times \eta$ is the control knob for rate-adapting aggressiveness; in Table 8.3 and Table 8.4, results that use equivalent $\tau \times \eta$ are given in identical colors. When packet losses are negligible (with CBR, negligible losses are equivalent to RTT$= 5/30\ ms$), the same $\tau \times \eta$ gives highly comparable throughput, which agrees with the findings in [92].

- However, this does not hold when RAPID suffers from frequent losses. With bursty cross traffic, a larger τ leads to higher throughput—a finding in direct contrast to the observation with the smoother CBR traffic. This is because when the avail-bw is highly dynamic, increasing R_{avg} more rapidly (within a smaller timespan of τ) makes it more likely that the path will be overloaded while the p-streams are in transit, resulting in higher loss rates (as shown in Table 8.4). On the other hand, using a larger τ to increase R_{avg} slowly helps to reduce packet losses and consequently yields higher average throughput.

Thus, when the cross-traffic on the network is consistently smooth, we recommend that network operators adopt more aggressive rate-adapting parameter settings (smaller $\tau \times \eta$); a more conservative setting of larger τ is preferred for high throughput if the cross-traffic on the network is bursty.

8.3 RAPID With TCP Variants

Next we compare RAPID's throughput with that of other TCP variants when the path is shared with non-responsive cross-traffic. Table 8.5 shows the average bottleneck link-utilization rates and loss rates observed during high-speed transfers. We find the following:

- A fixed RTT means that as the burstiness of cross-traffic increases, the loss rate also increases, reducing the throughput of the TCP flow. This is true for all protocols, including RAPID, and is to be expected; switches with finite buffer space simply have more losses in the presence of more bursty traffic.

- In spite of the general performance degradation observed with higher cross-traffic burstiness, the throughput of RAPID is less affected than that of other protocols. RAPID is more adaptable to cross-traffic than other protocols; it consistently utilizes a much higher fraction of the bursty avail-bw, especially with larger RTTs. Fig. 8.4, which plots the throughput of TCP flows every 1 ms when

they share the path with BCT, demonstrates that RAPID throughput tracks avail-bw closely, but that Scalable fails to utilize avail-bw after repeated packet losses.

- Despite the higher utilization, RAPID incurs much lower packet-loss rates because of the protocol's negligible queuing [2]. Even with BCT, RAPID ($\tau = 50\ ms, \eta = \frac{1}{4}$) yields 2 Gbps more throughput than Scalable with much smaller loss rates.

- For the state-of-the-art protocols, a certain type of cross-traffic causes TCP flow throughput to decrease drastically as RTT increases. For example, while sharing the path with CBR, the average throughput of Scalable drops by 30% once RTT increases from 5 ms to 30 ms, and by 62% once RTT reaches 100 ms. This is to be expected for these window-based congestion control schemes. The speed of the rate increase in the slow-start/loss-recovery phase is inversely proportional to RTT when the number of packets allowed in transit (*cwnd*) is updated only once per RTT.

- The above observation holds true for all protocols *except* RAPID, which observes only slight throughput reduction. RAPID continuously estimates avail-bw and adapts to it in sub-RTT timescales, which helps it to ramp up its throughput immediately after each loss recovery.

To sum up, *compared with existing TCP protocols, RAPID has significantly better adaptability to dynamic cross-traffic and better scalability to paths with longer RTTs.*

8.4 TCP Friendliness with Web Traffic

Web traffic that is transferred using conventional TCP on low-speed access networks continues to dominate the Internet. A high-speed TCP protocol can be widely deployed on the Internet only if it has minimal impact on co-existing conventional low-speed TCP transfers. To study this, we generate responsive web traffic (as described in Section 3.2) that shares the bottleneck link with a high-speed TCP transfer. We repeat the experiment multiple times using different protocols for the high-speed transfer, with TCP flows that emulate RTT$= 5\ ms$ and RTT$= 30\ ms$, respectively. By setting RTT to as short as 5 ms, we emulate the increasing use of content distribution caches, which store web contents on servers that are geographically close to end users.

The web transfers that share the bottleneck link with high-speed transfers are responsive to any increased delays and losses. One important metric of web traffic performance is flow duration; increased flow duration

strongly reduces user satisfaction. We compute the median, $5-95\%$ percentile, and $10-90\%$ of the distribution of web-flow duration without the TCP flow; we observe how these numbers change when the TCP flow joins the network. Ideally, the TCP flow should fully utilize the spare capacity on the path without significantly increasing these numbers.

8.4 Impact of τ and η

The more quickly RAPID grabs spare bandwidth, the higher its throughput; however, the more transient queuing it causes in bottleneck buffers, the more it impacts the performance of cross-traffic. In RAPID, this trade-off is controlled by the rate-adaptation parameters (τ, η) [2]. We first study the influence of these two parameters. Table 8.6 lists the throughput of the RAPID flow with different parameter settings, and Fig. 8.5 depicts the median, $5\%-95\%$ and $10\%-90\%$ of flow duration as candlesticks and bars. We summarize our findings below.

- *RAPID throughput:* With $\tau = 5/10\ ms$, RAPID throughput first increases with $\frac{1}{\eta}$ due to its more aggressive behavior, and then decreases due to higher induced losses.

  With $\tau > 10\ ms$, RAPID experiences negligible packet losses, and its throughput keeps growing with more aggressive choices of η. Identical $\tau \times \eta$ yields comparable throughput, which agrees with our findings in Section 8.3.1.

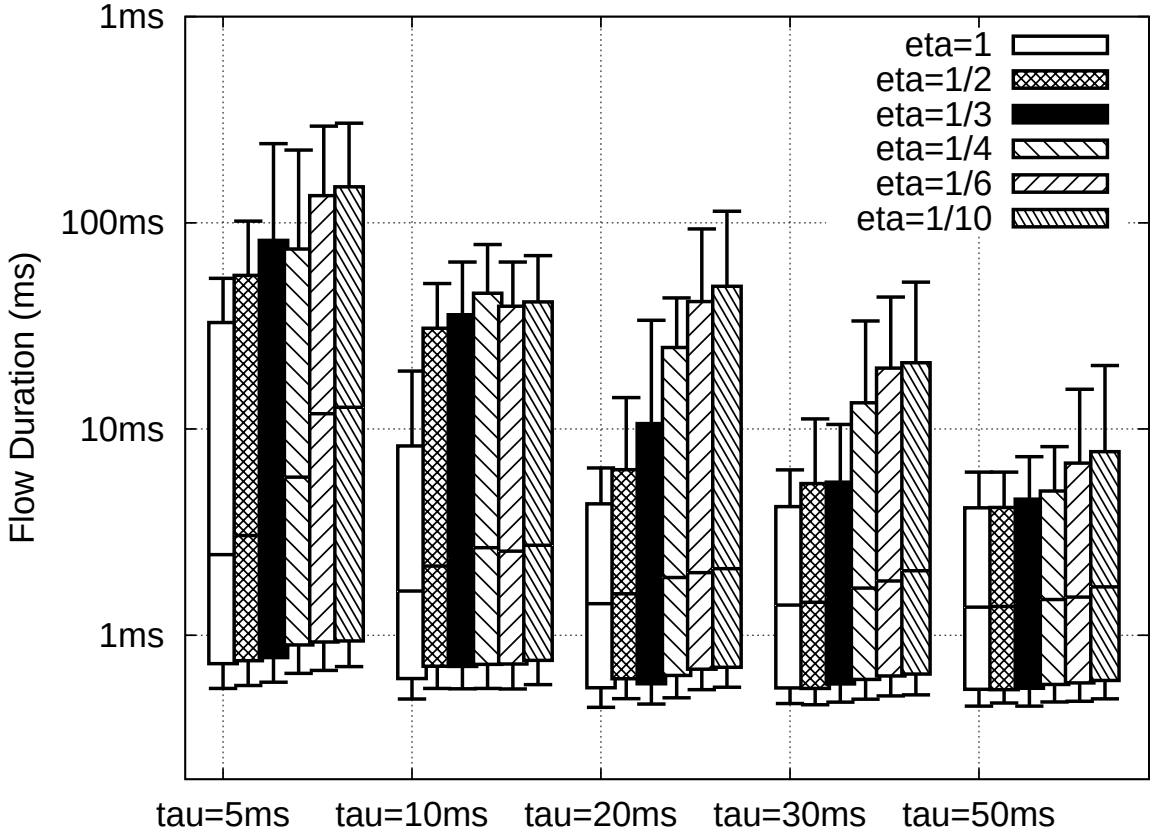
- *Web traffic performance:* As shown in Fig. 8.5(a), a smaller τ or a larger η value increases the duration of co-existing low-speed web transfers. This is expected; an aggressive RAPID flow tends to occupy the shallow buffer more quickly, introducing more losses to web transfers, which will back off by half upon every single loss. However, there are exceptions, as seen in Fig. 8.5(b). When τ =5 ms and 10 ms, the flow duration shrinks with smaller η once $\eta > \frac{1}{3}$. In these cases, the aggressive rate-adaptation parameters cause high rates of packet loss in the RAPID flow, reducing the overall throughput and consequently creating less impact on the web traffic.

- Although identical $\tau \times \eta$ yields similar RAPID throughput, a larger τ helps to reduce the median and the tail of the flow-duration distribution for web traffic. This holds for both RTT values.

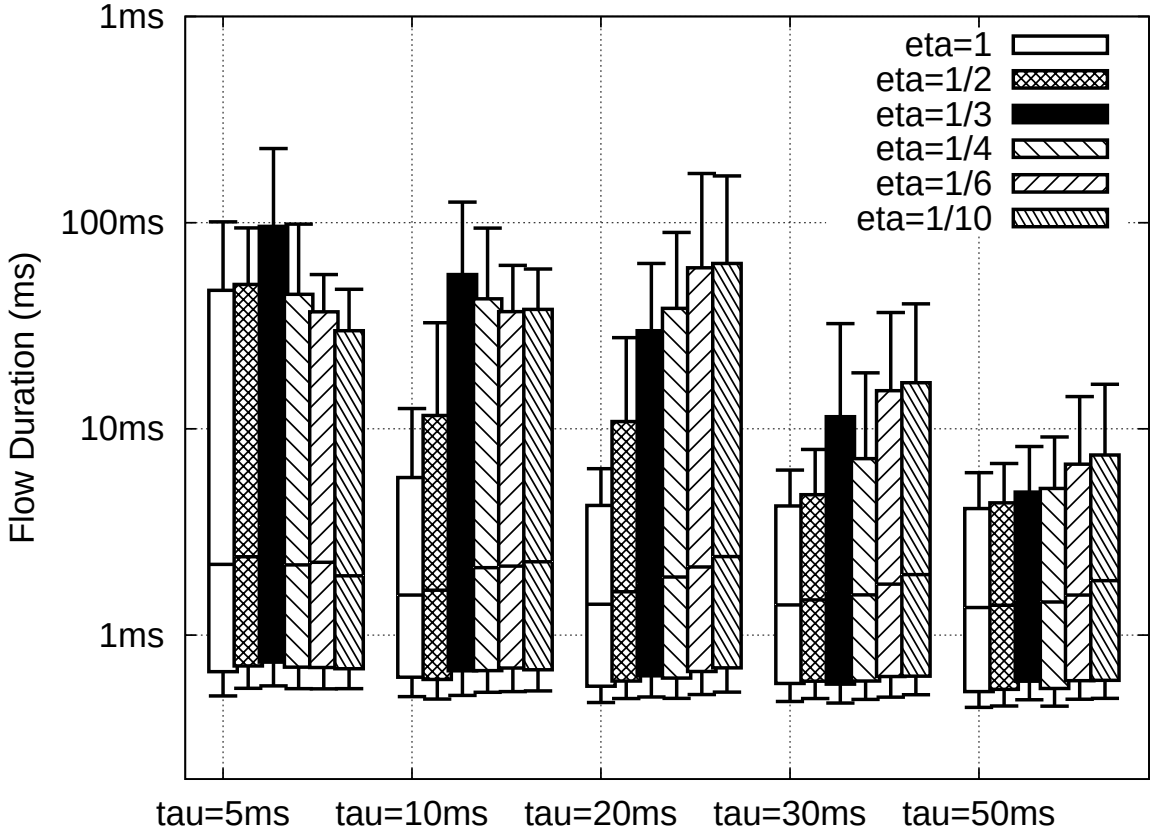
Thus, for network operators seeking minimal impact on web traffic, a more conservative RAPID configuration with a larger $\tau \times \eta$ and a larger τ is recommended. In our testbed $(\tau, \eta) = (50, 4)$ achieves high throughput

Table 8.6: Throughput and Loss Rate of RAPID Flow with Web Traffic

Throughput Loss Rate	η=1	1/2	1/3	1/4	1/6	1/10
RTT=5ms						
τ=5ms	6.81, 0.000	7.01, 0.002	7.03, 0.007	6.99, 0.013	6.89, 0.019	6.84, 0.021
10ms	6.61, 0.000	6.82, 0.000	6.95, 0.000	6.99, 0.007	7.06, 0.004	6.94, 0.006
20ms	6.47, 0.000	6.65, 0.000	6.74, 0.000	6.82, 0.000	6.92, 0.000	7.01, 0.003
30ms	6.39, 0.000	6.58, 0.000	6.67, 0.000	6.73, 0.000	6.81, 0.000	6.90, 0.003
50ms	6.30, 0.000	6.48, 0.000	6.56, 0.000	6.62, 0.000	6.71, 0.000	6.82, 0.000
RTT=30ms						
τ=5ms	6.79, 0.000	6.88, 0.003	6.80, 0.005	6.72, 0.008	6.59, 0.012	6.43, 0.017
10ms	6.61, 0.000	6.81, 0.000	6.91, 0.000	6.93, 0.001	6.92, 0.003	6.83, 0.004
20ms	6.46, 0.000	6.66, 0.000	6.72, 0.000	6.79, 0.000	6.90, 0.000	6.92, 0.001
30ms	6.38, 0.000	6.54, 0.000	6.64, 0.000	6.70, 0.000	6.79, 0.000	6.87, 0.000
50ms	6.27, 0.000	6.46, 0.000	6.55, 0.000	6.61, 0.000	6.69, 0.000	6.79, 0.000



(a) RTT=5ms



(b) RTT=30ms

Figure 8.5: Duration of Web Traffic (RTT=5ms)

while without significantly slowing down cross-traffic flows. We used this parameter setting for the following evaluation.

