TCP Rapid: From Theory to Practice

Qianwen Yin, Jasleen Kaur, Don Smith

University of North Carolina at Chapel Hill
{qianwen,jasleen,smithfd}@cs.unc.edu

***Abstract*—Delay and rate-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/or required changes to the standard TCP stack/headers. With the advent of high-speed networking, some of these concerns become even more significant.**

In this paper, we consider 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, which requires a potentially different and finely-controlled gap for every packet, 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.

We design a Linux implementation of Rapid after carefully considering each of these challenges. Our 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.

I. Introduction

Delay and rate-based alternatives to TCP congestion-control have seen a significant surge in interest [1], [2], [3], [4], [5], [6], [7], [8]—indeed, their performance gains (observed both in simulations and simple testbeds) seem quite promising, often exceeding TCP performance by several orders of magnitude. However, such designs have also faced significant reluctance in being adopted/deployed on a wide-scale across the Internet—this is primarily because the promised gains observed under controlled and simulated settings are not trusted to translate well to real-world settings.

Why? Firstly, because most of the alternatives rely on measurement of metrics such as end-to-end delay or available bandwidth as a measure of congestion, rather than rely on packet loss—these metrics can be fairly volatile, and their measurement can be quite prone to fine-scale buffering noise [9]. This is especially true in high-speed network environments. Secondly, several protocols rely on fine-scale pacing of inter-packet gaps (IPG), which are challenging to control predictably in interrupt-driven operating systems, especially at high speeds. Thirdly, stepping away from a conventional congestion-control framework that has been used and perfected for more than three decades, is resistance-worthy—why would an operator of production servers trust a new prototype? Given the promise of such congestion-control alternatives, especially in high-speed networks, it is important to address these challenges and open the real-world to their adoption.

Of all the proposed alternatives to TCP congestion-control, RAPID [7] perhaps stretches the above challenges to the greatest extent. In simulations, this protocol shows outstanding gains in terms of scalability, adaptability, TCP-friendliness, and fairness. However, as described in Sections II-III, RAPID needs to create μs-precision inter-packet gaps for *all* data packets sent out. Further, it relies on observing fine-scale changes in inter-packet gaps for continuously estimating end-to-end available bandwidth, which is fairly sensitive to the presence of fine-scale buffering noise. Finally, RAPID relies on a "gap-clocked" transmission of data packets, which is a significant departure from the conventional "ack-clocked" TCP framework. In this paper, we ask (in the context of TCP RAPID): *can these challenges be addressed in order to realize ultra-high speed real-world prototypes that perform as well as the promise delivered by simulations?* If the answer is a yes for a protocol as demanding as RAPID, then this would be a significant enabler for the practical adoption of delay, rate, and bandwidth-based protocol research.

Our Innovations This paper presents the following innovations:

- We tailor the state-of-the-art for creating inter-packet gaps in the Linux kernel, and show that it achieves μs accuracy at ultra-high speeds.[1]
- We adapt the state-of-the-art for alleviating the impact of fine-scale noise in inter-packet gaps, on RAPID bandwidth estimation.
- We propose and evaluate the decoupling of the probing and adapting timescales used in RAPID congestion control, for alleviating the tradeoff between responsiveness and stability in the presence of volatile available bandwidth.
- We implement all the component of RAPID as pluggable kernel modules, which can be loaded/unloaded on the fly, without bringing down production servers. These work with standard TCP headers and the socket API.
- We evaluate the implementation design on 10/40Gbps testbeds in the presence of representative and bursty cross-traffic, and show that it lives up to the simulation-promised performance. Furthermore, we show that this performance can not be achieved without any of the above innovations.

In the rest of this paper, we summarize RAPID in Section II and identify challenges in Section III. We present our design of a Linux implementation in Section IV and our evaluations in Sections V-VI. We conclude in Section VII.

[1] In this paper, "ultra-high speed" refers to 10 Gbps and more.

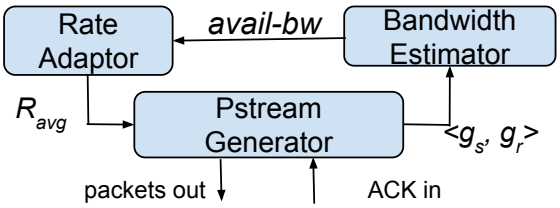


Fig. 1: TCP RAPID Architecture

II. Background – TCP RAPID

Instead of simply relying on packet loss as congestion feedback, RAPID continuously estimates the end-to-end avail-bw, and uses the estimates to send packets out in logical groups referred to as p-streams. The transmission of p-streams is rate-based (and not ack-clocked), and is managed by three components operating in a closed loop (Fig 1). Given an average send rate R_{avg} informed by the rate adaptor, 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 adaptor updates the sending rate for the next p-stream.

Pstream Generator RAPID uses each data packet sent for probing the end-to-end path for 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-spaced rates (with N_p packets sent at each rate): $R_i = R_{i-1} \times s, i \in [2, N_r], s > 1$. In Fig 2a, $N_r = 4$, and $N_p = 16$. The average send rate of the full p-stream is set to R_{avg}, informed by the rate adaptor.

Bandwidth Estimator The RAPID receiver records the arrival times of data packets in the ACKs it generates on receiving them. Once the sender receives all ACKs for a p-stream, it extracts these times, computes the receive gaps (g_r), and feeds them to the bandwidth estimator. The send and receive gaps are used to compute an estimate for the end-to-end avail-bw ($ABest$), based on the principle of self-induced congestion—for the i-th packet in a pstream, $g_r^i > g_s^i$ indicates that it experienced queuing at the bottleneck (respective probing rate was higher than the avail-bw). $ABest$ is computed as the largest probing rate *beyond* which packets *consistently* experience bottleneck queuing (the second probing rate in Fig 2a). Due to burstiness, simple smoothing heuristics are used in the computation, and the reader is referred to [7] for details.

Rate Adaptor 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:

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

wehre l is the duration of the just-acked p-stream , and tau and eta 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.

Note that while R_{avg} closely follows the end-to-end avail-bw (and helps ensure that p-streams are not sent at a rate higher than the network can currently handle), at smaller timescales, the exponentially-spaced p-streams simultaneously probe for rates both higher and lower than R_{avg}—this gives the protocol excellent agility in the presence of dynamic cross-traffic. In fact, simulation-based evaluations in [7] show that the protocol has close-to-optimal performance along several dimensions, most notable of which are: (i) discovering and adapting quickly to changes in avail-bw (due to continuous probing at sub-pstream timescales); (ii) negligible impact on co-existing TCP traffic (due to an extremely low queuing footprint); and (iii) RTT-fairness, with no bias against long-RTT transfers (due to shedding of ack-clocking).

III. TCP RAPID: Challenges in Practice

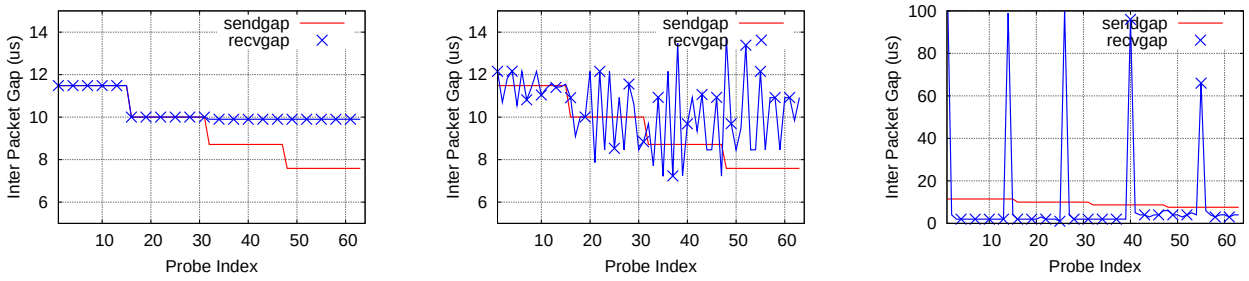
All of the performance gains reported in [7] have been observed solely in the NS-2 simulator environment. Three types of challenges can be identified in realizing the same in the real world.

