A Model Predictive Control Approach to Flow Pacing for TCP

David Fridovich-Keil, Nathan Hanford, Margaret P. Chapman,
Claire J. Tomlin, Matthew K. Farrens, and Dipak Ghosal

Abstract—
A key challenge in the management of Internet traffic is the design of algorithms that complement well-established protocols, such as the Transmission Control Protocol (TCP), and simultaneously address their limitations. The challenge becomes greater in the context of large so-called "elephant" flows over long paths that often transition from higher to lower bandwidth connections. At these transition points either persistent queues are formed when buffers are over-provisioned or packet loss occurs when buffers are under-provisioned; both cases lead to degraded and/or highly variable end-to-end performance. Ideally, for such scenarios, the source should "learn" and set a pacing rate that matches the lower bandwidth connection. In this paper, we adopt a model-based receding horizon control strategy to design a pacing control method. Each new round-trip time (RTT) measurement is first incorporated into a linear time-varying (LTV) predictive model. Subsequently, we solve a one-step look-ahead optimization problem which finds the pacing rate which optimally trades off RTT, variance in RTT, and throughput according to the most up-to-date model. We implemented our proof-of-concept control strategy on the Linux operating system alongside the existing CoDel queuing discipline (qdisc) and HTCP congestion-control algorithm. Our preliminary results indicate significant reduction in the variances of the RTT and the throughput, resulting in more predictable performance overall.

I. Introduction

Long-distance data transfer in the presence of bandwidth mismatches is relatively commonplace. Indeed, often the data we download, e.g. streaming videos from an online content provider, must first transfer from a high-speed backbone network to a low-speed access network and then in some cases transfer again to lower-speed home networks, much as cars travelling on highways must eventually transition to smaller roads. At locations where these bandwidth mismatches occur, either persistent queues – so-called *bufferbloat* [8] – are formed when buffers are over-provisioned or packet loss occurs when buffers are under-provisioned. Both cases lead to degraded end-to-end performance. Currently, these scenarios are addressed by various active queue management (AQM) strategies [12]. One algorithm growing in popularity is Controlled Delay Management (CoDel) [15]. CoDel ameliorates bufferbloat-induced delays with minimal impact on Internet link utilization by monitoring overall delay and accordingly placing limits on the sending queue size [16, 11]. For multiple flows, a well-designed scheduler called *fair queuing* (FQ) ensures that CoDel drops packets from flows with growing queues, which facilitates desirable system performance and effective bufferbloat management in many scenarios [11]. However, CoDel is not expected to mitigate delays experienced by a single sustained TCP flow, since the algorithm becomes too sensitive to the value of the *interval* parameter, which specifies what it means for packets to remain in queues for "too long" [16], and too many packets may be dropped. Further, single-stream data transfer from high bandwidth at the sender to low bandwidth at the receiver under TCP-CoDel is challenging, particularly in the context of large "elephant" flows over long distances.

For this bandwidth mismatch scenario, the ideal solution would be for the sender to "learn and set" a maximum pacing rate that matches the lowest bandwidth link in the end-to-end path. If done properly, a single connection would not exceed the capacity of the bottleneck bandwidth and, hopefully, have predictable throughput and delay performance. If, however, there are multiple flows sharing that bottleneck then the TCP protocol along with a well-designed fair queuing strategy will ensure that each flow receives an equal share of the capacity.

A key step in achieving predictable throughput and delay – which is one of our key goals in this work – is flow pacing. The idea behind flow pacing is that packets sent at regular intervals rather than infrequent bursts will lead to lower latency and higher throughput. As such, flow pacing is also applicable in general highly utilized networks, much as busy highways employ ramp metering. This is fundamentally different from AQM strategies such as Random Early Detection (RED) [7], BLUE [6], Controlled Delay (CoDel) [16], and Proportional Integral controller Enhanced (PIE) [17]. Broadly, AQM attempts to manage growing router buffers (i.e. network latency) by cleverly dropping packets from different flows to trigger loss-based congestion control mechanisms in the TCP [10]. By contrast, flow pacing tries to adjust the rate of data flow steadily in order to avoid bufferbloat in the first place.