8.4 RAPID with TCP Variants

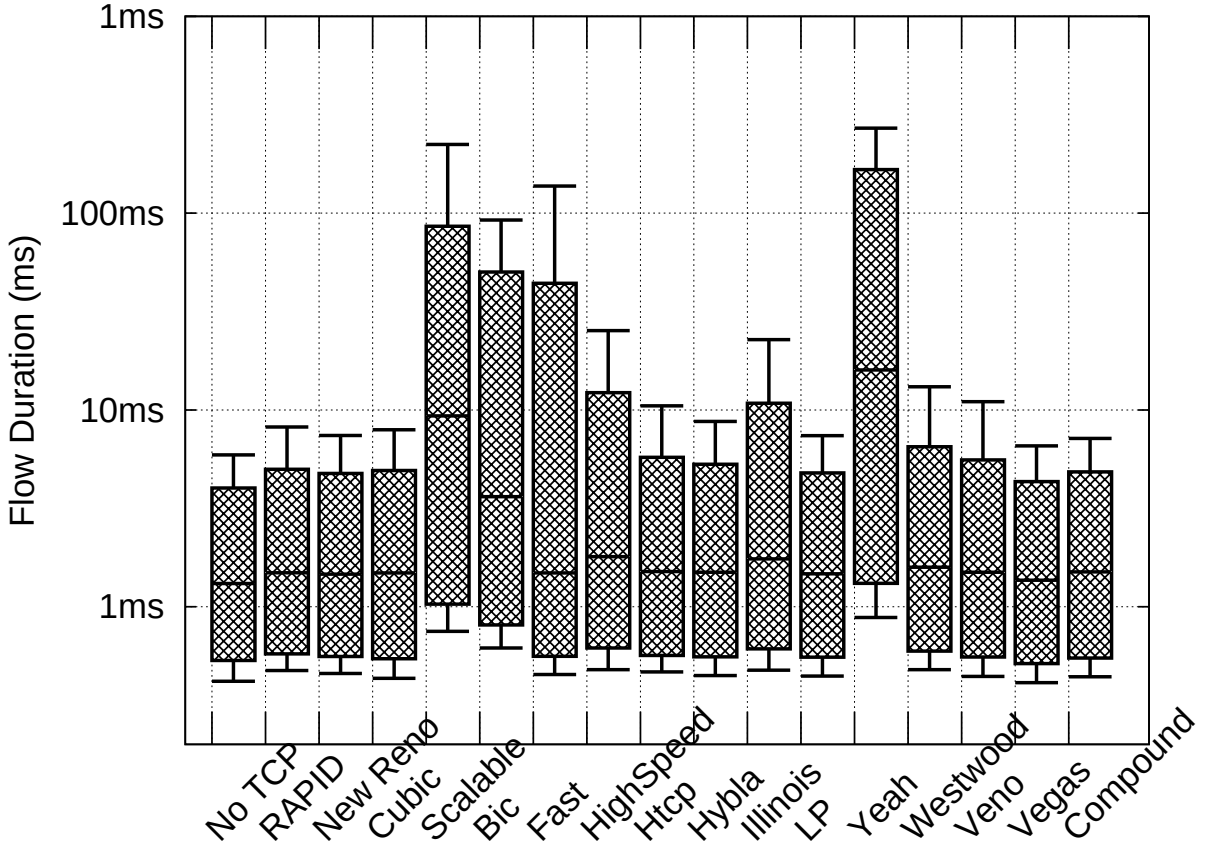
We compare the performance of RAPID ($\tau = 50\ ms, \eta = \frac{1}{4}$) with that of other TCP variants when the high-speed flow shares the path with responsive traffic. Fig. **??** depicts the throughput of those TCP flows with their corresponding loss rates denoted. Fig. 8.7 plots the median, $5 - 95\%$, and $10 - 90\%$ distribution of flow duration of the coexisting web traffic. We find the following:

- Once RTT increases from 5 ms to 30 ms, performance degradation—a drop of over 2 Gbps in throughput—is observed for all protocols except RAPID. Even though RTT is increased, RAPID maintains a good share of bandwidth and a low loss rate. This agrees with the observation in Section 8.3.
- For both RTT values, the RAPID flows experience the lowest loss rates of all the tested TCP protocols, while yielding comparably higher throughputs.
- With RTT$= 5\ ms$, Scalable, BIC, and Yeah yield higher TCP throughput than RAPID, but at the cost of an extreme extension of web-flow duration. For instance, Fig. 8.6(a) shows a comparison between web traffic co-existing with RAPID and traffic coexisting with Scalable; the tail distribution ($> 95\%$) of the web traffic's flow duration is increased by more than 10 times when it shares the path with Scalable.
- With RTT$= 30\ ms$, the RAPID flow enjoys the highest link utilization and does not significantly starve the web traffic. By contrast, Scalable's throughput is 2.38 Gbps lower than RAPID's while significantly increasing the duration of web flows.

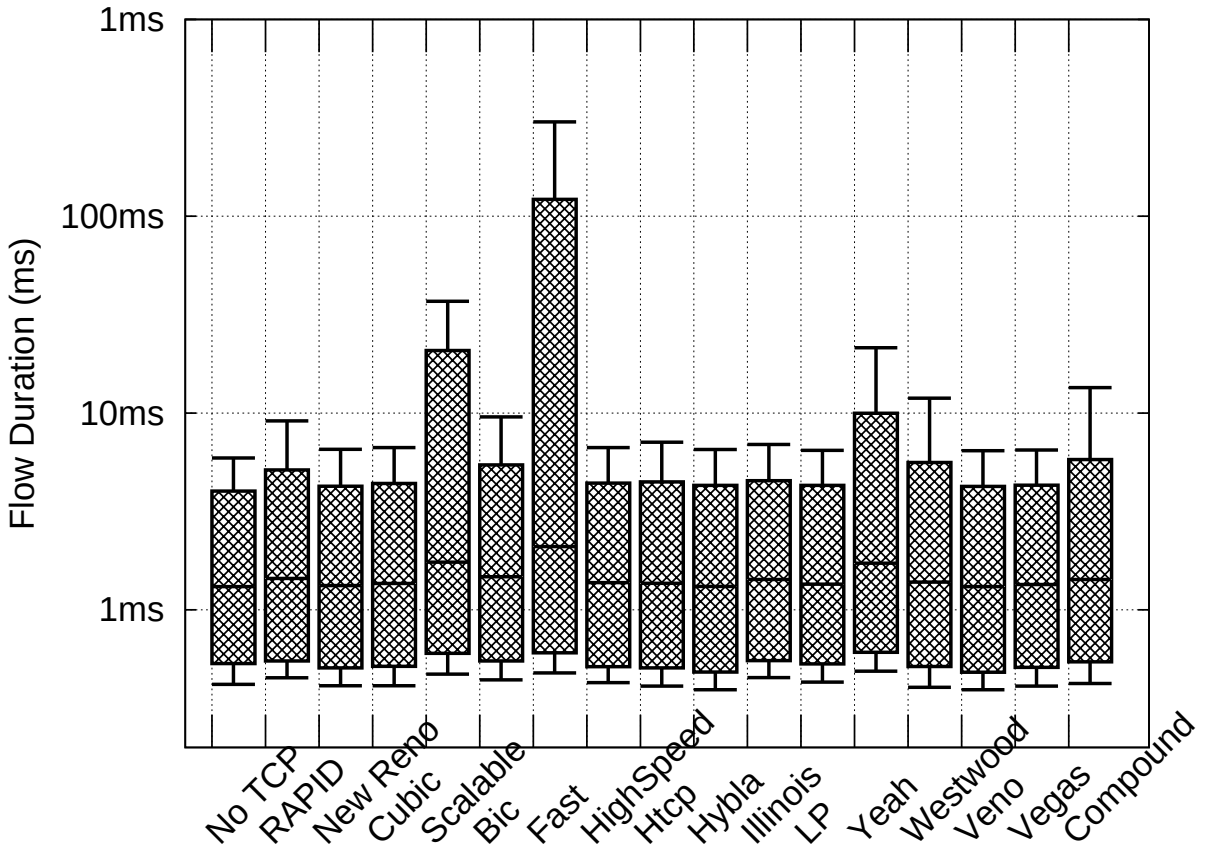
We conclude that among existing TCP protocols, RAPID best balances link utilization with TCP-friendliness; it achieves considerable link utilization while starving conventional TCP traffic less than other tested protocols.

8.5 Intra-protocol Fairness

In order to evaluate our implementation's intra-protocol fairness properties, we next look at a situation in which multiple active TCP flows are using the same congestion control protocol. We initiate three iperf flows

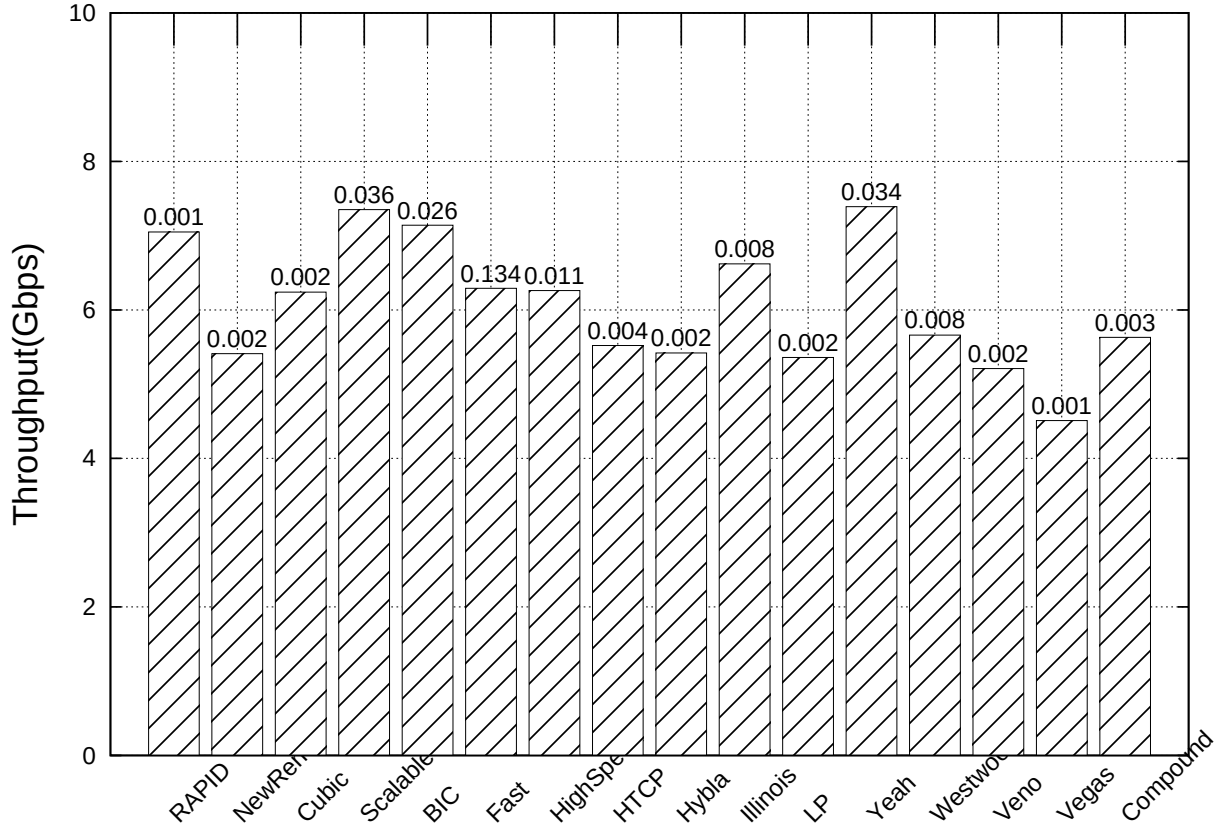


(a) RTT=5ms

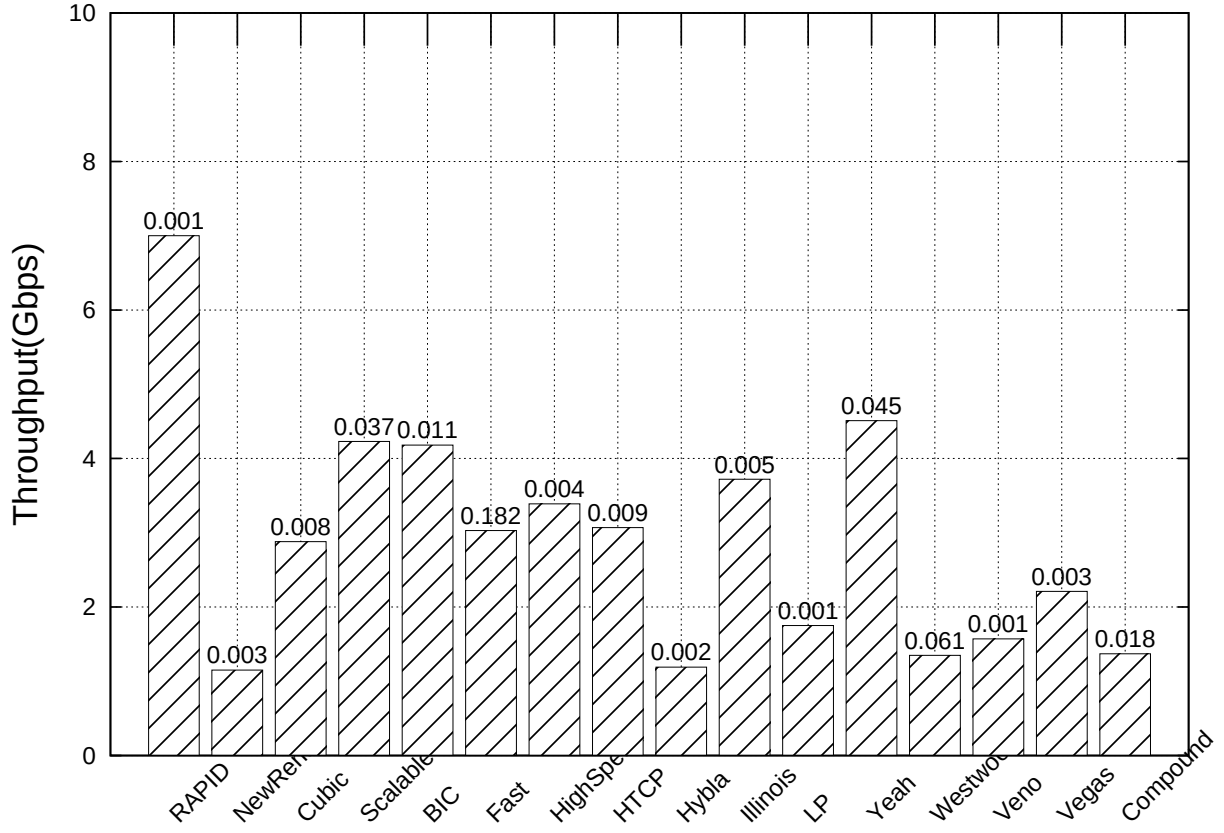


(b) RTT=30ms

Figure 8.6: Flow Duration of Web Traffic



(a) RTT=5ms



(b) RTT=30ms

Figure 8.7: Throughput and Loss Rate of TCP Flow When Sharing the Path with Web Traffic

between two pairs of end hosts. Each transfer emulates RTT$= 30\ ms$, and the transfers are active during different time intervals. In the beginning of each experiment, we start one flow between the first pair of hosts on the idle 10 Gbps path; this trial lasts for 150 s. After 30 s, we start another flow between the second pair of hosts; this trial lasts for 90 s. Another 30 s later we start the third flow between the first pair of hosts; this trial lasts for 30 s. With the optimal intra-protocol fairness property, the three flows should have an equal share of bandwidth between 60 s and 90 s. The first and second flows are expected to yield 5 Gbps throughput between 30 s and 60 s and between 90 s and 120 s.

Fig. 8.8 depicts the time-series of the throughputs achieved by the three transfers with different underlying protocols. Loss-based protocols, such as Reno, Cubic, Scalable, HTCP, Highspeed, and Yeah, exhibit strong bias toward the first flow; for these protocols, the flow that first occupies the path's buffers leaves few network buffers open to new flows, and this first flow consequently shows higher throughput. This agrees with the findings in previous research [11]. Protocols that use delays as the congestion indicator perform no better than loss-based protocols. As shown in Fig. 8.8(o), the first Vegas flow reduces its rate drastically after 30 s, when the second flow joins the path, but the sluggishness of the *cwnd* increase prevents the second flow from efficiently grabbing its fair share of bandwidth.