A. Fine-scale Inter-packet Gap Creation

Challenge RAPID requires the TCP sender to send packets out with *high-precision* and *fine-grained* inter-packet spacing for the purpose of bandwidth estimation. For instance, in order to probe for an avail-bw of 10-40 Gbps with even jumbo-sized frames, packets have to be sent with spacing as small as a few microseconds—and this value reduces proportionally as we consider higher network speeds. A spacing inaccuracy of even 1-2μs, can lead to a bandwidth-estimation error of 50%! Several other protocol proposals rely on fine-scale IPG creation and face a similar challenge in scaling up to ultra-high speeds [10], [11], [3]—the confounding aspect of RAPID is that it uses *every* packet for fine-scale probing, whereas these others rely on bandwidth probing only intermittently.

Sate of the art Existing bandwidth estimators [12], [13], [14] and transport protocols [10], [11] create gaps for bandwidth probing, by staying in a busy-wait loop (often in user space) until the desired time gap elapses. Unfortunately, ensuring the fine-grained and high-precision inter-packet gaps needed for scaling up to ultra-high speeds, can be fairly challenging in current software-based end-systems, mainly for two reasons. Firstly, most operating systems are interrupt-driven, and the process sending out packets of a p-stream may lose control of the CPU at any time while "waiting" for the required time-gap between two consecutive packets. The resultant send gaps are unpredictable, and lack high precision.

Secondly, before those packets get transmitted by the NIC, they can get buffered at several places—in the protocol layer buffers as they are being handed down the kernel protocol stack, at the NIC interface queue when the kernel directs them to the corresponding NIC, and on the NIC outgoing queue. Such buffering can completely destroy the intended inter-packet gaps. Some of the upper-layer buffering can be avoided by using in-kernel support that relies on software timers (e.g., **qdisc_watchdog_timer** used in Linux FQ scheduler and tasklet used in TRC-TCP [15]). However, such interrupt-driven



(a) Noise free **(b)** After bottleneck **(c)** At receiver

Fig. 2: Probe streams $N_r = 4, N_p = 16$

transmissions at highspeed will lead to increased overhead of context switch and to slow down the system when multiple high-speed flows coexist [16]. Besides, it can not prevent packet buffering at the sending host.

Goal Our first objective in this paper is to address this challenge and: *consider and evaluate high-speed techniques that enable fine-grained gap creation with high precision.*

B. Noise Removal from Receive Gaps

Challenge Bandwidth estimation is key to RAPID, which relies on the assumption that any fine-scale changes in inter-packet gaps are indicative of the bottleneck avail-bw, and can be used to robustly estimate it. However, even when p-streams are sent out with accurate spacing, there are two types of noise sources that can challenge this assumption:

- *Burstiness in cross-traffic at bottleneck resources:* In a packet-switched network, traffic arrival can be fairly bursty at short timescales [17]. As illustrated in Fig 2b, frequent arrival of short-scale traffic bursts introduces noise in the persistent queuing signature of Fig 2a, and can lead to low estimates of avail-bw.
- *Transient queuing at non-bottleneck resources:* Even a non-bottleneck resource can induce short-scale transient queues when it is temporarily unavailable while 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* can force packets to wait at the NIC before being handed to the OS for timestamping, even if the CPU is available — this can introduce noise worth *hundreds* of microseconds in inter-packet gaps [18]. Fig 2c plots the receive gaps observed when packets are delivered by the receiver NIC to the operating system — the inter-packet gap signatures are unrecognizable after interrupt coalescence.

It is important to note that such noise impacts not only other protocols that rely on bandwidth estimation, but also a myriad of protocols that rely on delay-measurements, with the performance of the former being more sensitive to noise [19], [10], [3], [20].

State of the art While heuristics for smoothing out noise have been designed for bandwidth estimators (e.g., [21]), most need fairly long p-streams in order to scale to 10 Gbps and beyond [18] — the longer the p-streams, the less are the performance gains of a protocol like RAPID. The recently-proposed denoising technique, BASS [18], aims for shorter p-streams—it can help estimate bandwidth on 10 Gbps paths with less than 10-15% error, using 96-packet p-streams[2]. While BASS is a promising technique, it has been mostly evaluated for bandwidth estimators — p-streams are sent far apart and assumed to be independent, and the average rate of each pstream is not influenced by the avail-bw. Both of these aspects do not hold within a congestion-control protocol like RAPID.

Goal Our second objective in this paper is to: *consider and evaluate such highspeed techniques for reducing the impact of fine-scale noise in inter-packet gaps.*

C. Alleviating the Stability/Adaptability Trade-off

Challenge Noise is a particularly significant concern for delay and bandwidth-based transport protocols due to the finer time scales at which their respective congestion metrics are probed for. The larger the timescales at which these protocols choose to probe the network path, the smoother (and less impacted by small-scale noise) their measurements will be. However, the resulting protocol will be less quick in responding to changes in network conditions. This tradeoff between stability and adaptability should be carefully balanced.

RAPID uses the same timescale (given by the length of a p-stream) for *probing* for end-to-end avail-bw, as well as for *adapting* the data sending rate to changes in avail-bw. Longer p-streams allow the protocol to react to avail-bw changes only at timescales at which the avail-bw is stabler and less noisy, however, shorter p-streams allow the protocol to sample avail-bw more frequently and track it more closely.

Goal Our third goal is to: *alleviate the tradeoff between stability and adaptability by tracking changes in avail-bw closely, while only adapting to it at stabler timescales.*

D. Deployability Within the TCP Stack

Challenge TCP is the dominant transport protocol used by most applications. In order to achieve widespread impact and allow applications and network edge devices to work seamlessly, RAPID should be implemented such that: (i) it works with existing TCP protocol headers and the socket API; and (ii) it is a pluggable module within widely-deployed TCP stacks— in the context of servers, this caters to the requirement

[2]In [22], a machine-learning approach is used to denoise inter-packet gaps—however, no kernel-friendly implementation exists.

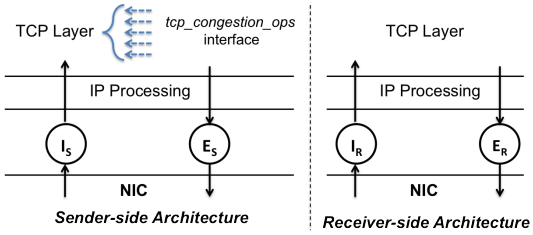


Fig. 3: Architecture of RAPID Implementation

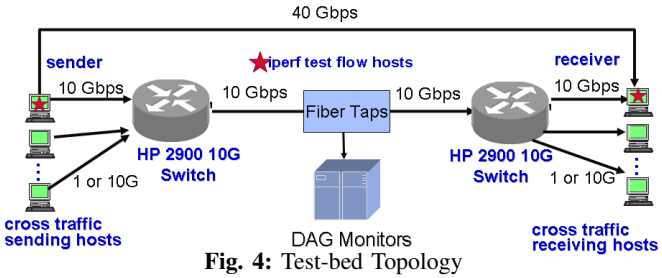


Fig. 4: Test-bed Topology

of system administrators that the protocol implementation can be loaded (or unloaded) on the fly without bringing down a production server.

In widely-deployed TCP stacks, sending of data packets is ACK-clocked and window-controlled—pluggable congestion-control modules are supported in operating systems like Linux, that allow changes to the amount by which window growth occurs when ACKs are received (how many packets are eligible to be sent out). However, these do not allow changes to *when* a packet gets sent out. In contract, the sending of packets in RAPID is "gap-clocked", in which elapsed time (and not ACK arrival) determines when the next packet will get sent out. In fact, the receiving of ACKs and sending of data packets are completely asynchronous of each other. Clearly, supporting gap-clocking needs modules beyond the TCP congestion-control framework.[3]

In addition, the RAPID receiver needs to observe in high-precision, and communicate back to the sender, the gaps between packets it receives. Delay-based protocols TCP-LP [8] and LEDBAT [23] rely on TCP timestamping option for computing one-way delays. However, timestamping with micro-second precision, which is needed for high-speed networking, is not available. Furthermore, timestamps are produced when ACKs are generated, and not when the respective data segments are received (subjecting gaps to variable ACK processing times). Fig 5 plots the time difference between the actual arrival of a data packet and the time recorded in TCP timestamps (as observed on our 10 Gbps testbed)—we find that the two can differ by up to 60μs.