There is much prior work on flow pacing [1, 18, 9]. Broadly speaking, flow pacing can be performed at the source host or at the edge where the access network connects to the core network. The former is referred to as host pacing, or more commonly TCP pacing, while the later is referred to as edge pacing and can be performed by the internet service provider (ISP) [9]. In this work we focus on TCP

D. Fridovich-Keil, M. Chapman, and C. Tomlin are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. N. Hanford, M. Farrens, and D. Ghosal are with the Department of Computer Science, University of California, Davis. D. Fridovich-Keil is supported by the NSF GRFP. This work is supported by NSF grant CNS 1528087. {dfk,chapmanm,tomlin}@eecs.berkeley.edu, {nhanford,mkfarrens,dghosal}@ucdavis.edu

pacing, which has been studied since the initial proposal [19]. More recent studies [1, 18, 9] have shown that host pacing in most cases does improve TCP throughput, although it is unclear if TCP paced flows receive their fair share of network bandwidth when competing with non-paced TCP flows [18].

While the mechanism of flow pacing makes intuitive sense, existing implementations in the Linux kernel, specifically in the traffic control subsystem [13], do not incorporate online validation of flow pacing's key assumption. That is, flow pacing implementations do not currently estimate their effect on packet round trip time (RTT) over time. This paper describes a preliminary attempt to incorporate such a model into a predictive controller which sets the maximum pacing rate to optimize flow control. Our results suggest that a relatively simple linear model of RTT and a quadratic control objective can effectively minimize RTT while keeping it very steady over time. Although it is difficult to measure, we hypothesize that maintaining such a low-variance RTT may improve overall network throughput efficiency.

II. Background

The sender-side functions that determine the rate at which data is injected into the network are shown in Figure 1. The rate at which data is written into the socket buffer depends on the application. In this work we are concerned with elephant flows and hence the rate at which data is injected into the network is not application limited. TCP operates on the socket send buffer and determines which and how much data should be sent. ACKs from the receiver provide information as to which data can be sent next. The amount of data to be sent, i.e., the window size, is determined by TCP flow and congestion control algorithms. To achieve end-to-end flow control, the receiver informs the sender how much data it can accept. This receiver advertised window size is sent with the ACKs. The congestion control algorithm determines the congestion window based on the estimate of the network congestion; the precise manner in which this is determined

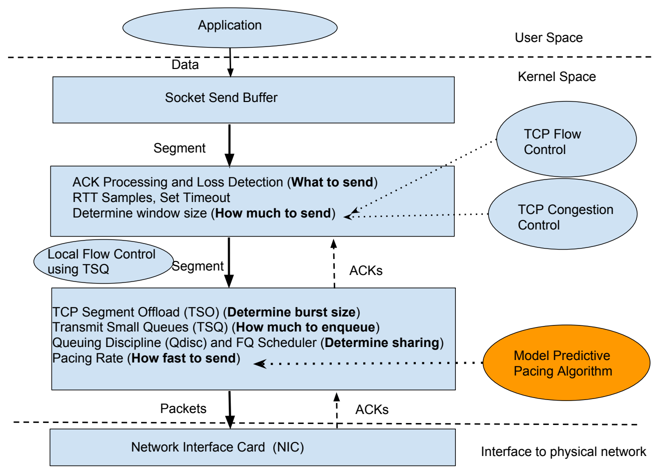

Application
User Space
Data
Kernel Space
Socket Send Buffer
Segment
TCP Flow Control
ACK Processing and Loss Detection (What to send)
RTT Samples, Set Timeout
Determine window size (How much to send)
TCP Congestion Control
Local Flow Control using TSQ
Segment
ACKs
TCP Segment Offload (TSO) (Determine burst size)
Transmit Small Queues (TSQ) (How much to enqueue)
Queuing Discipline (Qdisc) and FQ Scheduler (Determine sharing)
Pacing Rate (How fast to send)
Model Predictive Pacing Algorithm
Packets
ACKs
Network Interface Card (NIC)
Interface to physical network

Fig. 1: Sender-side functions that determine the rate at which data is injected into the network (adapted from [5]).

depends upon the particular congestion control algorithm in use [10]. The amount of data that is sent is the minimum of the window sizes determined by the TCP flow control and the congestion control algorithms.