RAPID demonstrates fairly good intra-protocol fairness, as shown in Fig. 8.8(a). Because it persistently estimates bandwidth at sub-RTT timescales, RAPID is able to agilely detect the presence of a new flow and adapt its transmission rate to a fairly equal share of the bandwidth. This result is very close to its fairness property observed under simulations in [2].

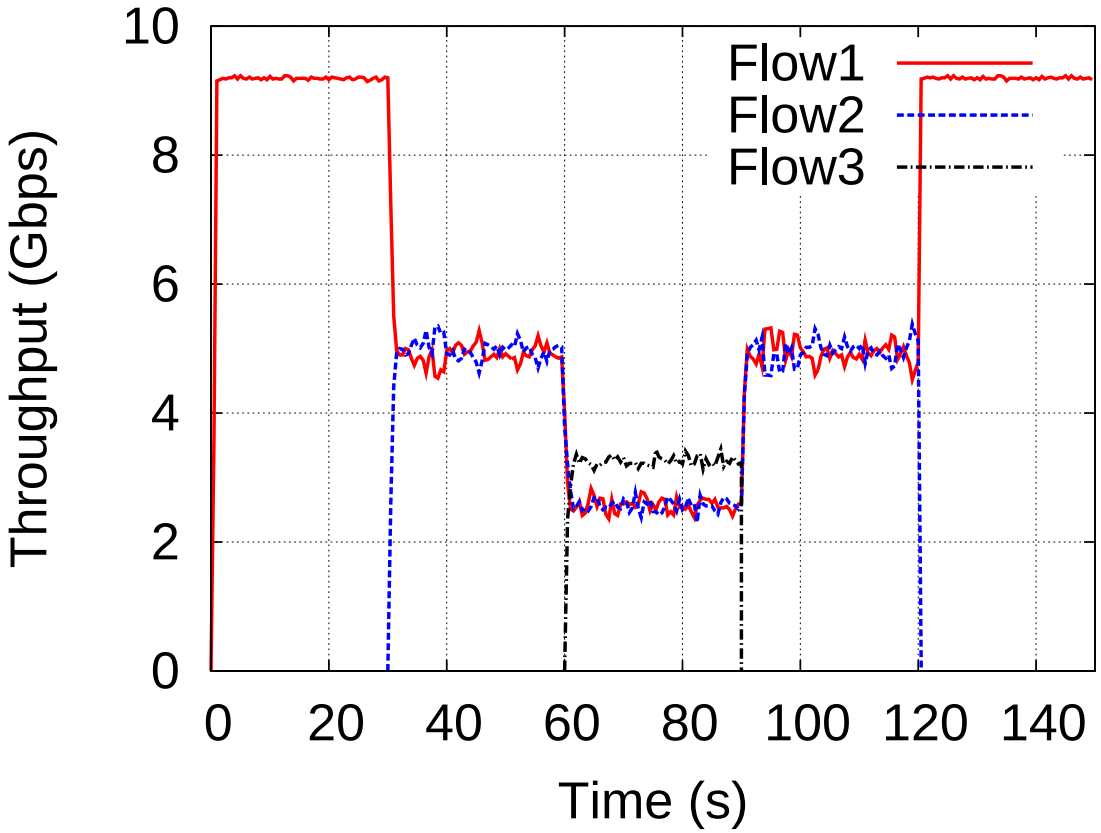
8.6 Necessity of Dealing with Challenges

We have made several efforts in order to realize a RAPID implementation in real-world Linux system (vs. a simulator, as in [2]). These include:

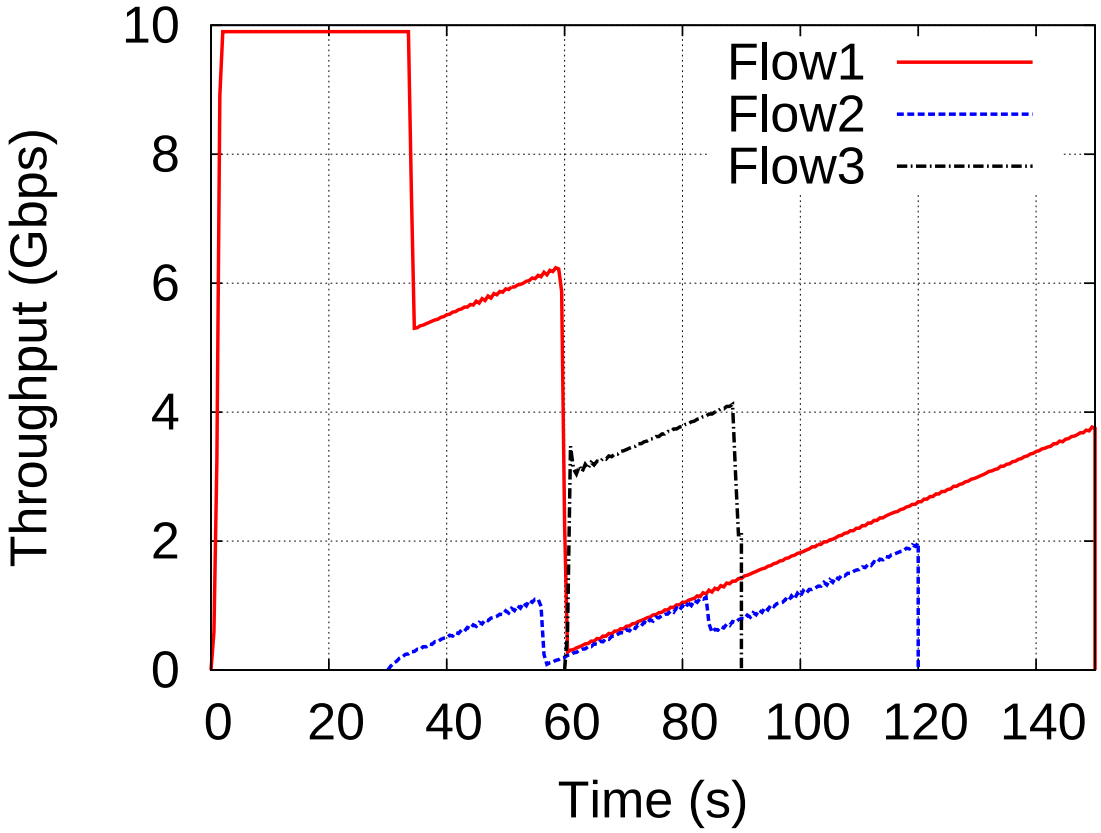
1. Implementing "Dummy-packet" mechanism to create accurate gaps of 1μ precision in Chapter 4;

2. Implementing a set of Qdiscs in Chapter 7 — I_R, E_R to timestamp packet arrivals with μs accuracy at the receiver, and I_S at the sender to correctly retrieve the packet arrival times from ACKs;

3. Using BASS introduced in Chapter 5 to denoise p-stream gap observations for achieving accurate bandwidth estimation with short p-streams of $N = 64$;