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

IV. Our Implementation Design

We present our design of a RAPID implementation for addressing the challenges identified above.

A. Realizing Gap-clocking in a Standard TCP Stack

Current Linux kernel provides a congestion handler interface, **tcp_congestion_ops**, which allows different congestion control algorithms to be implemented in a pluggable manner. Implementing RAPID boils down to: *removing the dependency of packet transmissions on ACK arrival* and *scheduling these based on intended IPGs*. We elaborate how both of these can be achieved using the above interface and loadable modules.

[3]Other transport protocols that rely on bandwidth estimation, even intermittently, require gap-clocking as well.

1) Removing ack-clocking: In RAPID, steady-state packet transmissions in large transfers are not triggered by the arrival of ACKs. ACK-clocking can be turned off with relative ease by simply using the tcp_congestion_ops interface to fix *cwnd* to a value much larger than the bandwidth delay product [4]. 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.

2) Incorporating gap-clocking: The tcp_congestion_ops interface does not allow control over *when* a segment is sent by TCP. For scheduling packet transmission, we instead create a new Linux Qdisc module (E_s in Fig 3) which is attached to a given network interface. Link-layer frames containing TCP segments are processed by E_s before being delivered to the NIC device driver. E_s responds to two calls—**enqueue** and **dequeue**. It maintains a FIFO queue for each TCP connection for buffering packets received from **enqueue**. Upon each **dequeue**, it chooses the next packet that will be transmitted.

E_s is responsible for enforcing IPGs within each TCP connection. It (i) groups packets into units of p-streams; (ii) computes per-packet gaps according to the average sending rate R_{avg} (using a data structure shared with the TCP layer); (iii) assigns each packet an intended transmission time (t_s) according to the computed gaps; (iv) schedules the departure of the head of queue at its appointed t_s (mechanisms discussed below).

3) Interleaving packets from multiple flows: To schedule packets, in the order of their t_s, from multiple RAPID connections that use the same NIC, we maintain a minimum heap data structure—the elements of the heap are the packets at the *head* (earliest t_s) of each per-connection FIFO queue. The **dequeue** function in E_s removes and sends the top packet of the heap to the NIC **only if** its intended t_s time has passed.

With the presence of multiple RAPID flows, it may not be possible to respect t_s for every packet of every flow (for instance, when the transmission time of two packets from different flows are nearly the same). It is important to realize that the resultant short-scale queuing is expected by avail-bw estimators. Indeed, the sender outbound link represents the first shared link for those flows—the intended send gaps control only the times at which the packets within each flow arrive at the NIC, and not depart the NIC.

B. Creating Accurate Inter-packet Gaps

Several research projects have relied on hardware support for fine-scale control of inter-packet gaps—for instance, the

[4]In our 10Gbps experiments, we set *cwnd* to 16000

Comet-TCP [15] protocol stack is fully implemented on a programmable NIC to create gaps with subnano precision. However, the requirement of such specialized hardware seriously limits the deployability of a new transport protocol, which is one of our prime goals.

[24] employs a novel approach for fine-scale control of inter-packet gaps—it inserts appropriately-sized Ethernet PAUSE frames to occupy the desired gap between two TCP data packets. These special control fames are specified as part of the IEEE 802.3x for flow control between two ends of a link. They are discarded by a receiving NIC and thus consume bandwidth only on the first link (typically sender NIC to the first switch) on the path. As a result, the intended gaps are preserved between successive TCP packets arriving at the first outbound queue. [24] uses this Ethernet feature for implementing paced TCP (constant gaps within a given flow).

Inspired by the above approach, we design E_s to send PAUSE frames and data packets in an interleaved back-to-back manner to the outgoing NIC, to be transmitted at line speed. For timekeeping and fine-scale gap-control, E_s relies on a $link\_clock$ (instead of the kernel clock), which tracks the transmission time that would be consumed by all outbound packets that have been sent so far to the NIC. The intended send time t_s assigned to each packet, is also compared to the $link\_clock$ (and not the kernel clock). Once **dequeue** is called, E_s checks whether the send time of the packets at the top of our heap is less than or equal to the current $link\_clock$. If true, that packet is dequeued and sent to the NIC; otherwise, E_s creates and sends a PAUSE frame of size $(t_s - link\_clock) \times C$, where C is the link capacity, and t_s corresponds to the packet at the top of the heap. $link\_clock$ is incremented by the expected transmission time of the frame being sent to the NIC ($\frac{framesize}{C}$). Creating a PAUSE frame of the above size ensures that the next time **dequeue** is called, the packet at the top of the heap would be eligible to be sent, and would have the desired gap from the previous data packet. Thus, inter-packet gaps are achieved by finely controlling the size of PAUSE frames.

We evaluate the accuracy of gap-creation using E_s by generating a large number of p-streams, covering a wide range of probing rates from 100Kbps to 10Gbps. The actual gaps are recorded by collecting traces using the DAG monitor in Fig 4, immediately after packets traverse the 1st switch. For comparison, we generate gaps with a user-level application modified from pathChirp [12]. We also implement a Qdisc that enforces t_s by registering a software timer (with qdisc_watchdog interface) for every packet departure. Fig 6 plots the distribution of the difference between actual gaps and intended ones—E_s limits the error within 1μs, which is significantly more accurate than using software interrupts or user-level application!

C. Timestamping Packet Arrivals

In order to record inter-packet receive gaps with μs precision and accuracy, and communicate back to the sender using standard header timestamp options, we create two Qdiscs

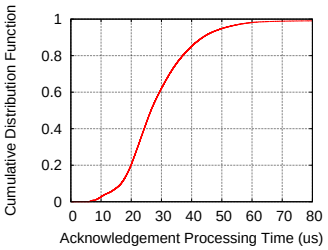


Fig. 5: ACK Processing Time

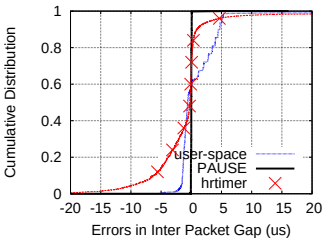


Fig. 6: IPG Creation Error

E_R and I_R at the receiver, and one I_S ingress Qdisc at the sender (Fig 3). 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—these are recorded in a table shared with E_R. Once an ACK is generated and sent by TCP 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.

When the ACK segment reaches the sender, the ingress Qdisc I_S saves the μs timestamp. To ensure correct TCP processing (which expects monotonically increasing millisecond timestamps), I_S restores TSval field with the local millisecond timestamp before handing the packet for upper-layer protocol processing. The saved μs timestamp is shared with E_S and the TCP layer — it is used by the bandwidth estimator implemented with **tcp_congestion_ops**, for computing $ABest$ once a p-stream is completely ACKed.

D. Denoising for Bandwidth Estimation

The recently-proposed Buffering-aware Spike Smoothing (BASS) technique has been shown to work well in denoising receive gaps within short multi-rate p-streams [18]. Below, we briefly summarize the technique and then evaluate it in the context of RAPID.

1) Buffering-Aware Spike Smoothing: BASS is based on the observation that even though buffering events like interrupt coalescence can completely destroy gaps for *individual* packet within p-streams (Fig 2c), the *average* receive gap within a single *complete* buffering event can still be recovered. BASS recovers this quantity by first carefully identifying boundaries of buffering events by analyzing receive gaps, g_r—each "spike-up" and following dips in Fig 2c correspond to packets within the same buffering event (packets queued in the receiver NIC before generation of the next interrupt). BASS looks for sudden changes in receive gaps to detect these. After estimating buffering event boundaries, within each event BASS replaces both g_s and g_r with their respective averages. Such "spike-removal" is repeated up to three times until a robust signature of persistent queuing delay is restored in the smoothed pstream. Fig 7 plots the BASS-smoothed gaps for the pstream in Fig 2c—the spikes are successfully eliminated. The smoothed gaps for the p-stream are then fed into the bandwidth estimator.