The data which TCP decides to send out is first passed through lower level protocol processing and then through the network interface. These processes are controlled by a number of functions that determine how much data is eventually injected into the network [5].

1) **TCP Segmentation Offload (TSO)**: While the physical network has a Maximum Transmission Unit (MTU) size, having the TCP layer handoff MTU-sized segments to the lower layer results in very poor CPU efficiency. TSO allows TCP to hand off large segments (up to 64 KB) to the lower layer which segments the large chunks into MTU size packets. While segmenting into larger chunks expends fewer CPU cycles, smaller chunks allow for smaller bursts to be transmitted into the network. This leads to lower network queue sizes which in turn reduces queuing delays and packet loss rates.
2) **TCP Small Queue (TSQ)**: TSQ performs local flow control between the network layer queue and the TCP layer. TSQ limits the amount of data in the queues on the sending host by controlling when TCP can handoff segments to the network layer. Similar to TSO, there is a tradeoff; keeping the queues small reduces latency and head-of-line blocking while keeping the queues large ensures flow pressure to keep the link fully utilized.
3) **Queuing Discipline (Qdisc) and FQ Packet Scheduler**: A qdisc is a packet scheduler which determines how to queue outgoing packets. It contains functions to filter and classify packets and methods to handle matched packets, i.e. packets with the same flow descriptor. Additionally, the scheduler implements a Fair Queue (FQ) mechanism which attempts to provide fair bandwidth sharing between flows. It does this by performing round-robin packet scheduling over a red-black tree which stores the queue state for each flow. Upon each visit to a queue, that queue is given a time quantum in which to dequeue packets.
4) **Pacing**: The goal of pacing is to spread out the transmission of chunks of data (determined by TSO), rather than immediately bursting them onto the line. Pacing also makes rate limiting possible. In the case of CoDel, pacing is done efficiently by setting limits on the queue size.

As shown in Figure 1 this work focuses on the pacing algorithm. In Linux, pacing is performed by a nanosecond-granularity timer that schedules packets for transmission. The overall sending rate is determined by the TCP congestion control algorithm. Most established TCP protocols [10] rely on a loss-based congestion control algorithm to determine the sending rate. Such strategies keep bottleneck queues full, leading to poor network latency. Recently, BBR TCP [4] has demonstrated the effectiveness of latency-based congestion control. BBR works by quickly acquiring both the bottleneck bandwidth and the nominal RTT, pacing data so that the amount in-flight matches the bottleneck bandwidth-delay product (which keeps queues small), and periodically probing the bottleneck bandwidth to adjust for competing flows or changes in network topology. TCP Vegas [3] also uses RTT as the control input. In TCP Vegas, the RTT is carefully monitored. If there is a packet drop or increase in RTT, the sending rate is decreased whereas if the RTT is steady, the sending rate is increased until either a packet drop occurs or RTT increase is observed. TCP Vegas suffers from fairness issues when competing with purely loss-based congestion control.

In this work we propose a different approach to latency-based congestion control. In particular, our controller sets the maximum pacing rate by solving a model-based receding horizon control problem at each time step. Each new round-trip time (RTT) measurement is first incorporated into a linear time-varying (LTV) predictive model. Subsequently, we solve a one-step look-ahead optimization problem which finds the pacing rate which optimally trades off RTT, RTT variance, and throughput according to the most recent model. Our method is computationally inexpensive making it readily implementable on current systems.

III. Model Predictive Pacing Algorithm

We take a model-based receding horizon control approach to flow pacing. Each new RTT measurement is first incorporated into a LTV predictive model of RTT at the next time step. Then, we solve a quadratic optimization problem which finds the pacing rate that optimally trades off predicted RTT, predicted RTT variance, and throughput according to the new model. Section III-A presents the LTV model, and Sec. III-B describes the corresponding model predictive control (MPC) problem.