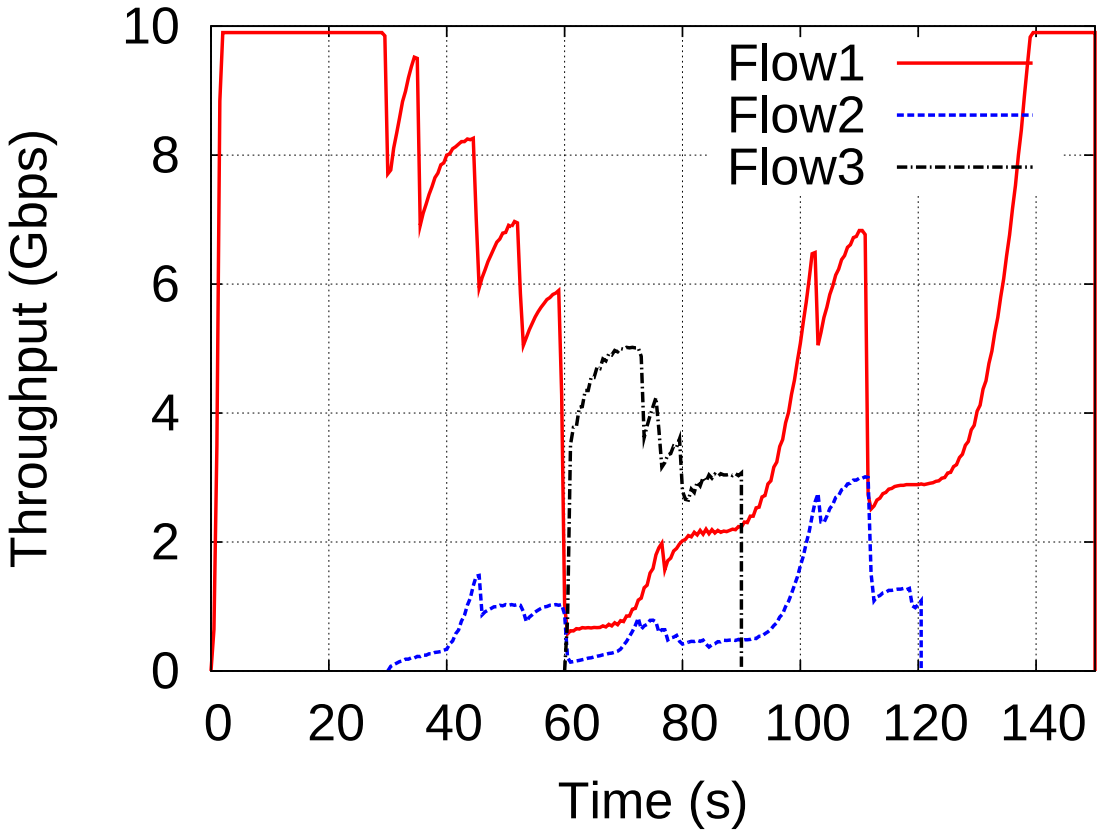


(a) Rapid

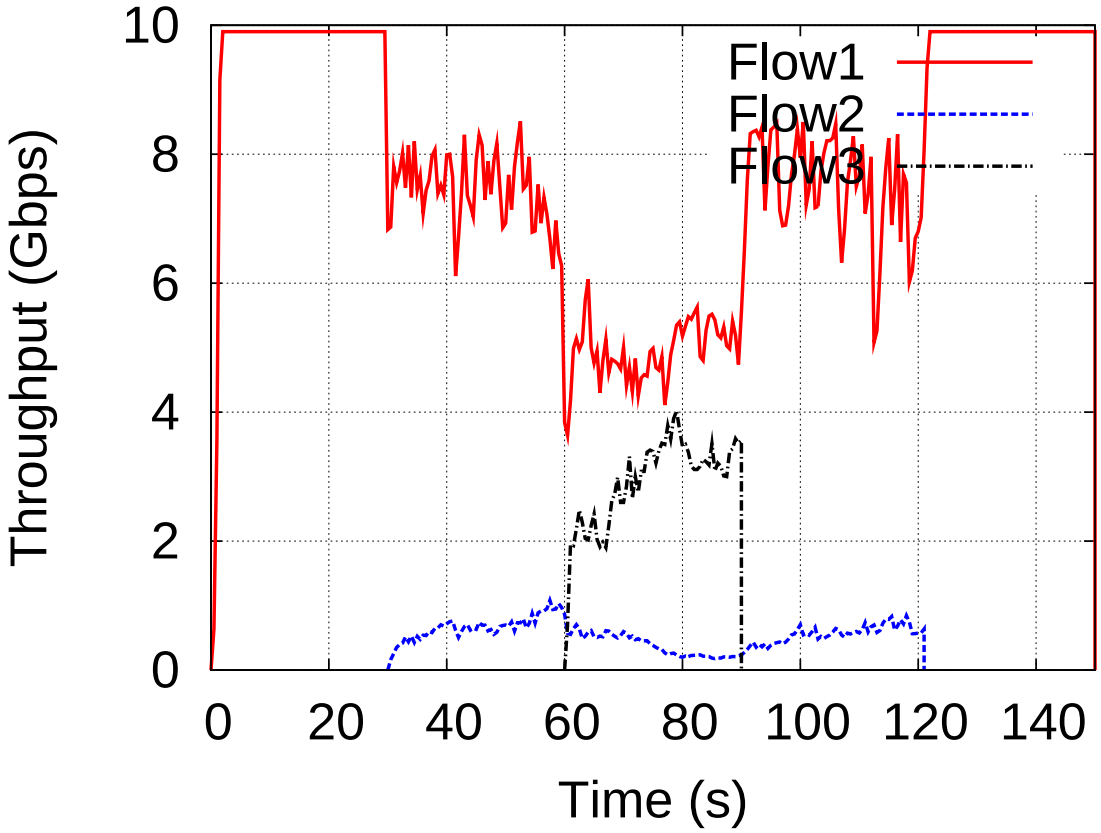


(b) New Reno

Figure 8.8: Intra-Protocol Fairness

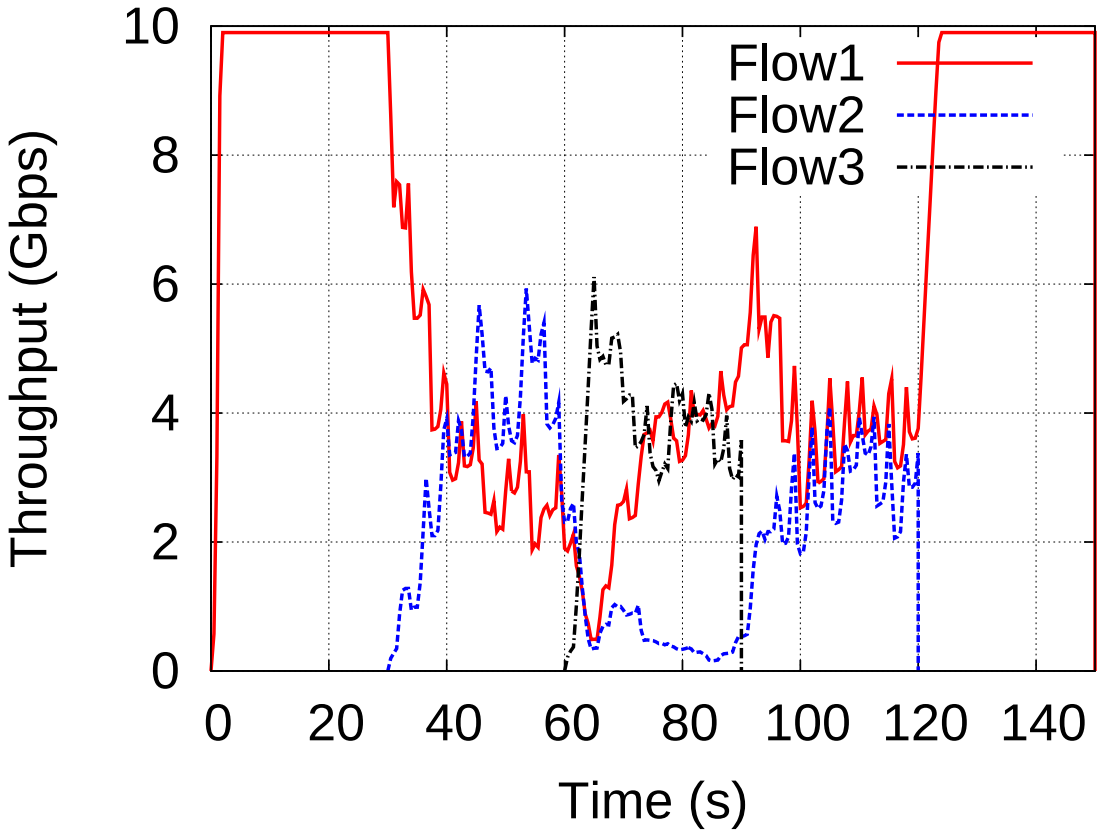


(c) Cubic

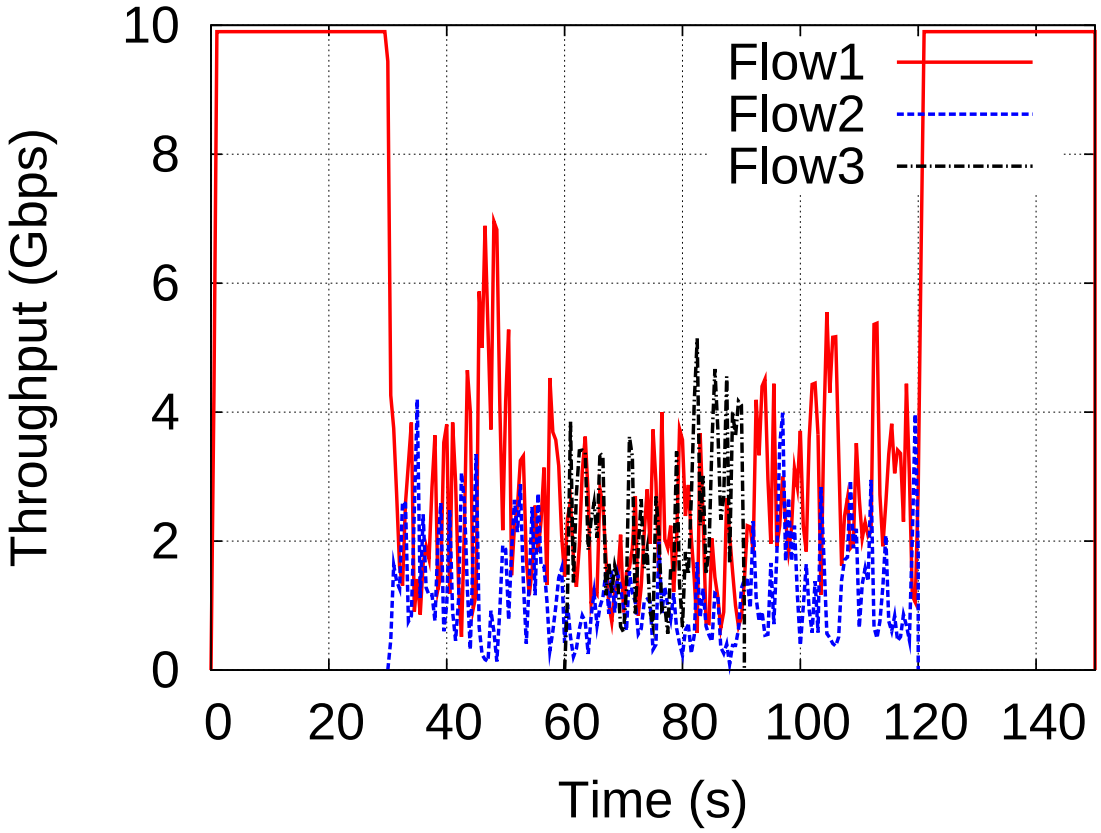


(d) Scalable

Figure 8.8: Intra-Protocol Fairness (Cont.d)

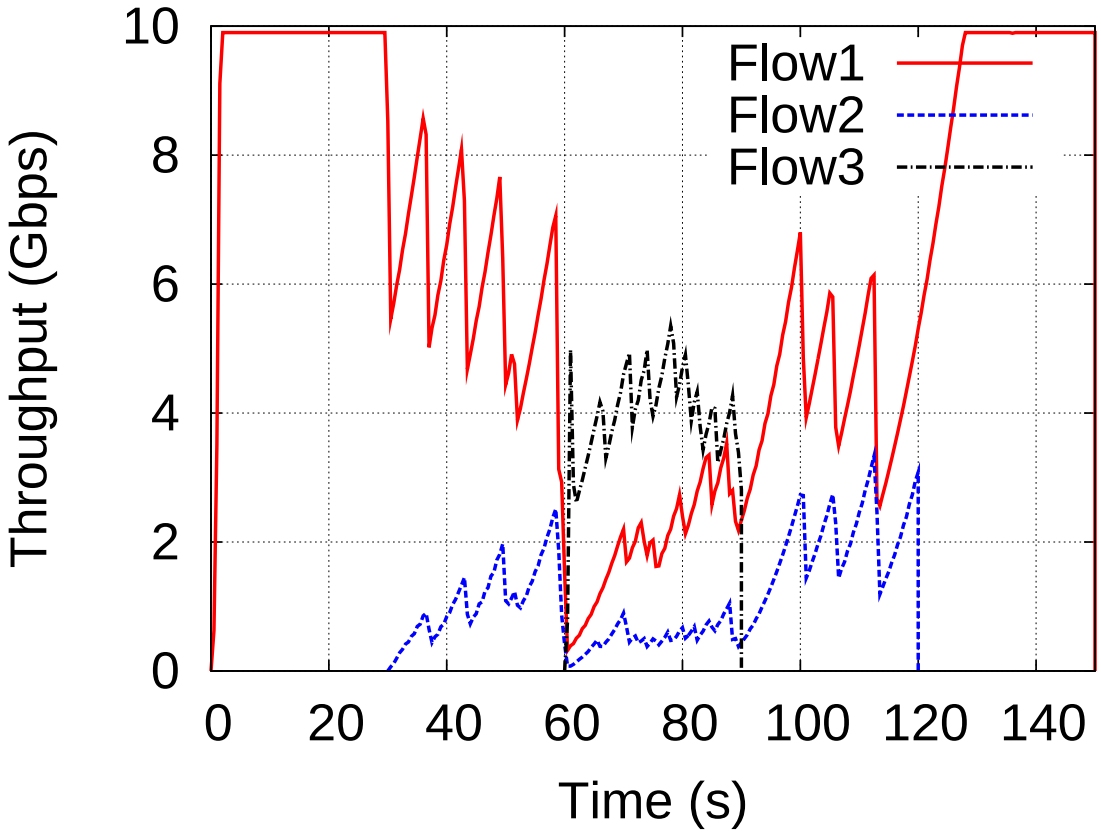


(e) BIC

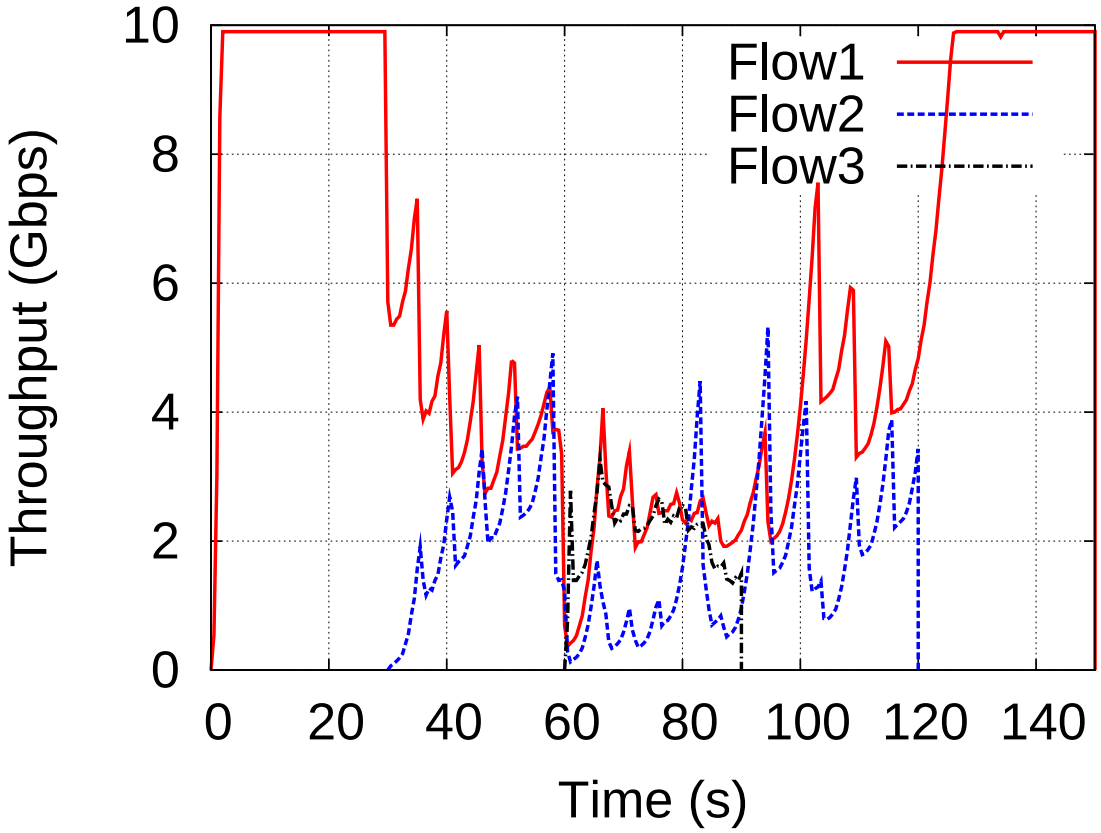


(f) Yeast

Figure 8.8: Intra-Protocol Fairness (Cont.d)

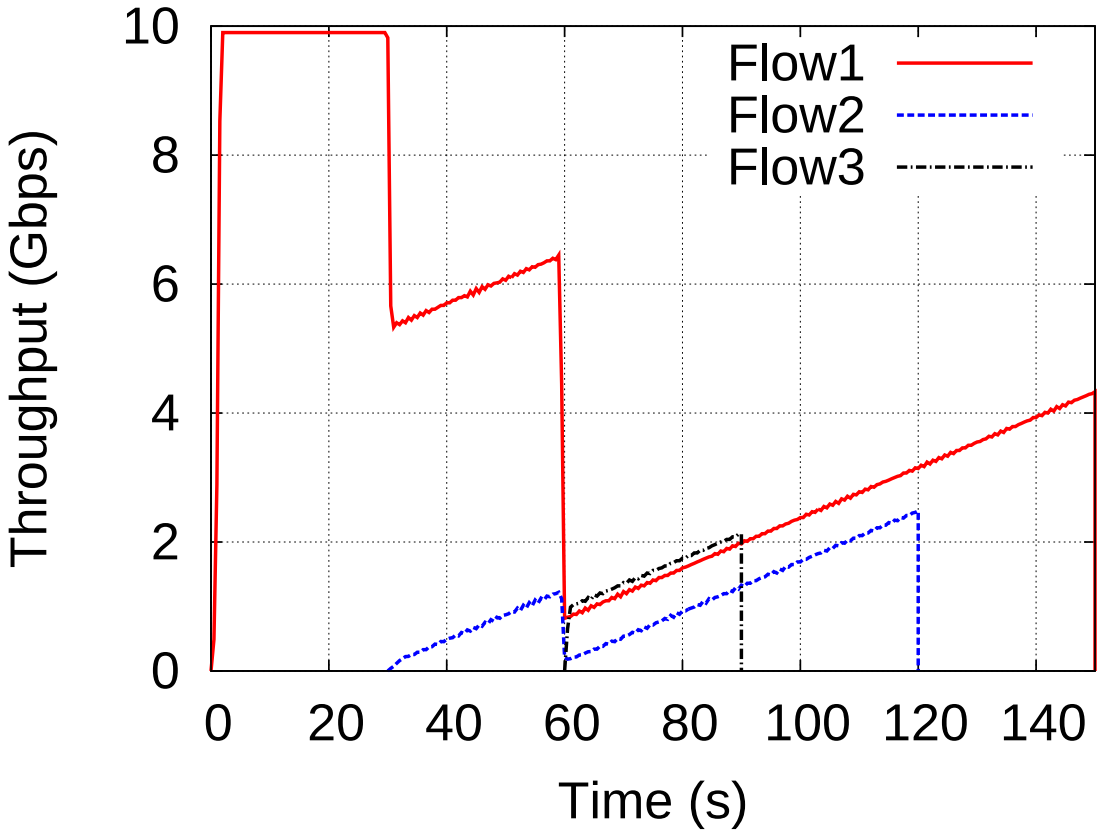


(g) HighSpeed

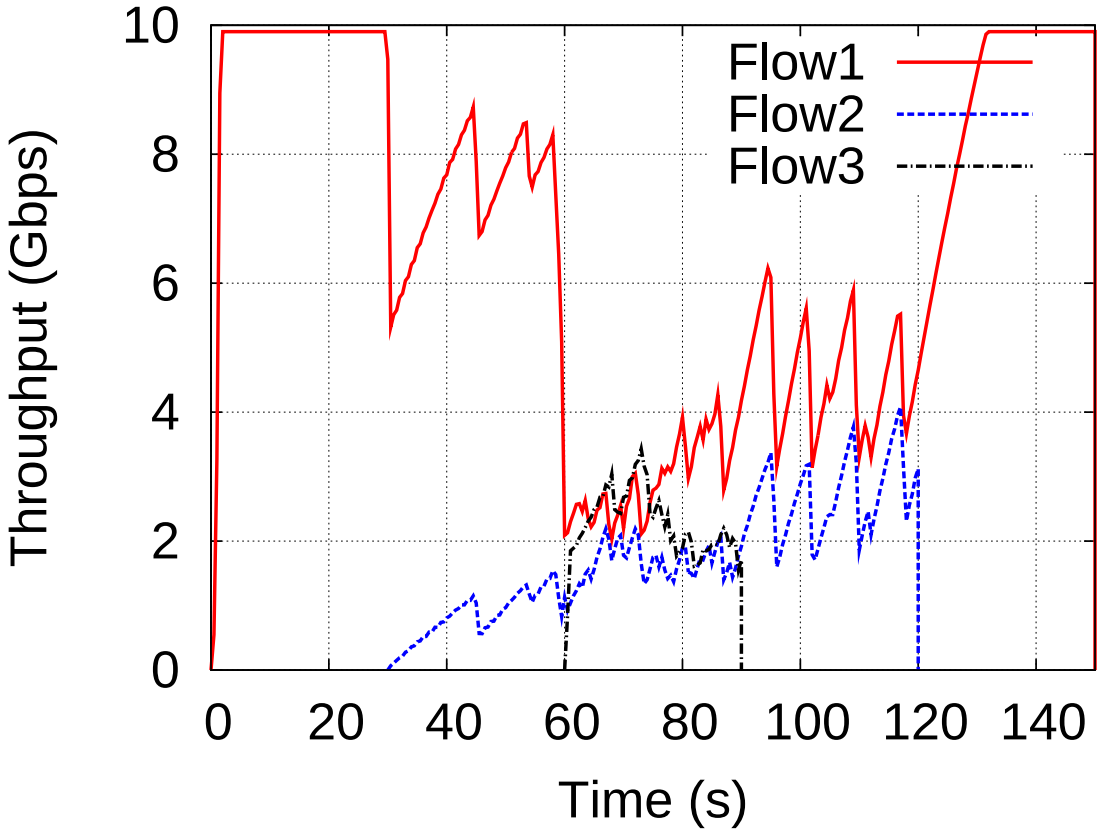


(h) TCP

Figure 8.8: Intra-Protocol Fairness (Cont.d)