2) Re-evaluating BASS: [18] evaluates BASS for several different settings of p-stream length —it shows that BASS can estimate bandwidth with less than 10-15% error, using p-streams with just 96 packets. For several reasons, however, it is necessary to re-evaluate BASS in the context of RAPID:

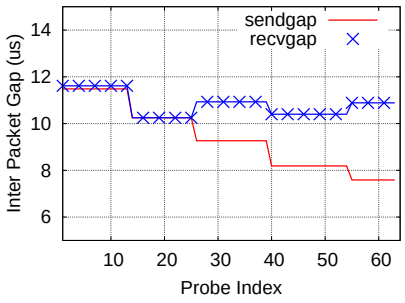


Fig. 7: BASS-denoised Gaps

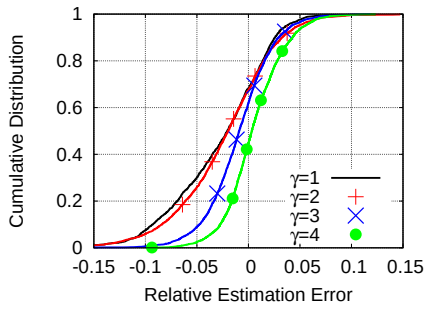


Fig. 8: Decoupling Probing/Adapting Timescales

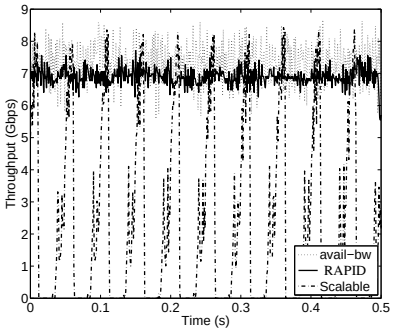


Fig. 9: Throughput with BCT

(i) Unlike RAPID , [18] introduces a significant gap between p-streams to ensure independence. RAPID, however, sends p-streams back-to-back—any queue buildup on the path caused by the previous p-stream may not drain out before the next one arrives. **(ii)** Unlike [18], in which p-streams were generated with pre-determined and controlled average send rates, the R_{avg} for a p-stream depends on the $ABest$ estimated by a recent one. In [18], in fact, p-streams were filtered out if the actual avail-bw did not fall within the range of their probing rates. **(iii)** Unlike [18], standard TCP implementations rely on delayed ACKs, and consequently, only every other packet gets a timestamp.

We incorporate BASS within the RAPID bandwidth estimator, to process receive gaps derived from ACK timestamps, before computing bandwidth estimates (in the presence of delayed ACKs and p-stream-generation by the RAPID control loop). We measure the bandwidth estimation accuracy of our implementation (using the methodology in Section V) with several different choices of p-stream length, including 32, 48, 64, 96. We find that $N = 64$ achieves the best bandwidth estimation accuracy (errors less than 10% for over 80% p-streams).

E. Alleviating the Stability/Adaptability Trade-off

The stability/adaptability tradeoff discussed in Section III-C is controlled by a single parameter—the p-stream length (which represents both the probing and adapting timescale). In order to alleviate this tradeoff, we propose to decouple the probing and adapting timescales of RAPID. We achieve this by not requiring the sender to update R_{avg} upon each $ABest$ computation, but rather do it at a lower frequency (γ). Specifically, we adapt the R_{avg} of the transfer only once every γ p-streams, and set it to the mean of all γ ABest collected since the last update.

Such decoupling naturally leads to a question: if we fix the rate-adapting timescale ($N \times \gamma$), do we get more accurate bandwidth estimates using longer p-streams (large N, $\gamma = 1$), or by using the mean bandwidth estimate of several smaller p-streams (small N, $\gamma > 1$)? To study this, we fix the rate-adapting timescale at 192 packets, and vary the probing timescale per N=48,64,96,192 ($\gamma = 4, 3, 2, 1$, respectively). Fig 8 plots the relative bandwidth estimation error observed across several p-streams sent over our 10 Gbps testbed. We find that using shorter p-streams (but the same rate-adapting timescale) reduces the estimation errors from 15% to 5%! However, when p-streams are shorter than 64 packets, the estimation errors do not reduce with larger γ—this is in agreement with our previous observation of the performance with $N = 64$. In the remaining evaluations, we adopt $N = 64$ with the rate-adapting timescale of 192 packets ($\gamma = 3$).

V. Evaluation of RAPID Implementation

In this section, we experimentally study how close our implementation gets to achieving the simulation-based performance reported in [7]. The key performance gains reported in [7] for RAPID were in terms of scaling to high-speed throughput, adapting quickly to changes in avail-bw, co-existing peacefully with low-speed TCP traffic, and inter-protocol fairness. We attempt to recreate similar experimental settings in our testbed, with some key differences: (i) We focus on contemporary ultra high-speed paths of 10/40Gbps capacity, while [7] focused mostly on 1Gbps paths, (ii) we use a bursty, representative traffic aggregate as cross-traffic for studying adaptability of RAPID, while [7] used a simple synthetic cross-traffic stream, (iii) we use shallow-buffered switches, while [7] provisions much larger buffers.

We also evaluate the Linux implementations of several protocols for comparison—New Reno, Bic, Cubic, Scalable, Highspeed, Hybla, Illinois, Vegas, Westwood, LP, Yeah and Fast.[5] For space constraints, we only present the results of Cubic and Scalable in this paper—the former is the default congestion control in Linux, the later consistently gives best link utilization. The remaining protocols are included in our extended report [25].

Unless specified otherwise, we use the following settings for RAPID transfers: $\tau = 10ms, \eta = 3$, and $RTT = 30ms$ (representative of the medium US continental RTT).

A. Testbed Topology

The dumbbell testbed of Fig 4 consists of two HP 2900 switches with multiple 1Gbps and 10Gbps ports. The 10Gbps switch-to-switch path is used to connect two pairs of 10Gbps TCP senders and receivers. These hosts are Dell PowerEdge R720 servers with four cores on 8 logical processors running

[5]Fast implementation 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. Compound is no longer supported in recent Linux kernels.

at 3.3GHz. The 10Gbps adaptors on the two sending hosts are PCI Express x8 MyriCom NICs, on the two receiving hosts are PCIe Intel 82599ES NICs. The other 10 pairs of hosts with 1Gbps NICs are used to generate cross traffic sharing the switch-to-switch link. The testbed also includes a 40Gbps direct fiber-attachment over QSPF+ ports (not a switched path) between one pair of the Dell servers. The 40Gbps adapters are PCIe x8 Mellanox NICs. All hosts in the testbed run the latest RedHat Linux 6 with 2.6.32 kernel.

For emulating path RTTs and loss properties for RAPID transfers, we use extended versions of our Qdiscs I_R and E_R on the receiver. The extended I_R drops packets randomly according to the required loss rate; E_R delays the transmission of ACKs to emulate RTT latency. For other TCP variants, we use *netem* to randomly drop packets at the sender, and to delay ACKs for RTT emulation at the receiver.

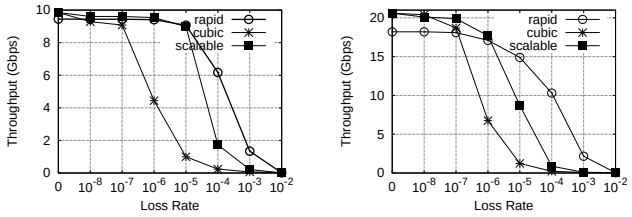
Limitations One limitation of using a real switch on our testbed is that we are unable to log or finely monitor the bottleneck queue size—instead, we must rely on indirect measures, such as packet losses and their impact on throughput. Second, our switches are fairly shallow-buffered—this implies that in all of our evaluations the bottleneck buffers are much smaller than the bandwidth-delay product (shallow buffers have been recommended widely for ultra-high speed networks [26]). Finally, the CPUs on our Dell servers are unable to keep up with 40Gbps throughput, and reach 100% utilization when the transfer rates reach 20Gbps—this is the maximum achievable throughput on the 40Gbps path.