A. Predictive Model

We estimate RTT at time step n, $\hat{\ell}(n)$, as a linear function of its immediate past history and that of the max pacing rate r, i.e.

$$\hat{\ell}(n) = \sum_{i=1}^{p} a_i \ell(n-i) + \sum_{i=1}^{q} b_i r(n-i) \tag{1}$$

The constants p and q govern how far back in time $\ell(n)$ is allowed to depend on the past. Since the behavior of RTT may change over time, e.g. as flows leave or join the route or as the route itself changes, we allow the $\{a_i\}$, $\{b_i\}$ to change as well. In particular, we adjust their values using stochastic gradient descent according to the following update rule after receiving each new RTT $\ell(n)$.

$$\begin{cases} a_i \longleftarrow a_i - \alpha\left(\hat{\ell}(n) - \ell(n)\right)\ell(n-i) \\ b_i \longleftarrow b_i - \alpha\left(\hat{\ell}(n) - \ell(n)\right) r(n-i) \end{cases} \tag{2}$$

These updates correspond to minimizing the objective function $\frac{1}{2}(\hat{\ell}(n) - \ell(n))^2$ by taking a gradient step scaled by α. We find that setting $\alpha = 0.5$ works well in practice, although the convergence of such approximate gradient methods is only guaranteed in theory for some range of α which depends upon the noise characteristics of the time series $\ell(n)$ and $r(n)$. Unfortunately these statistics are unknown *a priori* and moreover there is no guarantee of properties such as stationarity which are typically assumed.

B. Model Predictive Control Strategy

Given this model, we formulate the following optimization problem to choose the pacing rate at the n-th time step $r(n)$ which minimizes both the predicted RTT and its variance at the next time step while maximizing pacing rate,[1] according to trade-off parameters $\psi, \xi > 0$:

$$r(n) = \underset{0 \le r \le R}{\operatorname{argmin}} \frac{\hat{\ell}(n+1)}{\mu_\ell(n)} + \psi \frac{\mathrm{Var}(\hat{\ell}(n+1))}{\mu_v(n)} - \xi \frac{r}{\mu_r(n-1)} \tag{3}$$

Note that we have divided each term in the objective function by an exponentially-weighted moving estimate of its mean, i.e.

$$\begin{cases} \mu_\ell(n) &= \gamma \mu_\ell(n-1) + (1-\gamma)\ell(n) \\ \mu_v(n) &= \gamma \mu_v(n-1) + (1-\gamma)\mathrm{Var}(\hat{\ell}(n)) \\ \mu_r(n) &= \gamma \mu_r(n-1) + (1-\gamma) r(n) \end{cases} \tag{4}$$

The parameter γ controls the time-scale of the moving average. In practice, we set it to 0.5, which means that the mean estimators μ are heavily weighted toward recent measurements. This setting allows for rapid adjustment to changes in dynamics; in some settings however it may be more appropriate to set this somewhat higher in order to be less sensitive to rapid changes in RTT.

Dividing by the mean effectively normalizes the scale of ψ and ξ so that the solution to Eq. 3 remains meaningful for links with dramatically different nominal latencies, latency variability, and bottleneck bandwidth. Practically speaking, the effect is to place all three terms on the same relative scale so that, for example, a 10% increase in latency is offset by a 10% decrease in variance (assuming $\psi = 1$ for clarity). The normalization is delayed by one time step in order to decouple it clearly from the choice of $r(n)$, upon which $\hat{\ell}(n+1)$ depends. We expand $\hat{\ell}(n+1)$ as a function of r as follows:

$$\hat{\ell}(n+1) = b_1 r + \sum_{i=0}^{p-1} a_{i+1}\ell(n-i) + \sum_{i=1}^{q-1} b_{i+1} r(n-i) \tag{5}$$

and we approximate the variance of predicted RTT with the squared deviation from the (exponential) average, i.e.

$$\mathrm{Var}(\hat{\ell}(n+1)) \approx (\hat{\ell}(n+1) - \mu_\ell(n))^2 \tag{6}$$

With this approximation in mind, we may solve for $r(n)$ in closed form by minimizing the objective function in Eq.