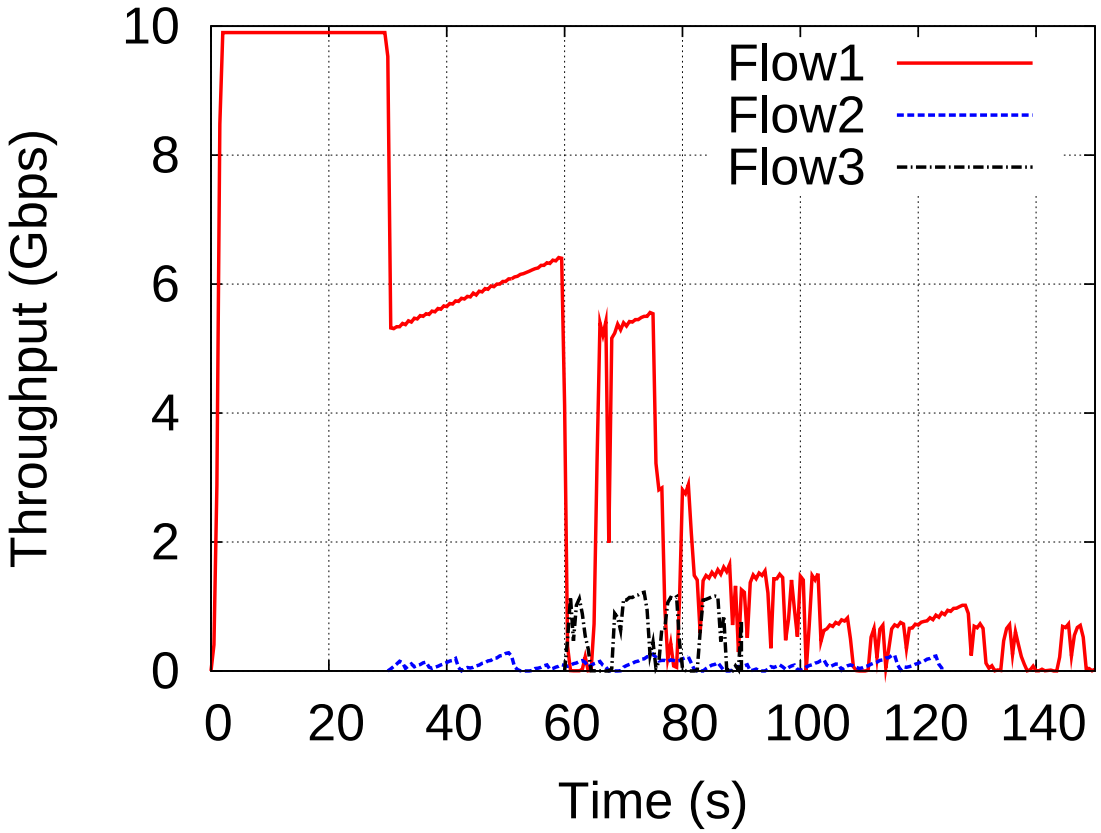


(i) Hybla

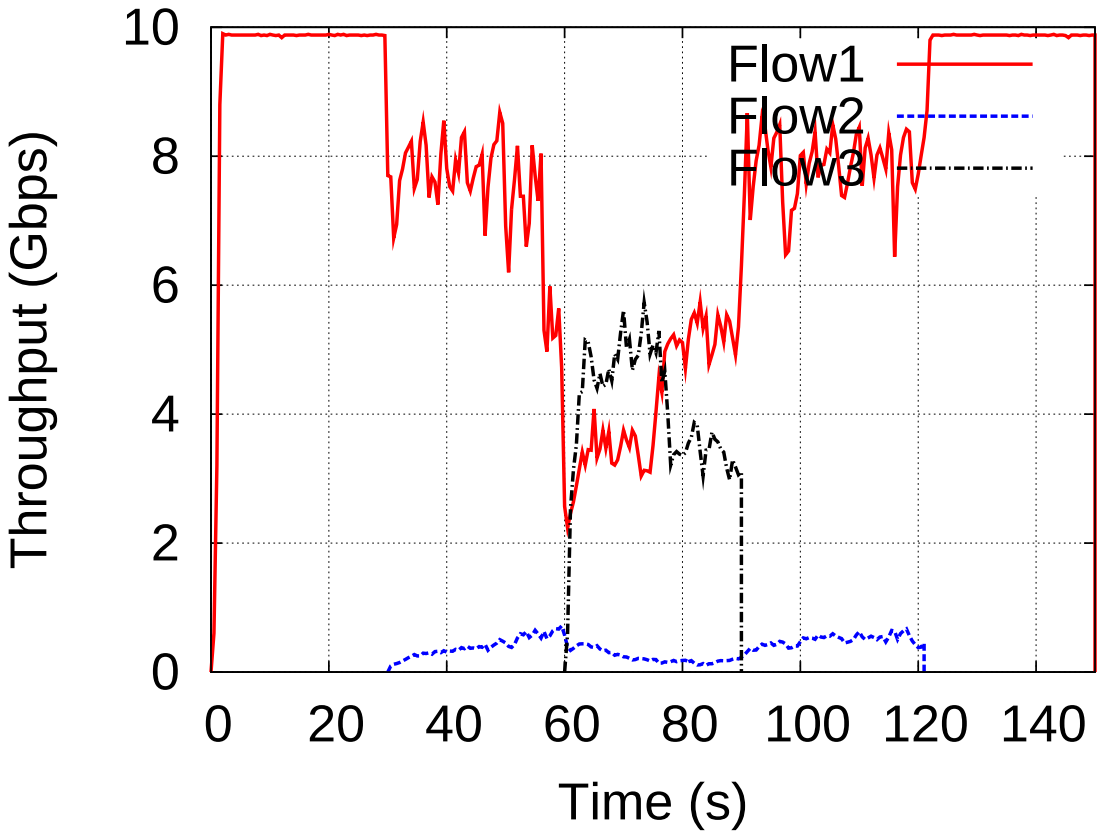


(j) Illinois

Figure 8.8: Intra-Protocol Fairness (Cont.d)

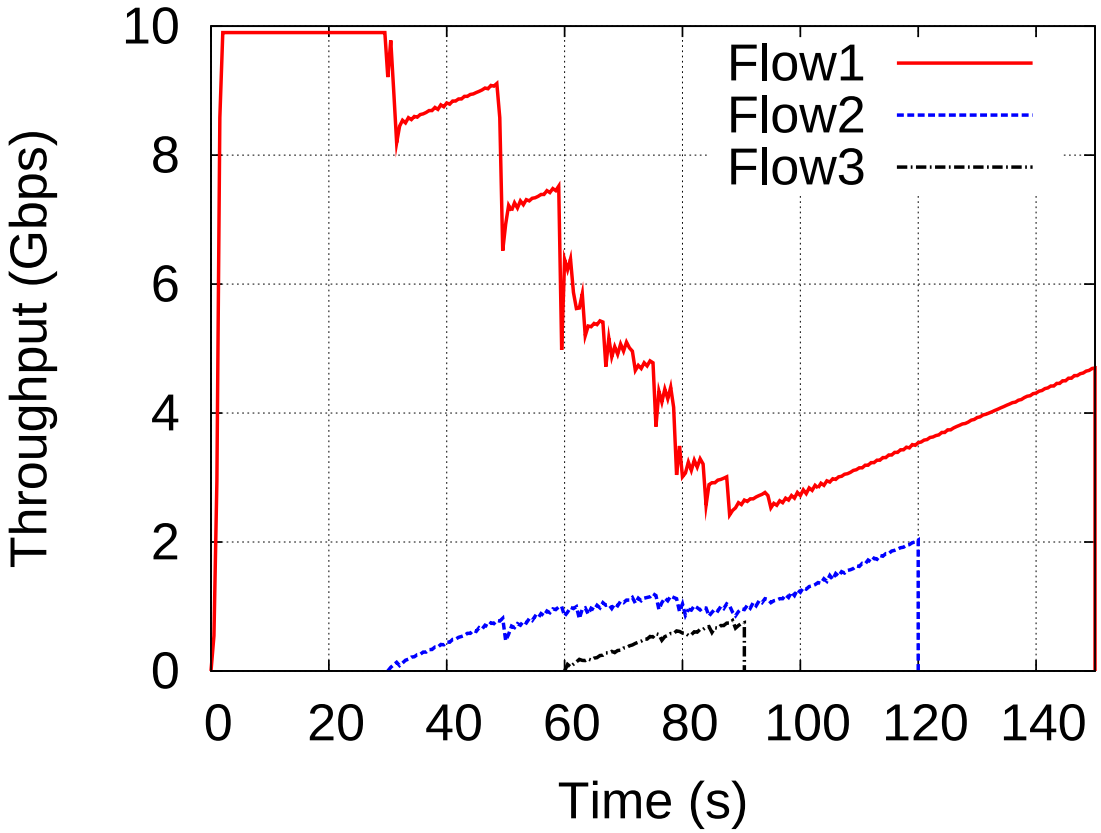


(k) LP

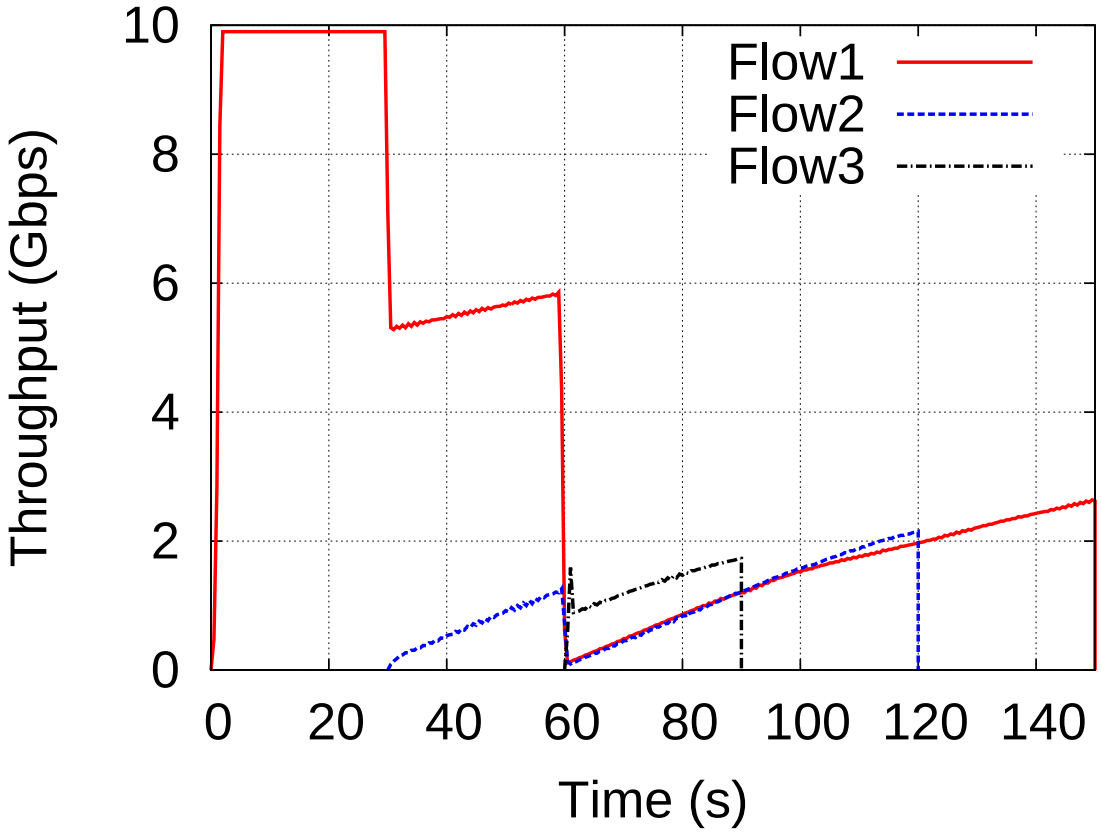


(l) Veah

Figure 8.8: Intra-Protocol Fairness (Cont.d)

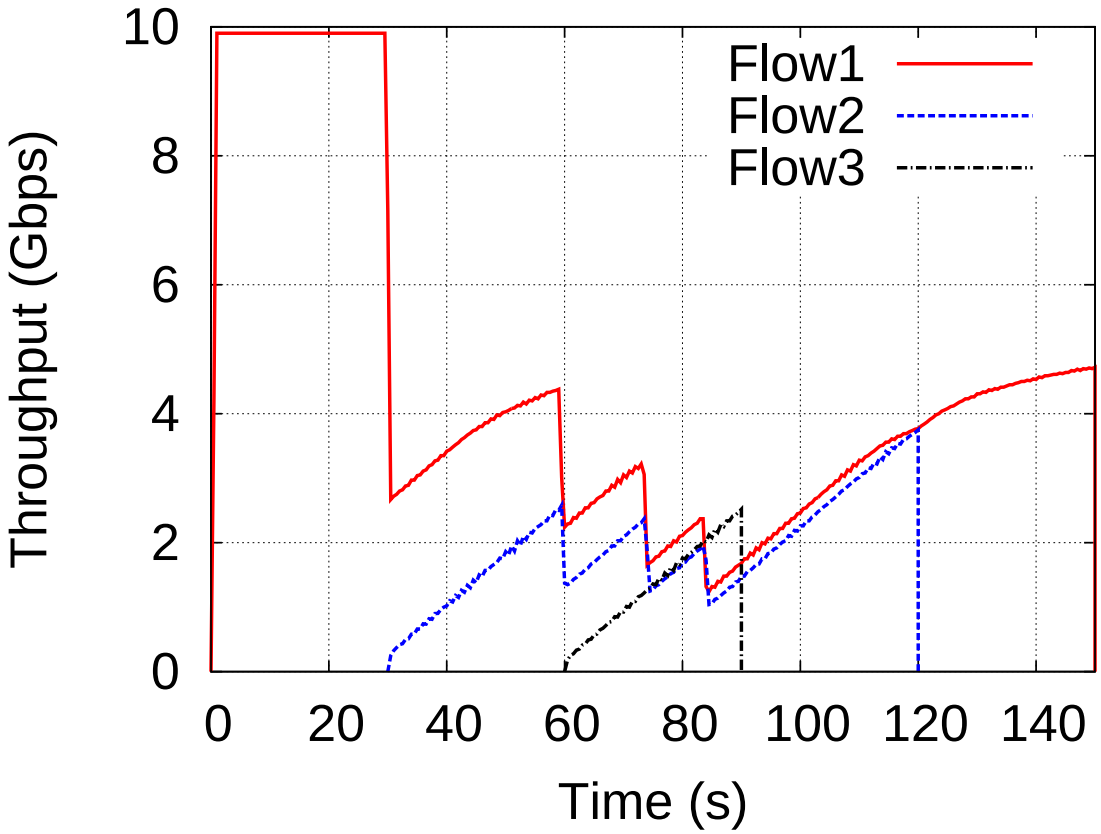


(m) Westwood



(n) Veno

Figure 8.8: Intra-Protocol Fairness (Cont.d)



(o) Vegas

Figure 8.8: Intra-Protocol Fairness (Cont.d)

4. In this chapter, decoupling the probing and adapting timescales to alleviate the stability-vs-adaptability trade-off by adapting to avail-bw estimates every γ p-streams.

In this section, we ask: *are each of these necessary for achieving RAPID performance gains in high-speed settings?*

To answer this, we remove each of the four mechanisms individually in the current implementation, and make five different variants of the RAPID implementation as follows:

- V1: Instead of employing *Dummy-packet* approach to create inter-packet gaps, V1 adopts the software-interrupt-based approach using *Hrtimer*. It removes the PAUSE-frame insertion function in E_S, but generates interrupts to trigger packet transmission in interrupt handlers with $qdisc\_watchdog$. All other RAPID mechanisms are retrained.

- V2: This version gets rid of high-resolution packet-arrival timestamping, but solely relies on TCP timestamp options for computing g_r. Thus the timestamps RAPID reads from ACK headers have only millisecond resolution. All other RAPID mechanisms are retrained.