1) Cross Traffic: We evaluate the RAPID implementation against two types of cross-traffic—responsive traffic from emulation of web users, and replayed traces of traffic aggregates (with different levels of burstiness).

Responsive Web Traffic In order to generate bursty traffic loads on the switch-to-switch bottleneck, we use 10 pairs of hosts, each emulating thousands of web users by running a locally-modified version of the SURGE [27]—they establish "live" TCP connections with a diverse set of RTTs and inter-arrival times, and produce representative and responsive HTTP traffic. The average throughput of such traffic is 2.42Gbps. We record the flow completion times for each TCP connection.

Replayed Web Traffic Aggregates We also generate bursty, but non-responsive cross-traffic—this helps ensure repeatability and burstiness control across experiments. For this, we record a packet trace for each SURGE data source from the responsive web traffic generation above. We then replay the trace using *tcpreplay*. The average rate of the aggregated replayed traffic from the ten traffic generators is around 2.5Gbps.

To obtain cross-traffic with different levels of burstiness, we generate a smoothed version of the replayed traffic by running a token bucket Qdisc on each sending host. We also generate constant-bit-rate traffic from each sender using a UDP flow. We denote these three burstiness levels as **BCT** (the most bursty, raw replayed traffic), **SCT** (smoothed version of BCT) and **CBR** (constant bit-rate). As a measure of burstiness, the 5-95% ranges of the bit-rates (observed at a 1ms timescale) are 2.62Gbps, 1.40Gbps, and 0.49Gbps, respectively.



(a) 10Gbps **(b)** 40Gbps

Fig. 10: Steady-state Throughput with Error-based Losses

TABLE I: Throughput and Loss Ratio of TCP flow

(Gbps)	with replayed traffic			with responsive web traffic	
(%)	UDP	SCT	BCT	RTT=5ms	RTT=30ms
RAPID	6.86	6.61	6.09	6.62	6.61
	0.000	0.014	0.060	0.000	0.001
cubic	3.58	3.25	2.51	6.18	2.79
	0.002	0.002	0.002	0.002	0.004
scalable	4.77	4.38	4.20	7.35	4.23
	0.072	0.046	0.040	0.036	0.037

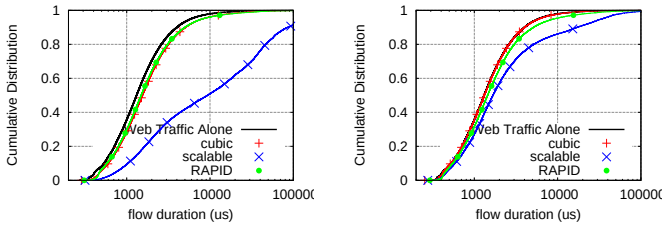
B. Sustained Throughput in Presence of Error-based Losses

Our first set of experiments evaluates achieved throughput in the presence of random bit error-based loss rates, ranging from 10^{-2} (very high) to 10^{-8} (very low). Fig 10 plots the steady-state throughput achieved by each TCP protocol on 10/40Gbps paths. We find significant throughput loss when error rates exceeds 10^{-6} for all protocols—however, RAPID scales much better than others and yields more than double throughput than most protocols. RAPID performs poorly with loss rate 10^{-2}—losses in practically every other p-stream prevent it from estimating avail-bw at all! The performance trends on the 40Gbps path are similar to those on 10Gbps—both agree with the scalability trends reported in [7]. For the remaining evaluations, we use only the switched 10Gbps path.

C. Adaptability to Bursty Traffic

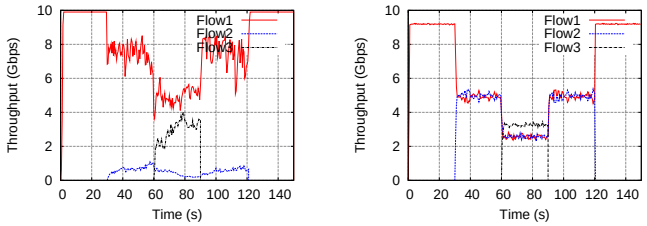
Next we evaluate the ability of the RAPID implementation to adapt to non-responsive, but bursty cross-traffic. We generate and experiment with cross-traffic with different levels of burstiness (as described in Section V-A1), and instantiate a high-speed transfer using different protocols to share the bottleneck link for 120 seconds. Table I shows the average bottleneck link utilization and loss rates observed within the high-speed transfer. We find that:

- The more bursty is the cross-traffic, the lower is the throughput (and link utilization) achieved by a high-speed transfer. This is true for all protocols and is to be expected—finite-buffered switches suffer more losses in the presence of burstier traffic.
- RAPID significantly outperforms other protocols in its ability to adapt to burstiness—it consistently utilizes a much higher fraction of the bursty avail-bw than other protocols. Fig 9 illustrates this for BCT cross-traffic.
- Despite the higher utilization, RAPID incurs much lower packet loss rates—this is indicative of the negligible queuing expected of the protocol [7]. With BCT, RAPID yields 1.8Gbps higher throughput than Scalable, but higher loss rate. However, with a less aggressive rate-adapting parameter (τ = 50ms,η = 1/4) applied, RAPID is able to reduce loss to 0.013% while maintaining high throughput.



(a) RTT=5ms **(b)** RTT=30ms

Fig. 11: Flow Duration of Web Traffic



(a) scalable **(b)** RAPID

Fig. 12: Intra-Protocol Fairness

TABLE II: Necessity of Implementation Mechanisms

RTT=30ms	No cross traffic(Gbps)	With 2.42 Gbps web traffic	
		Throughput(Gbps) Loss(%)	median low duration(ms)
Full RAPID	9.44	7.00, 0.001	1.45
V1 (no PAUSE)	8.37	6.77, 0.004	1.28
V2 (no μs timestamp)	9.83	7.64, 0.217	520.64
V3 (no arrival timestamp)	7.91	6.43, 0.001	1.24
V4 (no BASS)	9.82	7.83, 0.151	8705.24
V5 (no γ) N=64	8.99	6.45, 0.002	1.23
V5 (no γ) N=192	9.49	6.33, 0.007	1.45

D. TCP Friendliness with Responsive and Bursty Web traffic

Web traffic transferred using low-speed TCP continues to dominate the Internet. A high-speed protocol can be deployed over 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 V-A1, and instantiate a high-speed transfer that shares the bottleneck link. We repeat the experiment by using different protocols for the high-speed transfer, as well as with RTT=5ms (to emulate increasing use of content distributions caches).

Note that each of the web transfers sharing the bottleneck link with a high-speed transfer, is 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 use this as the primary metric in this section to study the impact on web traffic.

The more quickly RAPID grabs spare bandwidth, the higher is its throughput—however, the more transient queuing it causes in bottleneck buffers, and the more it impacts the performance of cross-traffic. In RAPID, this tradeoff is controlled by the rate-adaptation parameters (τ, η) [7]. We first study the influence of these two parameters and briefly summarize our findings below:[6]

- *RAPID throughput:* With fixed τ, RAPID throughput first increases with $\frac{1}{\eta}$ due to its more aggressive behavior, and then decreases with it due to more induced losses. Identical $\tau \times \eta$ yields comparable throughputs, which agrees with the simulation results in [28]. As long as $\tau \times \eta \geq 5$, RAPID experiences negligible losses.
- *Web traffic performance:* Smaller $\tau \times \eta$ increases the duration of co-existing low-speed web transfers. 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.

Thus, for network operators targeting minimal impact on web traffic, a more conservative RAPID configuration with larger $\tau \times \eta$ and a larger τ is recommended. For our experiments in this paper, we use $(\tau, \eta) = (50, 4)$.