[1]Ideally, we would like to maximize throughput, not pacing rate. However, as we are unable to measure throughput directly in real-time, we use pacing rate as a proxy. The quality of this proxy is validated in Figs. 2b and 3b where control rate closely matches throughput measured *post hoc*.

Algorithm 1 MPC Flow Pacing

```
1: initialize $\mu_\ell, \mu_v, \mu_r$
2: initialize $\{a_i\}, \{b_i\}$
3: while $\ell \longleftarrow$ NewPacketRTT() do
4:     update $\{a_i\}, \{b_i\}$                         ▷ Eq. 2
5:     update $\mu_\ell, \mu_v$                          ▷ Eq. 4
6:     compute optimal $r$                               ▷ Eq. 7
7:     update $\mu_r$                                    ▷ Eq. 4
8:     if time since last pacing rate change $> T$ ms then
9:         set max pacing rate to optimal $r$
10:    end if
11: end while
```

3. The result is given by

$$r(n) = \frac{\left(\frac{\xi}{\mu_r(n-1)} - \frac{b_1}{\mu_\ell(n)}\right)\frac{\mu_v(n)}{2\psi b_1} + \mu_\ell(n) - \left[\hat{\ell}(n+1)\right]_{r(n)=0}}{b_1} \tag{7}$$

where $\left[\hat{\ell}(n+1)\right]_{r(n)=0}$ denotes the predicted latency at time step $n+1$ if the controller were to set $r(n)$ to zero, i.e.

$$\left[\hat{\ell}(n+1)\right]_{r(n)=0} = \sum_{i=0}^{p-1} a_{i+1}\ell(n-i) + \sum_{i=1}^{q-1} b_{i+1}r(n-i) \tag{8}$$

The steps above are summarized in Alg. 1. In brief, upon receipt of a new packet we record its RTT, update the $\{a_i\}, \{b_i\}$ as in Eq. 2, and update $\mu_\ell(n)$ and $\mu_v(n)$ as in Eq. 4. Then, we compute $r(n)$ from Eq. 7, update $\mu_r(n)$, and set the max pacing rate accordingly. We allow the max pacing rate to be changed only every $T = 10$ ms, so if that amount of time has not yet elapsed, we leave the max pacing rate unchanged.

C. Validation

Other popular models such as the fluid-based model of Misra, Gong, and Towsley [14] are non-linear ordinary differential equations, yet our model is linear. The justification for this simplification is subtle, but the key point is that in order to make the next control decision a model only needs to be accurate near the current operating point and for a short time horizon. More precisely, if we are only changing the pacing rate by a small amount at every time step, a linear approximation to the full dynamics suffices to predict the effect on RTT. Moreover, as the operating point changes our model adaptively learns a linear approximation to the new dynamics.

Additionally, as we use stochastic gradient descent to update model parameters in Eq. 2, the model represented by the $\{a_i\}, \{b_i\}$ may take several iterations to converge. As a general rule, more complicated models may take even more iterations to converge than our relatively simple linear model. Thus, our choice of a linear model is intended to facilitate model convergence and rapid adaptation to new dynamics about a changing operating point.

Figure 2a demonstrates the effectiveness of this modeling approach for a testing configuration with bottleneck capacity of 10 Gbps (see Sec. IV-A for details). The model's predicted latency closely tracks the actual measured latency through its characteristic saw-tooth pattern despite high-frequency noise. We present further results in Sec. IV.

IV. Results

A. Experimental Setup

Our experimental setup simulates an increasingly prevalent cluster computing or networked high performance computing scenario where results of a computational task must be sent first over a very high-speed Local Area Network (LAN), and then over a high-speed Wide-Area Network (WAN). We used two Dell T630 servers running Ubuntu Linux 16.04. The sending system was connected to the intermediate system with a 40 Gbps local link. The intermediate system was then connected to a 10 Gbps WAN link. The sending system would send data as quickly as possible to different remote hosts through the intermediate system.