- V3: Rather than timestamping packet arrivals with I_R, V3 generates timestamps when ACKs are being generated. E_R records the time when each ACK arrives at the Qdisc in μs resolution, and replaces the TSval field in the ACK header with this value. All other RAPID mechanisms are retrained.

- V4: This version opts out BASS denoising, but simply applies a sliding-window smoothing with $AVG\_WINDOW = N_p$ to the raw g_r, and then feeds the smoothed gaps to the bandwidth estimating logic. All other RAPID mechanisms are retrained.

- V5: V5 does not decouple the probing/adapting timescales. Rather, it sets γ to 1 to make the two timescales identical. In other word, both of probing and adapting timescales are determined by the p-stream length N. Recall that in Section 8.1 the probing timescale is $N = 64$ and the rate-adapting timescale is $\gamma \times N = 192$. In V5 we consider both N values ($N_r = 4$ for all p-streams). All other RAPID mechanisms are retrained.

For each of the five versions, we repeat the experiments from Section 8.3 and Section 8.4. In all experiments the rate-adapting parameter is ($\tau = 50ms, \eta = \frac{1}{4}$), and RTT=30ms. Table 8.7 lists the performance of different versions. We find that:

- V1: With non-responsive cross traffic, V1 persistently yields higher loss rates and lower throughputs then RAPID. In Chapter 4, Fig 4.5(b) illustrated that there are significant errors in the intended send gaps using *HrTimer*. Besides, Fig 4.9 also indicates severe over-estimation. The throughput achieved by RAPID is naturally impacted. However, the impact of inaccurate g_s is lower than expected — when sharing the path with responsive traffic, V5 does not starve web flows severely. This is due to the alleviation by BASS, which is good at handling buffering related noise, even if noise is introduced while creating inter-packet gaps at the senders.

- V2: We find that V2 persistently over-estimates avail-bw as full link capacity. This is because with the low millisecond-resolution receiver timestamps, the bandwidth estimator finds no evidence of persistent queuing. This also explains why V2 persistently obtains higher throughput than RAPID. Such over-estimation starves cross-traffic (the median of web-flow duration increases by 500ms), and causes considerably high packet losses.

- V3: The one-way delays involve ACK processing delay at the receiver side, which can delay packets for up to 70μs according to Fig 1.4. Consequently, V3 under-estimates the avail-bw, and under-utilizes even at idle bottleneck link.

- V4: Similar to V2, V4 also persistently over-estimates the avail-bw as 10 Gbps—the spike-dips pattern (Fig 1.3) in the p-streams wipes out any underlying trend of persistent queuing delays. As a result, it incurs significant packet losses, and severely impacts web traffic by making the median duration longer than 700ms.

- V5: With identical probing and adapting timescale, both $N = 64$ and $N = 192$ yield lower throughput and higher loss rates than RAPID— neither a shorter, nor a longer timescale outperforms the decoupled RAPID. A short timescale (N=64) suffers from noisy $ABest$—it fails to fully utilize the empty path. A longer timescale increases the duration for which each p-stream overloads the bottleneck, causing more losses. Also, with cross traffic, $ABest$s at N=192, γ=1 are less accurate than at N=64, γ=3. Consequently, it yields less goodput.

To sum up, we find that *each* of the four design components added in this research, as listed earlier in this section, is critical for ensuring that RAPID achieves its promised performance in practice.

Table 8.7: Necessity of Implementation Mechanisms

RTT=30ms	No cross traffic (Gbps)	With Non-responsive Traffic Throughput (Gbps), Loss(%)			With responsive web traffic	
		UDP	SCT	BCT	Throughput(Gbps), Loss(%)	median flow duration (ms)
Full RAPID tau=50ms, eta=1/4	9.44	6.65,0.000	7.00,0.000	6.40,0.013	7.00, 0.001	1.45
V1	8.37	7.06,0.006	6.37,0.038	6.08, 0.070	6.77, 0.004	1.28
V2	9.83	5.25,3.246	5.12,3.381	4.97,3.141	7.64, 0.217	520.64
V3	7.91	6.81,0.000	6.80,0.000	6.45,0.008	6.43, 0.001	1.24
V4	9.82	6.18,1.266	6.08,1.175	6.00,0.928	7.83, 0.151	8705.24
V5 (N=64)	8.99	6.44,0.002	6.76,0.001	6.17,0.018	6.45, 0.002	1.23
V5 (N=192)	9.49	6.65,0.001	6.87,0.001	6.13,0.026	6.33, 0.007	1.45

8.7 Summary

In this chapter, we conduct evaluation of RAPID congestion control using our Linux implementation. We first study the impact of two parameters, τ and η, that control how aggressively the rate-adapter adapts to changes in avail-bw. Based on the experimental results, when the RAPID flow experiences negligible losses, identical $\tau \times \eta$ settings yields comparable throughput. However, once the cross traffic grows more bursty, a larger τ helps to reduce packet losses and to obtain higher link utilization. Meanwhile, when sharing the path with responsive web-like traffic using the legacy TCP, a larger τ imposes less impact of the low-speed TCP flows.

We also compare the performance of RAPID with other promising TCP variants in our 10Gbps testbed. We find that our implementation of RAPID truly lives up to the performance gains stated in [2] — it achieves higher link utilization with bursty cross traffic, incurs fewer packet losses for lower queuing footprints, maintains friendly to co-existing TCP flows, and achieves better intra-protocol fairness.

Furthermore, we investigate the necessity of the four mechanisms developed for RAPID in this thesis, namely, "Dummy-packet" gap-creation mechanism for gap-creation, Qdiscs for μs resolution timestamping of packet arrivals, BASS denoising mechanism for robust bandwidth estimation, and decoupling probing/adapting timescale using γ. We remove them one by one from our RAPID implementation, and find that each of these is necessary to achieve the full performance gains of our RAPID implementation.

CHAPTER 9: APPLICATION TO OTHER PROTOCOLS

This dissertation addresses the challenges faced by TCP RAPID on 10 Gbps networks: implementing gap-clocking and μs-second receiver timestamping in kernel modules; creating precise inter-packet gaps; denoising for robust bandwidth estimation; and addressing the trade-off between stability and availability. These challenges do not only apply to RAPID; they are also relevant to other delay-based and bandwidth-based protocols and bandwidth-estimation tools, although none of these other protocols impose requirements as stringent as those of TCP RAPID — some examples are provided below:

- The bandwidth estimation tool Pathload [20] is, like RAPID, vulnerable to the noise introduced by receiver-side interrupt coalescence. But Pathload's single-rate probing framework is regarded as more robust to noise than the multi-rate probing framework used by RAPID [35].

- Other rate-based protocols, such as NF-TCP [26], only create gaps intermittently; RAPID must engage in gap-creation for the transmission of every single packet.

- TCP Pacing [62], like RAPID, enforces pacing rates by creating per-packet gaps, but the performance of TCP Pacing is not degraded by slightly inaccurate gaps. The logic of bandwidth estimation in RAPID is based on precise probing rates for *every single* packet; a single erroneous inter-packet gap has the potential of severely misleading RAPID's bandwidth estimation.

- Delay-based protocols like TCP Vegas and Fast enjoy the luxury of denoising delay samples once per RTT, which relaxes the timestamping accuracy required. In contrast, RAPID uses packet-arrival timestamping for each individual packet to observe fine-scale changes in each packet's one-way delay, and it therefore requires exceptional timestamping accuracy.

As demonstrated in Chapter 8, TCP RAPID, an incredibly demanding protocol, is able to achieve expected performance gains (demonstrated in simulation-based evaluations) by addressing these challenges with the mechanisms developed in previous chapters. These mechanisms can therefore also significantly

improve the performance of other protocols and tools that face similar challenges. In this chapter, we apply three mechanisms developed for RAPID to other bandwidth-estimation tools and TCP protocols and investigate how their performance can be improved by addressing these challenges.

9.1 Applying BASS to Other Bandwidth Estimation Tools

Chapter 5 surveys the state-of-the-art bandwidth estimation tool IMR-Pathload, and shows that its denoising algorithms — window-based averaging (IMR-avg) and wavelet transformation (IMR-wavelet), fail to recover queuing delay signatures from noise. Both IMR-avg and IMR-wavelet yield significant ambiguity and high estimation error in the presence of noise[1]. In this section, we investigate whether BASS denoising algorithm is able to improve the robustness of IMR-Pathload.

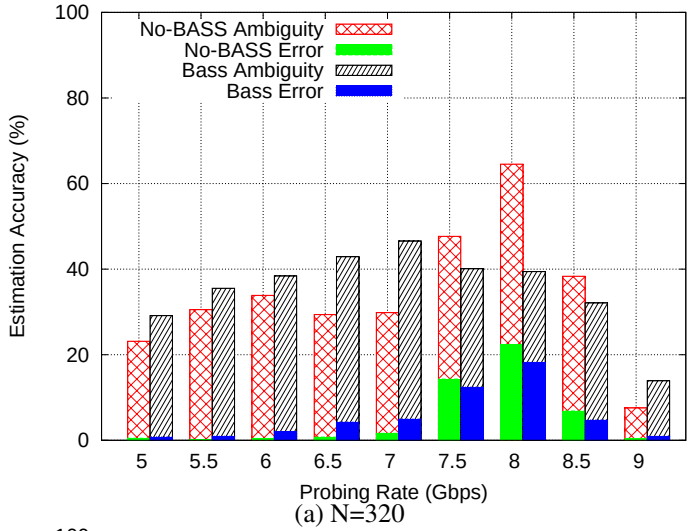
We repeat the experiments in Fig 5.3(a) and Fig 5.3(b), in which single-rate p-streams probing at 9 discrete rates are sharing the 10 Gbps path with BCT traffic. We first apply BASS to these p-streams, and then feed the BASS-denoised streams to IMR-avg and IMR-wavelet for probing-rate evaluation. To ensure that at least 8 data points remain after IMR-avg and IMR-wavelet run, Fig 9.1 and Fig 9.2 depict the resultant ambiguity rate and estimation error both with and without BASS. The vanilla IMR-avg and IMR-wavelet overestimate when the probing rates exceed avail-bw by a large extent (more than 7.5 Gbps). With the aid of BASS, the estimation accuracy is significantly improved—the estimation error rate is reduced from over 30% (the green bars) to below 10% (the blue bars) for both IMR-avg and IMR-wavelet.

Although BASS improves accuracy, both IMR-avg and IMR-wavelet suffer from high rates of ambiguity. For IMR-wavelet with $N = 320$, over 60% p-streams probing at 7 Gbps are incapable of yielding a definite answer on whether or not its probing rate exceeds avail-bw. This is because the Pathload PCT and PDT tests developed on low-speed networks are intrinsically not suitable to handle high-speed noise.

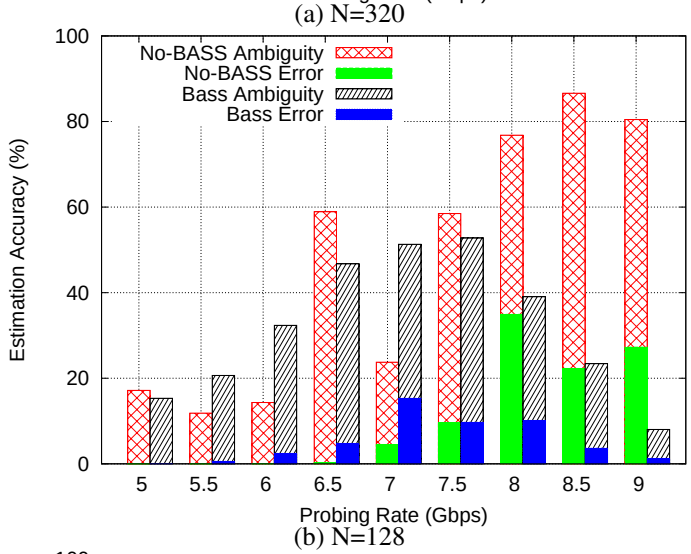
9.2 Applying "Dummy-packet" to TCP Pacing

Legacy TCP transmits a *cwnd* worth of packets out in a burst, leading to self-induced transient queuing. On high-speed paths and in environments with small network buffers, the queuing produced by burstiness increases the risk of packet losses and degrades link utilization. To reduce flow burstiness, TCP pacing aims to uniformly distribute the *cwnd* worth of packets in one RTT by finely controlling the inter-packet gaps. In a

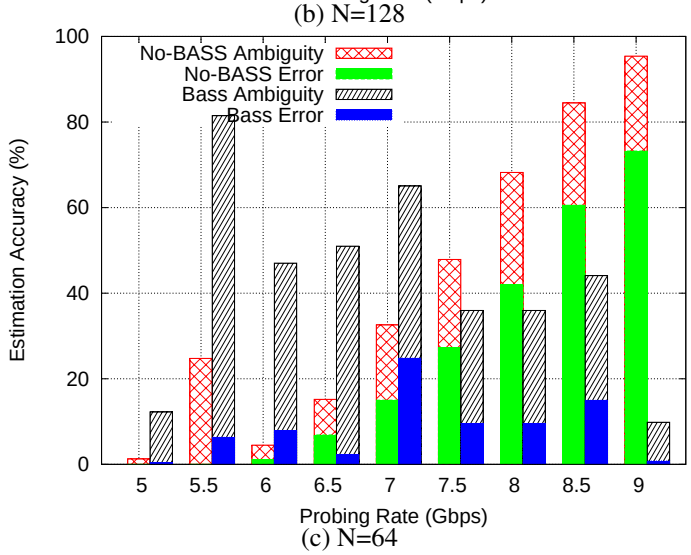
[1]Recall that IMR-Pathload uses single-rate p-streams and adopts the PCT and PDT tests of the vanilla Pathload algorithm. If the results of PCT and PDT tests disagree with each other, the estimation for that p-stream is marked as "ambiguous".