For $(\tau, \eta) = (50, 4)$, Fig 11 depicts the cumulative distribution of flow durations for web traffic, when they share the bottleneck link with a high-speed transfer (the corresponding throughput and loss rate of the high-speed transfer is listed in Table I). We notice that RAPID impacts web flow duration similarly to Cubic, while yielding much higher throughput (especially with RTT=30 ms). All other protocols either fail to grab a good share of bandwidth (e.g., Cubic), or behave so intrusively so as to more than double the median of flow duration and significantly lengthen the tail distribution (e.g., Scalable). We conclude that RAPID best addresses the trade-off between link utilization and TCP-friendliness—it achieves considerable link utilization while least starving conventional TCP traffic.

[6]For space constraints, we include detailed results only in [25].

E. Intra-protocol Fairness

We next evaluate the intra-protocol fairness properties yielded by our implementation. We initiate three iperf flows between two pairs of end hosts. Each transfer emulates RTT=30ms and is active during different time intervals.

Fig 12a depicts the time-series of throughput obtained by the three transfers while we use Scalable as the underlying protocol—the protocol fails to yield any notion of fair share of the avail-bw. The performance with each of the other state-of-the-art protocols was quite similar [25]. However, RAPID yields much greater fairness among co-existing flows in Fig 12b. This experiment and results are very close to the fairness observed under simulations in [7].

VI. How Critical Are the Added Mechanisms?

We have introduced several mechanisms in order to realize a RAPID implementation on a real system (vs. a simulator, as in [7])—these include: (i) inserting PAUSE frames to ensure precise gaps; (ii) implementing I_R, E_R, and I_S for higher accuracy in timestamping packet arrivals at the receiver; (iii) adapting BASS for accurate bandwidth estimation with short p-streams of $N = 64$; and (iv) decoupling the probing and adapting timescales to alleviate the stability-vs-adaptability tradeoff. In this section, we ask: *are each of these necessary for achieving RAPID performance gains in high-speed settings?*

To answer this, we first run a RAPID flow with the complete set of mechanisms, under two experimental conditions—without any cross-traffic, and with the 2.42 Gbps bursty

responsive cross-traffic on the 10 Gbps switch-to-switch link. Then, we reduce each of these mechanisms individually as follows, and repeat the two experiments above (results are tabulated in Table II):

- V1: Instead of inserting PAUSE frames for gap creation, E_S registers an hrtimer (using **qdisc_watchdog**) for scheduling each packet departure. Fig 6 illustrated that there are significant errors in the intended send gaps in V1—the throughput achieved by RAPID is naturally impacted. However, the impact of inaccurate g_s is lower than expected—this is due to alleviation by BASS, which is good at handling buffering related noise.
- V2 gets rid of I_R and E_R, but relies purely on the TCP timestamp option for estimating receive gaps (no μs precision, ACK processing delays). We find that V2 persistently over-estimates avail-bw as full link capacity[7], starving cross-traffic and causing considerably high packet losses. This is because, with V2, the ms granularity of TCP timestamp obscures any fine-scale queueing delays within p-streams.
- V3 does not generate timestamps at packet arrivals with I_R, but it does replace TCP timestamps with a μs precision value. E_R gets a timestamp by calling ktime_to_ns, and writes it in the TSval header field of returning ACKs. We find that V3 is influenced by ACK processing delays (just like queuing delays) — it under-estimates the avail-bw, and under-utilizes even at idle bottleneck link.
- V4 gets rid of the BASS denoising algorithm, and uses the raw receive gaps of p-streams for bandwidth estimation. We find that V4 persistently over-estimates the avail-bw as 10 Gbps—the spike-dips pattern (Fig 2c) in the p-streams wipes out any underlying trend of persistent queuing delays.
- V5 does not decouple the probing/adapting timescales—both are the p-stream length. We consider both, N=64 and N=192 (recall that in RAPID, probing timescale is N=64 and rate-adapting timescale is N=192). We find that in V5, 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 design components added in this paper is critical for ensuring that RAPID achieves its promised performance in practice.

VII. Concluding Remarks

This paper presents an ultra-high speed implementation of TCP RAPID. We conduct evaluations to show 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, rate, or bandwidth-based congestion control—this paper takes a significant step in presenting evidence to convince them otherwise.

For future work, we will continue to conduct intensive evaluations—we hope to upgrade our end hosts and conduct evaluations at truly 40 Gbps speeds. We also plan to deploy our implementation within scientific applications, especially in a wide-area setting. Besides, we are planning on evaluating our implementation in environments other than the large bandwidth-delay product networks considered here—specifically, data center environments, wireless environments, as well as for wide-area streaming media applications.

[7]This is why V2 and V4 deceitfully offer higher goodputs than RAPID on idle paths (full RAPID is expected to under-estimate avail-bw within 10%).

References

[1] Keith Winstein et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *NSDI*, 2013.

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

[3] 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.

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

[5] Kun Tan et al. A compound tcp approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, 2006.

[6] Andrea Baiocchi et al. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

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

[8] Aleksandar Kuzmanovic et al. Tcp-lp: low-priority service via end-point congestion control. *IEEE/ACM TON*, 2006.

[9] Guojun Jin et al. System capability effects on algorithms for network bandwidth measurement. In *Proc. SIGCOMM*. ACM, 2003.

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

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

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

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

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

[15] 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.

[16] Antony Antony et al. Microscopic examination of tcp flows over transatlantic links. *Future Generation Computer Systems*, 2003.

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

[18] Qianwen Yin et al. Can bandwidth estimation tackle noise at ultra-high speeds? In *Proc. ICNP*. IEEE, 2014.

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

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

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

[22] Qianwen Yin et al. Can machine learning benefit bandwidth estimation at ultra-high speeds? In *Proc. PAM*. Springer, 2016.

[23] Sea Shalunov et al. Low extra delay background transport (ledbat). Technical report, 2012.

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

[25] Extended version of this paper. https://dl.dropboxusercontent.com/u/99049080/infocom17_extension.pdf.

[26] Yashar Ganjali et al. Update on buffer sizing in internet routers. *SIGCOMM*, 2006.

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

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

TABLE III: Throughput with error-based losses (10Gbps)

loss 10^x	no loss	-8	-7	-6	-5	-4	-3	-2
RAPID	9.45	9.44	9.44	9.41	9.07	6.17	1.34	0.01
new reno	9.75	6.13	4.98	2.54	0.923	0.262	0.076	0.03
vegas	9.01	9.66	5.67	3.59	0.97	0.32	0.11	0.03
bic	9.85	9.76	9.62	9.30	5.36	0.919	0.144	0.04
cubic	9.85	9.29	9.08	4.44	1.00	0.24	0.07	0.03
highspeed	9.54	9.54	9.48	8.53	2.43	0.45	0.11	0.03
htcp	9.35	9.35	9.15	6.76	2.29	0.40	0.10	0.03
hybla	9.35	9.23	8.02	6.66	1.96	0.41	0.11	0.03
lp	9.81	7.56	4.97	2.44	0.76	0.26	0.071	0.0269
scalable	9.85	9.61	9.61	9.54	8.98	1.73	0.213	0.01
yeah	9.83	9.83	9.83	9.81	9.03	1.72	0.223	0.07
Fast*	9.10	9.10	6.71	6.25	5.32	4.72	1.32	0.174
illinois	9.61	9.62	9.57	9.32	5.67	1.87	0.62	0.17
westwood	9.73	5.77	5.54	3.59	1.13	0.69	0.252	0.053
veno	9.80	4.99	4.21	2.32	0.90	0.40	0.12	0.04