The Linux network stack provides several user-space utilities for influencing flow behavior. In particular, the Linux advanced routing and traffic control (LARTC) facility provides `tc`, an advanced traffic control subsystem which mimics some of the functionality of high-performance routers. `tc` allows for the implementation of queuing disciplines (qdiscs), which control the behavior of flows as they leave a system. Controlled Delay (CoDel) is an egress queue management system which puts hard limits on the real queue size in order to minimize RTT. In our case we used the CoDel's hard limits on the real size of egress queues in order to limit throughput quickly and with low overhead. In this manner, our control is implemented "on top of" CoDel. This qdisc was used on both the sending system and the intermediate node.

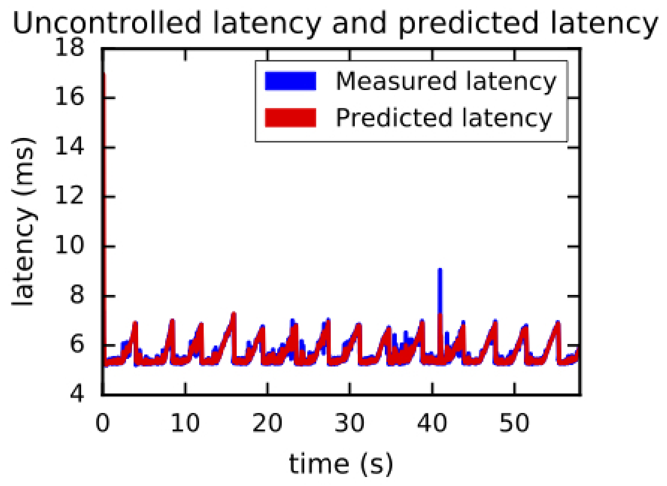
TABLE I: Summary of the Systems Under Test

System	Dell T630
Processor	Dual Intel E5-2637v3
40G NIC	Mellanox ConnectX-3
10G NIC	Intel 82599ES
Operating System	Ubuntu Linux 16.04
Kernel	4.4.0-83-generic x86_64
Queuing Discipline	FQ CoDel
TCP Congestion Control	HTCP and Reno

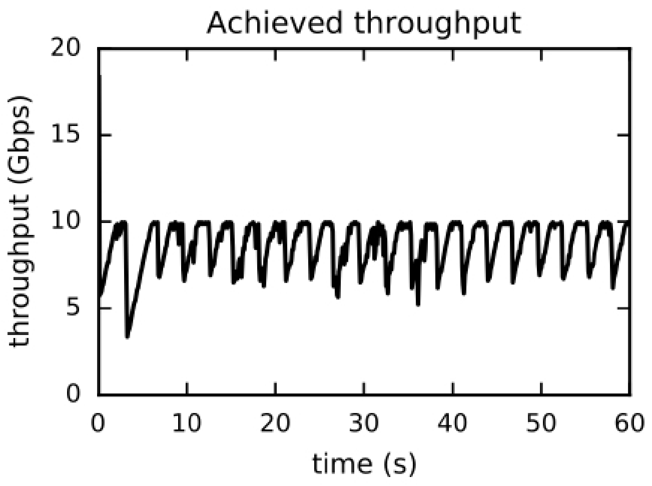
B. Single Flow Over Paths with Capacity Mismatch

We begin by demonstrating the efficacy of our control strategy for a single bulk flow on the experimental setup described in Sec. IV-A. Figure 2 shows RTT over time for such a flow from UC Davis to a test host in Sacramento both with and without control. Our controller completely eliminates the saw-tooth pattern in RTT which is characteristic of loss-based congestion control, and quickly converges to a pacing rate of just slightly less than the true bottleneck bandwidth of 10 Gbps.