(a) N=320



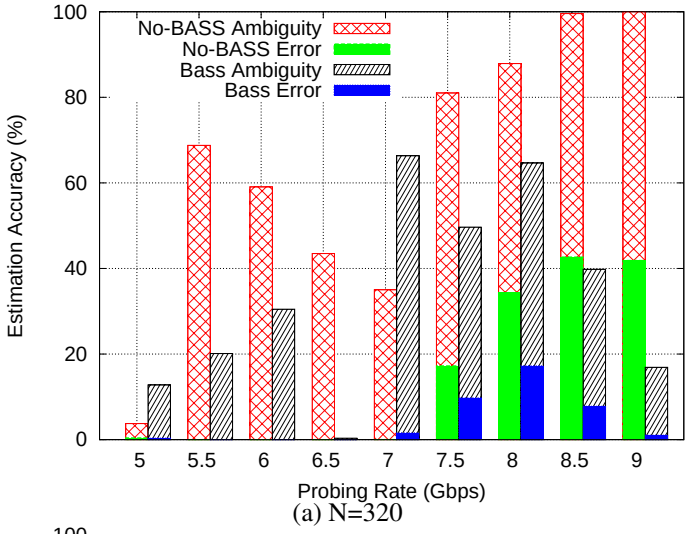
(b) N=128



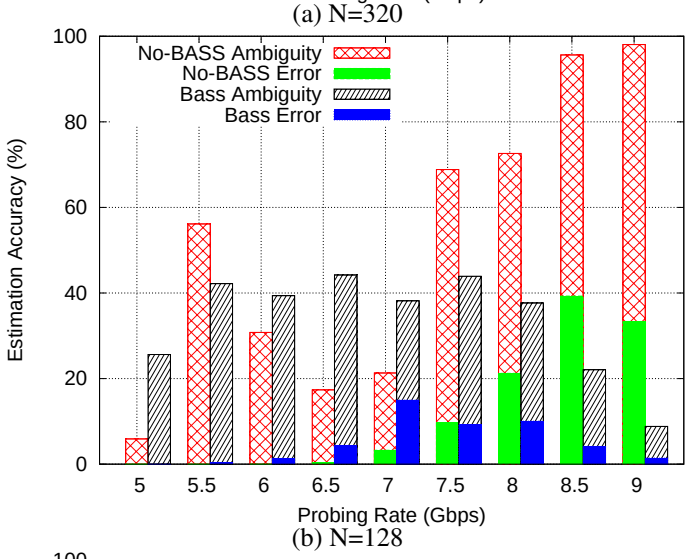
(c) N=64

Figure 9.1: IMR-avg Estimation Error

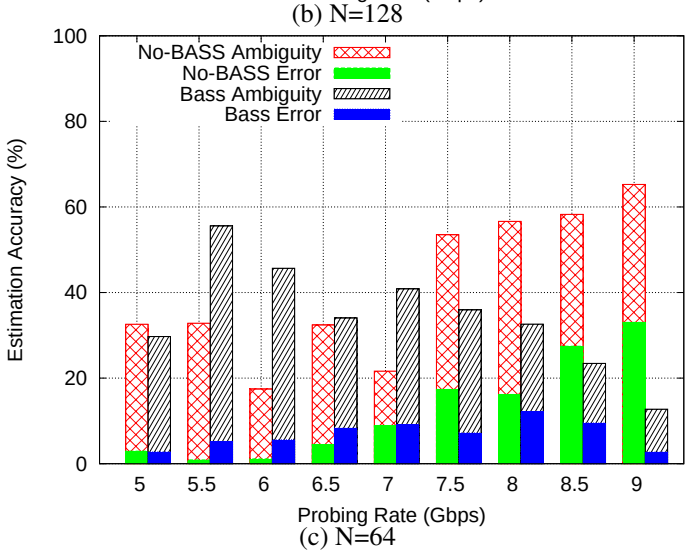
The above plots show the percentage of probe streams (p-streams probing at 9 discrete rates, sharing the 10 Gbps path with BCT traffic) that give wrong estimation errors of the probing rate and that give ambiguous estimations.



(a) N=320

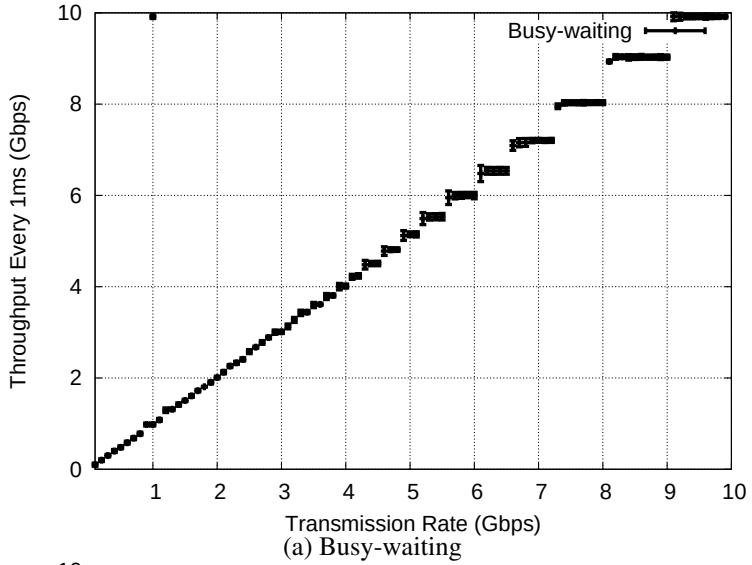


(b) N=128

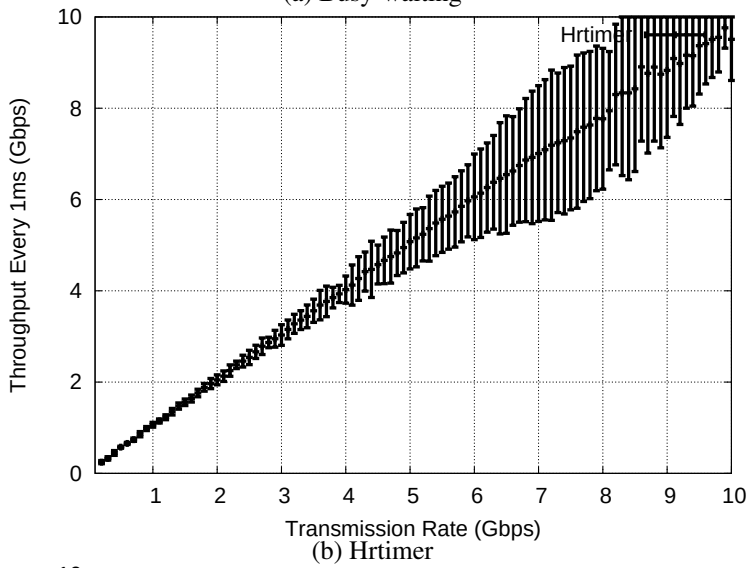


(c) N=64

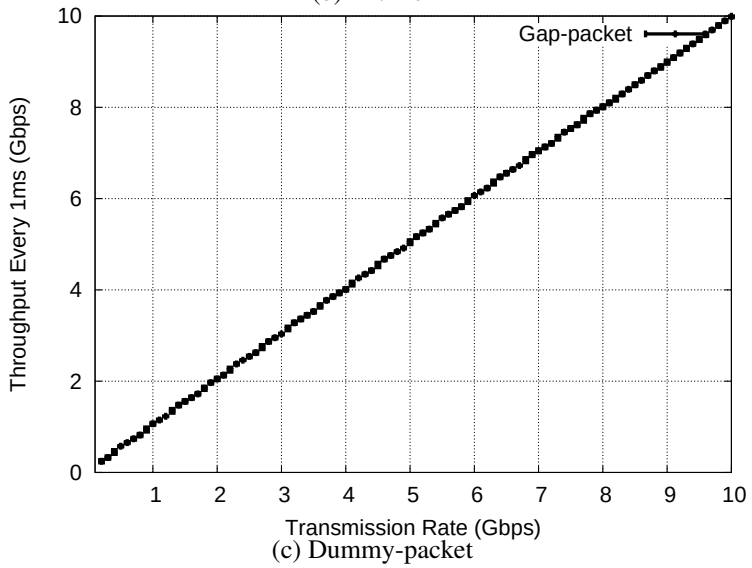
Figure 9.2: IMR-wavelet Estimation Error



(a) Busy-waiting



(b) Hrtimer



(c) Dummy-packet

Figure 9.3: Burstiness of Paced Traffic

The three gap-creation mechanisms are used to pace traffic with targeted rates from 100 Mbps to 10 Gbps, stepped by 100 Mbps. The mean and standard deviation of the actual paced rates at 1 ms intervals are plotted as error bars. The *Dummy-packet* approach reduces traffic burstiness to the best extent.

simulation-based study, this mechanism has been proven to significantly improve fairness and aggregate the throughput of legacy TCP on 10 Gbps paths [62]. Whether the performance gains promised by TCP pacing are achievable in real-world situations depends on whether the fine-scale inter-packet gaps can be precisely controlled.

In this section, we consider gap creation in the context of traffic pacing: the less bursty the traffic, the better the performance gains. We use our "Dummy-packet" mechanism to generate paced traffic at a wide range of rates (from 100 Mbps to 10 Gbps) and examine the smoothness of the resultant traffic at each rate. As the intended inter-packet gaps become finer-grained at higher rates, the smoothness of the paced traffic grows more sensitive to errors in created gaps. The testing covers a wide range of pacing rates, from 100 Mbps to 10 Gbps, and we investigate how well the "Dummy-packet" mechanism that we introduced in Chapter 4 reduces traffic burstiness as rates increase. We also compare the performance of this mechanism with the performance of two other gap-creation mechanisms that attempt to smooth out burstiness: "Busy-waiting," which waits until the gap-time elapses to trigger packet transmission, and "Hrtimer" interrupt, which triggers packet transmission at moments when there is little other traffic flow.

The traffic generated at each of the 100 different rates in Section 4.6.1 is regarded as the paced traffic. Previous research [93] has shown that the 10 ms burstiness control timescale (i.e. controlling the number of outgoing packets every 10 ms) is too coarse-grained to effectively reduce traffic burstiness, but 1 ms is a fine enough granularity to do so. For each rate, we measure the paced throughput that is achieved every 1 ms, and we define the traffic's burstiness at a certain intended transmission rate as the standard deviation of the measured throughput samples every 1 ms. Fig. 9.3 plots the mean throughput at each intended pacing rate as the midpoint of the candlestick and its burstiness as the candlestick. Longer candlesticks indicate a higher degree of burstiness.

As shown in Fig. 9.3(b), with *Hrtimer*, the burstiness of paced traffic increases significantly with transmission rates; when transmitting at a rate higher than 7 Gbps, the throughput may vary by more than 2 Gbps at every 1 ms interval. This is expected, because the timer interrupts grow more frequent at higher rates; this increased interrupt frequency increases system overhead, batches more packets in the softirq context, and leads to a higher degree of burstiness.

The other two mechanisms yield less than 50 Mbps burstiness across all pacing rates. However, in Fig. 9.3(a) the traffic paced using *Busy-waiting* exhibits a step-wise pattern once the transmission rate exceeds 3 Gbps; it fails to distinguish the sub-μs differences in the intended gaps because $gettimeofday()$, which is

used in the application, has insufficient timing resolution at higher speeds. The *Dummy-packet* mechanism achieves extremely smooth traffic at every requested pacing rate, from very low speeds to the ultra-high 10 Gbps.

9.3 Applying Receiver-side High-resolution Timestamping to Delay-based Protocols

TCP Fast is a protocol that solely relies on delay-based congestion control, and it has been demonstrated in simulation to better utilize high-speed network paths while maintaining TCP friendliness [11]. In this section, we first demonstrate that Fast suffers from inaccurate delay measurement on real-world networks. Next, we improve its performance by adopting the receiver-side high-resolution timestamping technique developed in Section 7.2 for RAPID.

9.3 Inaccurate Timestamping Paralyzes Delay-based Protocols

According to previous evaluations in Chapter 8, TCP Fast fails to deliver the optimal performance promised by simulation-based evaluations. As Table 8.5 shows, when sharing the path with non-responsive cross traffic, Fast causes considerably high loss rates in spite of the high throughput achieved. Moreover, in Fig. 8.7, Fast starves web traffic sharing the path and increases the web-flow duration significantly.

The performance mismatch between simulation and our testbed evaluation is due to inaccurate RTT measurements. By default, a Linux protocol stack uses 1 ms low-resolution timestamping, and (as introduced in Section 2.1.3) Fast infers congestion by keeping track of the round-trip delay experienced by each packet. It relies on $RTT = minRTT$ for *cwnd* adaptation, where $minRTT$ is the minimum RTT observed in the previous congestion window. The extent of congestion is inferred from the queuing delay ($RTT - baseRTT$), and *cwnd* is updated based on that inferred delay — it increases more slowly with increasing queuing delay, and decreases if the queuing delay exceeds some threshold.

However, when RTT measurements are logged in kernel for an experiment with $baseRTT = 5ms$, we find that the RTT value is either 5 ms or 6 ms throughout the entire transfer. The $minRTT$ used to update *cwnd* is 5 ms in over 85% cases, and this prohibits Fast from perceiving any congestion. Consequently, Fast enters a repeated pattern of aggressive *cwnd* increments followed by packet losses. It is worth noticing that the same phenomenon happens to Vegas; however, Vegas fails to cause heavy losses because its *cwnd* increment is as sluggish as that of New Reno.

In this section, we equip Fast with the accurate receiver-side timestamping mechanism developed for RAPID, and examine whether this mechanism can improve the performance of this protocol that infers congestion based solely on delay. If so, many other delay-based controls can also benefit from the adoption of our timestamping mechanism.

9.3 Applying Accurate Timestamping to Fast

Using High-resolution Timestamps

We denote the raw Fast implementation used in Table 8.5, which suffers from millisecond low-resolution RTT measurement, as "Fast-vanilla." To improve the resolution, we modify the Fast congestion control module to use the high-resolution timer when timestamping ACK arrival; we denote this Fast version as "Fast-μs".

We repeat the experiments with $RTT = 5ms, 30ms$ in Table 8.5, and list the throughput and loss-rate experienced by Fast flows in Table 9.1. Note that in each experiment, Fast is sharing the path with cross traffic of different levels of burstiness. We also make "Fast-μs" share the path with responsive web traffic used in Section 8.4; we list the throughput of the Fast flow and the median duration of web flows in Table 9.2.

When $RTT = 5ms$, "Fast-μs" significantly reduces loss rates, but yields lower link utilization than "Fast-vanilla" — it fails to fully utilize the path even when there is no cross traffic. This is because "Fast-μs" mistakenly inflates RTTs by including protocol-stack processing delay and reverse-path queuing delay. Consequently, "Fast-μs" overestimates the congestion, leading to under-utilization of avail-bw.

Unlike the short-RTT scenario, when RTT grows to 30 ms, "Fast-μs" is able to yield higher throughput than "Fast-vanilla," because "Fast-μs" reduces packet loss rates drastically from over 0.13% to below 0.013%.

Table 9.1: Fast with Non-responsive Traffic ($RTT = 5ms$)

Fast-μs uses μs-resolution TCP timestamping. "Fast+Qdiscs" applies the Qdisc-based μs-resolution receiver-side timestamping mechanism to Fast.

RTT of Fast flow	RTT=5ms			RTT=30ms		
Cross Traffic	Fast-vanilla	Fast-μs	Fast-Qdiscs	Fast-vanilla	Fast-μs	Fast-Qdisc
No Cross Taffic	9.90Gbp	8.32 Gbps	9.09 Gbps	9.90 Gbps	8.99 Gbps	9.49 Gbps
CBR	6.12 Gbps 0.141%	4.95 Gbps 0.000%	5.94 Gbps 0.001%	3.09 Gbps 0.145%	3.58 Gbps 0.006%	3.66 Gbps 0.006%
SCT	5.98 Gbps 0.132%	4.74 Gbps 0.001%	5.92 Gbps 0.002%	2.42 Gbps 0.232%	2.80 Gbps 0.007%	3.09 Gbps 0.008%
BCT	5.66 Gbps 0.159%	4.69 Gbps 0.000%	5.51 Gbps 0.001%	2.19 Gbps 0.375%	2.52 Gbps 0.013%	2.50 Gbps 0.016%

Qdisc-assisted Delay Measurement

Table 9.2: Fast with Responsive Web Traffic (RTT=5ms)

	RTT=5ms			RTT=30ms		
	Fast-vanilla	Fast-μs	Fast-Qdiscs	Fast-vanilla	Fast-μs	Fast-Qdisc
Throughput	6.29 Gbps	5.23 Gbps	6.01 Gbps	3.03 Gbps	3.81 Gbps	3.96 Gbps
Loss Rate	0.134%	0.001%	0.007%	0.128%	0.013%	0.016%
Median web flow duration	1.63ms	1.39ms	1.54ms	1.95ms	1.74ms	1.75ms

Recall that we developed a set of Qdiscs for OWD measurement in our RAPID implementation, namely E_r, I_r and I_s (see Fig. 7.1). Qdiscs are delay-measurement mechanisms that are free of the impact of end-host processing delay and reverse-path queuing. We tailor this mechanism to achieve high-resolution RTT measurement in Fast.