TABLE IV: Throughput with error-based losses (40Gbps)

loss 10x	noloss	-8	-7	-6	-5	-4	-3	-2
RAPID	18.2	18.2	18.1	17.1	14.9	10.3	2.16	0.03
new reno	20.3	20.3	6.18	1.96	0.67	0.17	0.07	0.03
vegas	19.6	11.0	13.9	3.98	1.02	0.35	0.10	0.04
bic	20.3	20.3	20.3	16.1	5.01	1.01	0.157	0.05
cubic	20.5	20.4	18.7	6.75	1.23	0.22	0.07	0.03
highspeed	20.5	19.9	18.9	8.78	1.72	0.24	0.07	0.03
htcp	20.4	20.0	17.9	8.11	0.51	0.17	0.07	0.03
hybla	20.4	16.9	9.18	2.04	0.54	0.17	0.07	0.03
lp	15.2	10.7	10.4	2.02	0.59	0.16	0.07	0.03
scalable	20.6	20.1	19.9	17.7	8.69	0.88	0.09	0.06
yeah	20.5	20.3	20.3	20.0	11.4	1.78	0.18	0.07
Fast*	20.5	20.2	20.1	6.40	1.84	0.55	0.14	0.05
illinois	20.3	20.3	20.1	18.1	6.11	1.92	0.52	0.14
westwood	20.3	20.3	20.1	6.40	1.84	0.55	0.14	0.05
veno	19.1	19.1	9.94	2.81	0.90	0.21	0.09	0.04

TABLE V: Responsiveness to cross-traffic burstiness

	UDP		SCT		BCT	
	link util(%)	loss (%)	link util(%)	loss (%)	link util(%)	loss (%)
RAPID	90.62	0.000	90.54	0.014	83.43	0.060
new reno	23.77	0.001	26.30	0.001	17.81	0.001
vegas	39.37	0.000	40.41	0.001	34.52	0.001
bic	52.31	0.021	56.16	0.017	53.01	0.011
cubic	47.29	0.002	44.70	0.002	34.38	0.002
highspeed	49.27	0.004	49.59	0.003	46.68	0.002
htcp	45.57	0.008	43.56	0.006	40.82	0.003
hybla	25.10	0.001	23.83	0.001	22.88	0.001
lp	25.39	0.001	25.48	0.001	16.58	0.001
scalable	63.01	0.072	60.24	0.046	57.54	0.040
yeah	64.56	0.064	58.90	0.051	53.01	0.043
Fast*	40.95	0.145	36.98	0.310	28.92	0.375
illinois	49.67	0.002	54.34	0.003	46.03	0.003
westwood	21.93	0.001	22.74	0.001	22.19	0.001
veno	19.29	0.001	20.68	0.001	20.96	0.001

TABLE VI: Impact on Web Traffic (RTT=5ms)

(tau,eta)	throughput (Gbps)	loss (%)	flow duration (ms)		
			50%	10∼90%	5∼95%
no RAPID			1.31	0.54∼4.01	0.42∼5.92
(50,1/4)	6.62	0.000	1.48	0.58∼5.01	0.47∼8.20
(10,1)	6.61	0.000	1.64	0.62∼8.27	0.49∼19.09
(20,1/2)	6.65	0.000	1.59	0.62∼6.34	0.49∼14.21
(30,1/3)	6.67	0.000	1.52	0.58∼5.55	0.47∼10.52
(50,1/6)	6.71	0.000	1.53	0.58∼6.84	0.47∼8.20
(5,1)	6.81	0.000	2.25	0.70∼36.89	0.55∼62.35
(10,1/2)	6.82	0.000	2.16	0.71∼30.78	0.55∼50.76
(20,1/4)	6.82	0.000	1.90	0.64∼24.84	0.48∼43.19
(30,1/6)	6.81	0.000	1.81	0.63∼19.71	0.51∼43.65
(10,1/3)	6.95	0.000	2.48	0.71∼35.78	0.55∼64.47
(20,1/6)	6.91	0.000	2.01	0.68∼41.48	0.54∼93.38
(5,1/2)	7.01	0.002	3.04	0.75∼55.58	0.57∼101.85
(10,1/4)	6.99	0.001	2.65	0.72∼45.55	0.55∼78.43
(5,1/3)	7.03	0.007	4.44	0.78∼82.15	0.59∼241.98
(10,1/6)	7.06	0.004	2.56	0.72∼39.35	0.55∼64.54

APPENDIX

To complement Section V, we present experimental results of all TCP variants available in our testbed: new reno, vegas bic, cubic, highspeed, htcp, hybla, lp, scalable, illinois, westwood, veno, lp, as well as the Fast implementation ported from their simulation code. Unless specified otherwise, we emulate RTT 30ms for all experiments mentioned in this section.

A. Sustained Throughput in Presence of Error-based Losses

We emulate random packet drops of a TCP flow with ratio ranging from 10^{-2} to 10^{-8}. Table III lists their steady-state throughput. RAPID adapts better to none-congestion-induced losses than others. On the 40Gbps link, its performance gain is further scaled — in the 10^{-5} to 10^{-3} regime to more than triples the throughputs of most of other protocols!

B. Adaptability to Bursty Traffic

Table V shows how well each TCP variant adapt to the three burstiness levels of non-responsive cross traffic in our testbed. All protocols observe throughput reduction with higher extent of cross-traffic burstiness — RAPID consistently yields highest goodput and meanwhile least losses.

C. TCP Friendliness with Responsive and Bursty Web Traffic

1) Study the impact of (τ, η): We list in Table VI the supporting results for the observations we made in Section V.D. The 4th column shows the median of web-flow durations sharing the bottleneck with the RAPID flow, 5th column for its 10%-90% cumulative distribution, the 6th 5%-95%. We find that:

- With fixed τ, RAPID throughput first increases with $\frac{1}{\eta}$ due to its more aggressive behavior, and then decreases with it due to more induced losses.
- Identical $\tau \times \eta$ yields comparable throughputs and loss rates.
- With a not-too-small $\tau \times \eta$ (≥ 5), RAPID experiences negligible losses.
- With a smaller $\tau \times \eta$, RAPID grabs more bandwidth share, while lengthens low-speed web flows.
- Although identical $\tau \times \eta$ yeilds similar throughputs, a larger τ helps to reduce the medium and the tail distrbution of flow durations.

2) Performance of all protocols: Based on previous observation, we choose a conservative rate-adaptation parameters $(\tau, \eta) = (50, \frac{1}{4})$. In Table VII and Table VIII we present the throughput of the TCP flow, as well as its impact on web-flow duration. As stated before, RAPID best addresses the trade-

TABLE VII: With Web Traffic (RTT=5ms)

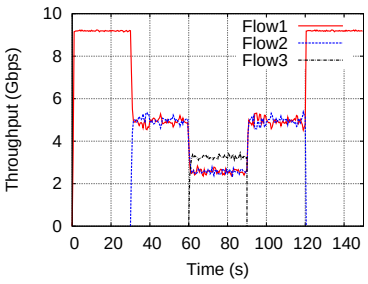
RTT=5ms	throughput (Gbps)	loss (%)	flow duration (ms)		
			50%	10∼90%	5∼95%
no TCP			1.31	0.54∼4.01	0.42∼5.92
RAPID	6.62	0.000	1.48	0.58∼5.01	0.47∼8.20
new reno	5.42	0.002	1.47	0.56∼4.76	0.46∼7.42
vegas	4.43	0.001	1.36	0.51∼4.32	0.41∼6.57
bic	7.14	0.026	3.63	0.809∼50.19	0.62∼92.33
cubic	6.18	0.002	1.47	0.600∼5.07	0.48∼8.12
highspeed	6.26	0.009	1.80	0.62∼12.24	0.48∼25.30
htcp	5.52	0.004	1.51	0.57∼5.74	0.47∼10.50
hybla	5.54	0.002	1.49	0.56∼5.30	0.45∼8.75
lp	5.36	0.002	1.47	0.56∼4.79	0.45∼7.41
scalable	7.35	0.036	5.29	0.93∼71.54	0.70∼146.98
yeah	7.42	0.033	15.96	1.3∼166.65	0.88∼270.47
Fast*	6.29	0.143	1.49	0.56∼44.04	0.45∼137.38
illinois	6.60	0.008	1.75	0.61∼10.81	0.47∼22.78
westwood	5.66	0.008	1.59	0.60∼6.51	0.48∼13.126
veno	5.21	0.002	1.50	0.56∼5.58	0.43∼11.03