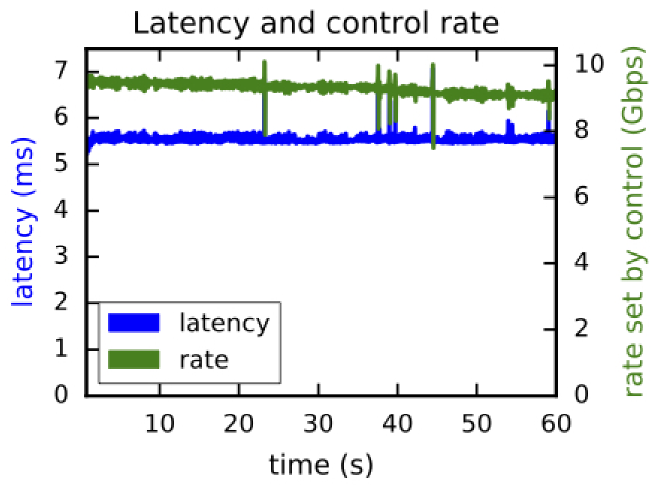
Figure 3 shows the corresponding throughput for the same flows. As before, the controller effectively eliminates the



(a) Uncontrolled RTT for a bulk flow from UC Davis to Sacramento.

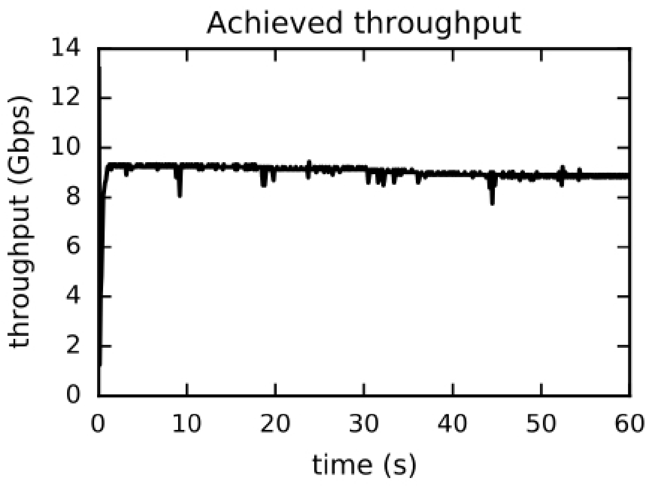


(a) Uncontrolled throughput for the flow shown in Fig. 2a.



(b) Controlled RTT and controller output over time for a bulk flow from UC Davis to Sacramento.

Fig. 2: Our controller (2b) successfully eliminates the characteristic saw-tooth pattern in RTT in (2a). Note that our controller immediately converges upon a pacing rate of just slightly less than the true bottleneck capacity of 10 Gbps.



(b) Controlled throughput for the flow shown in Fig. 2b.

Fig. 3: Our controller (3b) successfully eliminates the characteristic saw-tooth pattern in throughput in (3a).

saw-tooth pattern in the uncontrolled throughput. The slight downward slope in control rate in Fig. 2b and throughput in Fig. 3b is likely an indication that the ξ parameter could be increased, which would make the controller value throughput more highly.

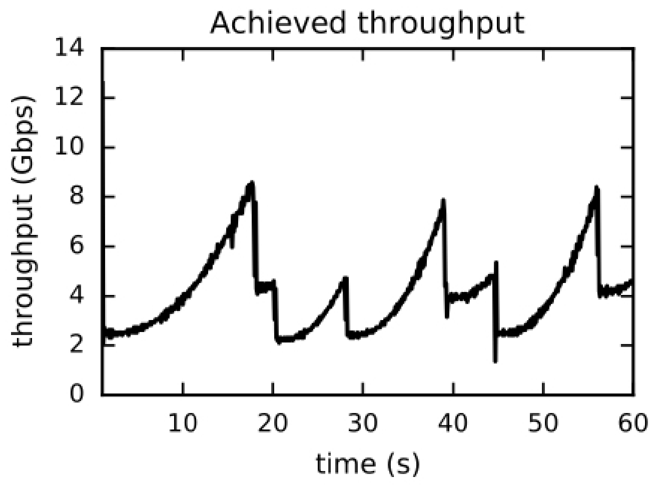
C. Single Flow with Transient Congestion

Figure 4 shows how our controller reacts to transient congestion artificially induced by periodically changing the bottleneck capacity between 10 and 5 Gbps. Despite these sudden changes in capacity, our model is effectively able to disambiguate the effect of control rate on RTT and stabilize throughput at a level just under the lower of the two capacities (5 Gbps). Although such performance does not take full advantage of the available capacity while the capacity is 10 Gbps, that was not our goal. In this work, we are more concerned with maintaining a steady, predictable, low RTT and hence more consistent behavior of communication-intensive applications. Future work may be devoted to optimizing performance for throughput.