The essence of Fast congestion control is to infer queuing delay from $minRTT - baseRTT$; to serve the same purpose, we use $minOWD - baseOWD$, where $baseOWD$ is the minimum OWD observed throughout the transfer and $minOWD$ is the minimum OWD in the latest RTT. We use E_s to calculate OWD for each packet by taking the difference between the TSval field and TSecho field in the returning ACK. The OWD measurements are stored in the circular table, as they are with RAPID, and are shared between E_s and Fast congestion control module. Using the following equation, we modify the Fast congestion control module to update $minRTT$ once an entire *cwnd* of packets are acknowledged:

$$minRTT = (minOWD - baseOWD) + baseRTT. \tag{9.1}$$

We denote this Fast implementation "Fast-Qdisc".

We summarize the results of the experiments in Table 9.1 and Table 9.2 using "Fast-Qdisc" as the underlying transport protocol. Its precise RTT measurement allows "Fast-Qdisc" to detect the existence of queuing more agilely than "Fast-vanilla" and perceive the extent of queuing more accurately than "Fast-μs." Therefore, it yields higher throughput than "Fast-μs" and fewer losses than "Fast-vanilla." The Qdisc-assisted mechanism also helps Fast to better utilize the avail-bw left unused by responsive low-speed web traffic and makes it more TCP-friendly than "Fast-vanilla."

9.4 Summary

In this chapter, we demonstrate that the mechanisms designed for TCP RAPID also successfully address the challenges faced by other protocols on 10 Gbps networks.

- We apply the BASS denoising algorithm to IMR-Pathload, showing that BASS improves the robustness for this vulnerable single-rate bandwidth estimation tool on 10 Gbps networks; however, the accuracy of BASS-enhanced Pathload is limited by the deficiency of the Pathload algorithm itself.
- We apply the "Dummy-packet" gap-creation mechanism to TCP pacing, and we show that this mechanism best reduces traffic burstiness, achieving the most accurate pacing rate among state-of-art mechanisms.
- Finally, we incorporate the Qdisc-assisted receiver-side high-resolution timestamping mechanism into the delay-based TCP Fast protocol. We find that Fast benefits from the more accurate RTT measurement enabled by the Qdiscs, yielding higher link utilization and experiencing fewer losses.

CHAPTER 10: CONCLUSIONS

This thesis considers TCP RAPID, a recent proposal for ultra-high speed congestion control, which yields outstanding performance in simulation-based evaluations but faces significant challenges to be put to practice. Rapid adopts a framework of continuous fine-scale bandwidth probing and rate adapting. It requires finely-controlled inter-packet gaps, high-precision timestamping of received packets, and reliance on fine-scale changes in inter-packet gaps. This thesis addresses the challenges RAPID encounters on 10Gbps links for each of these aspects, and translate the promised performance gains in simulation to the real-world.

This thesis surveys several state-of-the-art methods of creating inter-packet gaps using only software. By evaluating them in the 10Gbps testbed, we find that the "Dummy-packet" approach, which inserts proper-sized PAUSE frames to occupy gap-time, significantly outperforms other approaches with superior gap-creation accuracy and non-significant system overhead. The "Dummy-packet" approach is adopted to create inter-packet gaps for RAPID p-streams, and is realized in the Linux implementation using a set of Qdisc packet schedulers at the sending as described.

This thesis deals with the noise-paralyzed bandwidth estimation at ultra-high speeds. In Chapter 5 we develop a novel denoising technique — Buffering-Aware Spike Smoothing (BASS), which detects the boundaries of buffering events and average out observations within each buffering event. BASS is applied it to different bandwidth estimators, and yield more robust estimation at 10Gbps links for both single-rate and mullti-rate p-streams. More importantly, BASS successfully reduces the p-stream length required for robust estimation from 320 packets to merely 64 in each. BASS is adopted to denoise for RAPID bandwidth estimator.

RAPID requires "Gap-clocked" packet transmission and μs-resolution receiver-side timestamping, which are not supported in Linux congestion control interfaces. To fulfill these requirements using common Linux interfaces, we develop a set of Qdisc packet schedulers described in Chapter 7. Chapter 8 evaluates the Linux implementation of TCP RAPIDin the 10Gbps testbed. The result demonstrate that the implementation

successfully tackles several real-world challenges faced by the protocol, and meets the performance bar set by simulation-based evaluations.

Even more fundamentally, the networking community is generally skeptical about the practicality of delay-based and bandwidth-based congestion control due to the similar challenges they face as RAPID does — the latter stretches these challenges to extreme. This thesis applies the mechanisms developed for RAPID to other protocols in Chapter 9 — the "Dummy-packet" gap-creation approach helps TCP Pacing to maximize traffic smoothness, and the receiver-side high-resolution timestamping mechanism developed in Section 7.2 improves TCP Fast performance by providing more accurate delay measurement. This study takes a significant step to convince the community that as long as the challenges can be addressed properly, delay-based and bandwidth-based protocols can live up to their desired performance.

BIBLIOGRAPHY

[1] Sally Floyd. Highspeed tcp for large congestion windows. 2003.

[2] Vishnu Konda et al. Rapid: Shrinking the congestion-control timescale. In *Proc. INFOCOM*. IEEE, 2009.

[3] Tom Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review*, 33(2):83–91, 2003.

[4] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514–2524. IEEE, 2004.

[5] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[6] Ryan King, Richard Baraniuk, and Rudolf Riedi. Tcp-africa: An adaptive and fair rapid increase rule for scalable tcp. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1838–1848. IEEE, 2005.

[7] Andrea Baiocchi, Angelo P Castellani, and Francesco Vacirca. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[8] Sangtae Ha, Long Le, Injong Rhee, and Lisong Xu. Impact of background traffic on performance of high-speed tcp variant protocols. *Computer Networks*, 51(7):1748–1762, 2007.

[9] Lawrence S. Brakmo et al. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 1995.

[10] Carlo Caini and Rosario Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International journal of satellite communications and networking*, 22(5):547–566, 2004.

[11] David X Wei et al. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM ToN*, 14, 2006.

[12] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Tcp nice: A mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 36(SI):329–343, 2002.

[13] Aleksandar Kuzmanovic and Edward W Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1691–1701. IEEE, 2003.

[14] Michele C Weigle, Kevin Jeffay, and F Donelson Smith. Delay-based early congestion detection and adaptation in tcp: impact on web performance. *Computer Communications*, 28(8):837–850, 2005.

[15] Kun Tan Jingmin Song, Q Zhang, and M Sridharan. Compound tcp: A scalable and tcp-friendly congestion control for high-speed networks. *Proceedings of PFLDnet 2006*, 2006.

[16] Kazumi Kaneko, Tomoki Fujikawa, Zhou Su, and Jiro Katto. Tcp-fusion: a hybrid congestion control algorithm for high-speed networks. *Proc. PFLD-net*, 2007.

[17] Shao Liu et al. Tcp-illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65, 2008.

[18] Jingyuan Wang, Jiangtao Wen, Jun Zhang, and Yuxing Han. Tcp-fit: An improved tcp congestion control algorithm and its performance. In *INFOCOM, 2011 Proceedings IEEE*, pages 2894–2902. IEEE, 2011.

[19] Radhika Mittal et al. Timely: Rtt-based congestion control for the datacenter. In *SIGCOMM*. ACM, 2015.

[20] Manish Jain et al. Pathload: A measurement tool for end-to-end available bandwidth. In *Proc. PAM*, 2002.

[21] Ningning Hu and Peter Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE journal on Selected Areas in Communications*, 21(6):879–894, 2003.

[22] Vinay Joseph Ribeiro et al. pathchirp: Efficient available bandwidth estimation for network paths. In *Proc. PAM*, 2003.

[23] Jacob Strauss et al. A measurement study of available bandwidth estimation tools. In *Proc. SIGCOMM*. ACM, 2003.

[24] Thomas E Anderson et al. Pcp: Efficient endpoint congestion control. In *NSDI*, 2006.

[25] Yunhong Gu et al. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 2007.

[26] Mayutan Arumaithurai et al. Nf-tcp: a network friendly tcp variant for background delay-insensitive applications. In *International Conference on Research in Networking*. Springer, 2011.

[27] Bryan Veal, Kang Li, and David Lowenthal. New methods for passive estimation of tcp round-trip times. In *International Workshop on Passive and Active Network Measurement*, pages 121–134. Springer, 2005.

[28] Jim Martin, Arne Nilsson, and Injong Rhee. Delay-based congestion avoidance for tcp. *IEEE/ACM Transactions on Networking (TON)*, 11(3):356–369, 2003.

[29] Saad Biaz and Nitin H Vaidya. Is the round-trip time correlated with the number of packets in flight? In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 273–278. ACM, 2003.

[30] Ravi S Prasad, Manish Jain, and Constantinos Dovrolis. On the effectiveness of delay-based congestion avoidance. In *Proc. PFLDNet*, volume 4, 2004.

[31] Z-L Zhang et al. Small-time scaling behaviors of internet backbone traffic: an empirical study. In *Proc. INFOCOM*. IEEE, 2003.

[32] Qing-jia Lin, Di Chen, and Yun-cai Liu. Non-stationary and small-time scaling behavior of internet traffic. In *2006 International Conference on Communications, Circuits and Systems*, volume 3, pages 1717–1721. IEEE, 2006.

[33] Seong-Ryong Kang et al. Imr-pathload: Robust available bandwidth estimation under end-host interrupt delay. In *Proc. PAM*. Springer, 2008.

[34] Seong-Ryong Kang et al. Characterizing tight-link bandwidth of multi-hop paths using probing response curves. In *Proc. IWQoS*. IEEE, 2010.

[35] Manish Jain et al. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proc. SIGCOMM*. ACM, 2004.

[36] Ryousei Takano et al. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. *Proc. PFLDnet*, 2005.

[37] Statistical information for the caida anonymized internet traces.

[38] Sally Floyd, Andrei Gurtov, and Tom Henderson. The newreno modification to tcp's fast recovery algorithm. 2004.

[39] Douglas Leith and Robert Shorten. H-tcp: Tcp for high-speed and long-distance networks. In *Proceedings of PFLDnet*, volume 2004, 2004.

[40] Saverio Mascolo et al. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proc. MobiCom*. ACM, 2001.

[41] Cheng Peng Fu and Soung C Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

[42] Hadrien Bullot, Roger Les Cottrell, and Richard Hughes-Jones. Evaluation of advanced tcp stacks on fast long-distance production networks. *Journal of Grid Computing*, 1(4):345–359, 2003.

[43] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 75–86. ACM, 2010.

[44] Alok Shriram, Margaret Murray, Young Hyun, Nevil Brownlee, Andre Broido, Marina Fomenkov, et al. Comparison of public end-to-end bandwidth estimation tools on high-speed links. In *International Workshop on Passive and Active Network Measurement*, pages 306–320. Springer, 2005.

[45] Alok Shriram and Jasleen Kaur. Empirical evaluation of techniques for measuring available bandwidth. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2162–2170. IEEE, 2007.

[46] Thiago S Guzella and Walmir M Caminhas. A review of machine learning approaches to spam filtering. *Expert Systems with Applications*, 36(7):10206–10222, 2009.

[47] Christopher M Bishop. Pattern recognition. *Machine Learning*, 128:1–58, 2006.

[48] Wei Huang, Yoshiteru Nakamori, and Shou-Yang Wang. Forecasting stock market movement direction with support vector machine. *Computers & Operations Research*, 32(10):2513–2522, 2005.

[49] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP)*. IEEE, 2010.

[50] T.T. Nguyen el al. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys & Tutorials*, 2008.

[51] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. A machine learning approach to tcp throughput prediction. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 97–108. ACM, 2007.

[52] A.W. Sivaraman et al. An experimental study of the learnability of congestion control. In *SIGCOMM*. ACM, 2014.

[53] P. Geurts et al. A machine learning approach to improve congestion control over wireless computer networks. In *ICDM*. IEEE, 2004.

[54] Overfitting and underfitting. `https://en.wikipedia.org/wiki/Overfitting#Underfitting`.

[55] H. Zou et al. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2005.

[56] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[57] Y. Freund et al. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 1997.

[58] J.H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001.

[59] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. Iperf: The tcp/udp bandwidth measurement tool. *htt p://dast. nlanr. net/Projects*, 2005.

[60] P. Barford et al. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS*, 1998.

[61] Aaron Turner and Matt Bing. Tcpreplay, 2011.

[62] D Wei et al. Tcp pacing revisited. In *INFOCOM*. IEEE, 2006.

[63] Amit Aggarwal et al. Understanding the performance of tcp pacing. In *INFOCOM*. IEEE, 2000.

[64] Linux programmer's mannual: Time(7). `http://man7.org/linux/man-pages/man7/time.7.html`.

[65] Time stamp counter. `https://en.wikipedia.org/wiki/Time_Stamp_Counter`.

[66] netem. `https://wiki.linuxfoundation.org/networking/netem`.

[67] NIC Tuning. `https://fasterdata.es.net/host-tuning/nic-tuning/`.

[68] Hiroyuki Kamezawa et al. Inter-layer coordination for parallel tcp streams on long fat pipe networks. In *Proc. ACM/IEEE conference on Supercomputing*. IEEE, 2004.

[69] Ki Suh Lee et al. Sonic: precise realtime software access and control of wired networks. In *Proc. NSDI*, 2013.

[70] TSO sizing and the FQ scheduler. `https://lwn.net/Articles/564978/`.

[71] Mirja Kühlewind et al. Chirping for congestion control-implementation feasibility. *Proc. PFLDNeT*, 2010.

[72] Ryousei Takano et al. High-resolution timer-based packet pacing mechanism on the linux operating system. *IEICE transactions on communications*, 2011.

[73] Keon Jang et al. Silo: predictable message latency in the cloud. *ACM SIGCOMM Computer Communication Review*, 2015.

[74] Zhong-Zhen Wu and Han-Chiang Chen. Design and implementation of tcp/ip offload engine system over gigabit ethernet. In *Proceedings of 15th International Conference on Computer Communications and Networks*, pages 245–250. IEEE, 2006.

[75] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127–138, 2012.

[76] Moving average. `https://en.wikipedia.org/wiki/Moving_average`.

[77] Rami Cohen. Signal denoising using wavelets. 2012.

[78] Daubechies wavelet. `https://en.wikipedia.org/wiki/Daubechies_wavelet`.

[79] Constantinos Dovrolis Parameswaran Ramanathan and David Moore. What do packet dispersion techniques measure? Citeseer.

[80] Thuy TT Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4):56–76, 2008.

[81] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[82] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 479–490. ACM, 2014.

[83] T.G. Dietterich. Machine-learning research. 1997.

[84] A. Liaw et al. Classification and regression by randomforest. *R news*, 2002.

[85] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[86] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. 1990.

[87] A.J. Wyner. On boosting and the exponential loss. In *the 9th Annual Conference on AI and Statistics Jan*, 2003.

[88] F. Pedrogosa et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 2011.

[89] T. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning*, 2000.

[90] F.donelson smith. Private Communication.

[91] Rebecca Lovewell, Qianwen Yin, Tianrong Zhang, Jasleen Kaur, and Frank Donelson Smith. Packet-scale congestion control paradigm. *IEEE/ACM Transactions on Networking*, 2016.

[92] Rebecca Lovewell et al. Impact of cross traffic burstiness on the packet-scale paradigm. In *LANMAN*. IEEE, 2011.

[93] Hao Jiang and Constantinos Dovrolis. Why is the internet traffic bursty in short time scales? In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 241–252. ACM, 2005.