TABLE VIII: With Web Traffic (RTT=30ms)

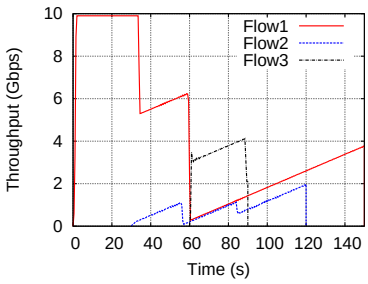
RTT=30ms	throughput (Gbps)	loss (%)	flow duration (ms)		
			50%	10∼90%	5∼95%
no TCP			1.31	0.54∼4.01	0.42∼5.92
RAPID	6.61	0.001	1.45	0.55∼5.14	0.45∼9.13
new reno	1.27	0.001	1.33	0.51∼4.25	0.41∼6.54
vegas	2.21	0.003	1.34	0.51∼4.30	0.41∼6.49
bic	4.18	0.011	1.53	0.56∼6.68	0.45∼15.02
cubic	2.79	0.004	1.32	0.52∼4.01	0.42∼6.05
highspeed	3.39	0.004	1.38	0.52∼4.41	0.43∼6.68
htcp	3.07	0.009	1.37	0.51∼4.47	0.41∼7.11
hybla	1.19	0.002	1.32	0.48∼4.30	0.39∼6.53
lp	1.75	0.001	1.35	0.53∼4.29	0.43∼6.46
scalable	4.23	0.037	1.67	0.59∼17.12	0.468∼34.40
yeah	4.51	0.045	1.73	0.61∼9.98	0.49∼21.47
Fast*	3.03	0.182	2.10	0.61∼121.88	0.48∼301.57
illinois	3.55	0.002	1.43	0.55∼4.53	0.45∼6.92
westwood	1.35	0.033	1.39	0.52∼5.61	0.40∼11.89
veno	1.57	0.001	1.31	0.48∼4.24	0.39∼6.43

off between throughput and friendliness to low-speed TCP flows. Especially with the longer RTT, it maintains comparable throughput as the short RTT while inducing negligible losses. Other protocols designed for large BDP networks, e.g. highspeed, scalable, yeah, observe significant throughput drop when RTT scales from 5ms to 30ms — however, they still lengthen the tail distribution of flow durations to a considerable extent.

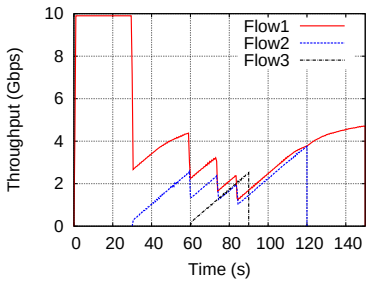
3) Intra-protocol Fairness: Fig 13 plots the throughputs of three TCP flows of the same protocol sharing the bottleneck 10Gbps path. The aggregate of three flows with reno, hybla, lp, westwood and veno severely underutilize the path due to their sluggishness in acquiring avail-bw after losses on such high-speed path. For other protocols, only RAPID maintains near-optimal fair-share.



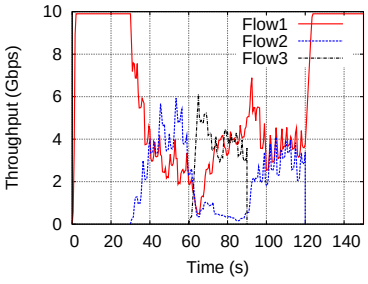
(a) RAPID



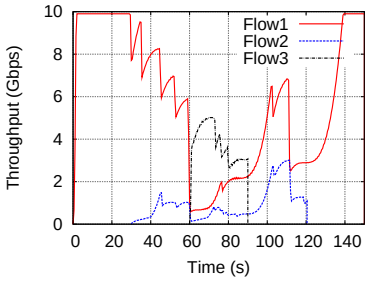
(b) New Reno



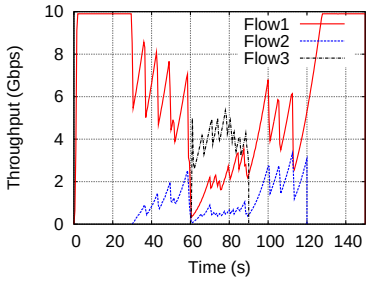
(c) vegas



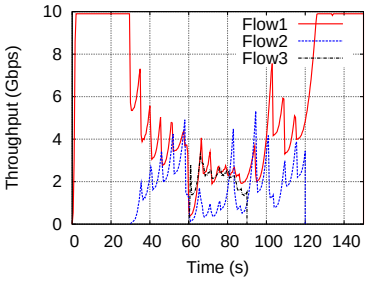
(d) bic



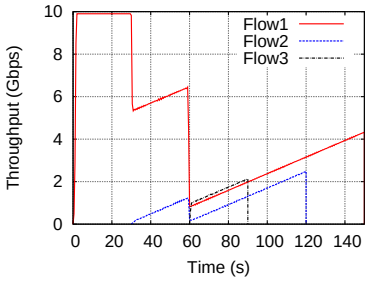
(e) cubic



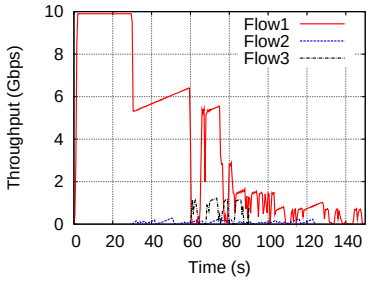
(f) highspeed



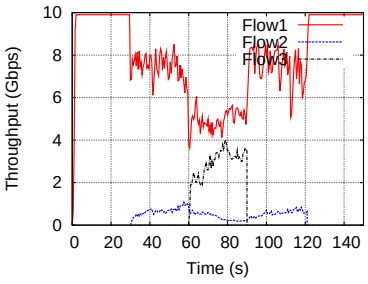
(g) htcp



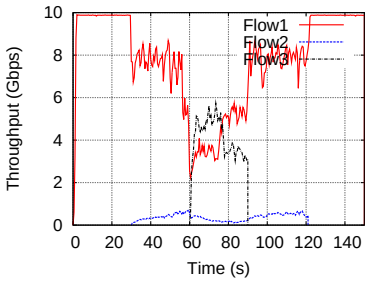
(h) hybla



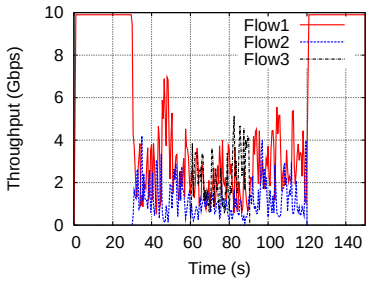
(i) lp



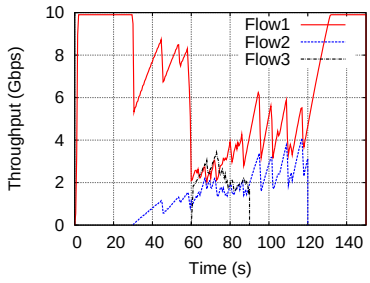
(j) scalable



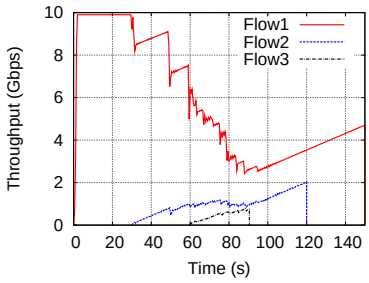
(k) yeah



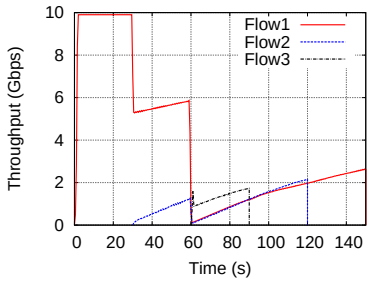
(l) Fast



(m) illinois



(n) westwood



(o) veno

Fig. 13: Fairness