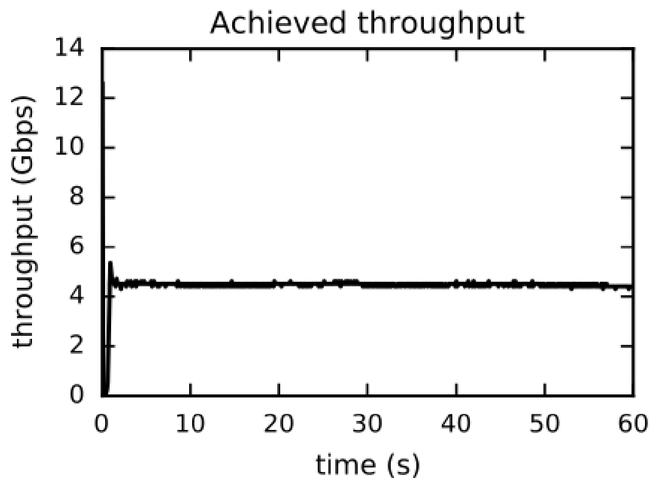
D. Fairness

An important concern for any novel TCP control algorithm is the notion of fairness. That is, a flow using a new controller should ideally consume the same share of network bandwidth as any other competing flows, regardless of what congestion control algorithm they are using.

Figure 5 shows how our controller reacts to a competing flow running HTCP, a particularly aggressive congestion control algorithm which builds its window quickly. The controller quickly claims its fair share (half) of the available 10 Gbps bottleneck capacity, then stabilizes around 4 Gbps. Although this is slightly less than the fair share, the controller is able to maintain a stable RTT throughout, which was our design objective.



(a) Uncontrolled throughput in the presence of transient congestion.



(b) Controlled throughput in the presence of transient congestion.

Fig. 4: Our controller (4b) successfully eliminates the large fluctuations in throughput in (4a) caused by transient router congestion.

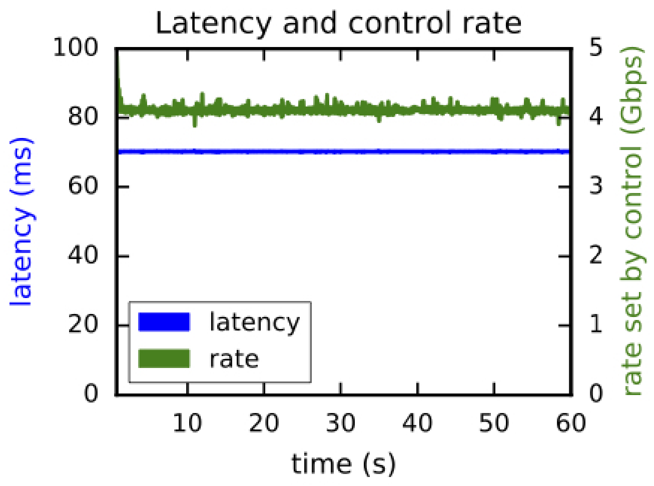


Fig. 5: Performance in the presence of a competing flow running HTCP.

E. Comparison with BBR TCP

As explained in Sec. II, our work is similar in spirit to BBR TCP [4], the major difference being our use of a simple, explicit learned model of RTT dynamics and corresponding optimal control law. Since BBR has only recently been released in beta form on the Linux kernel, we have not had the chance to make a direct empirical comparison between our work and BBR. We are excited to pursue this direction in the future.

Based on the description of BBR and results presented in Cardwell et al. [4], the major qualitative difference between our approach and BBR lies in how each acquires the nominal RTT. BBR periodically probes the network by increasing the pacing rate (see Fig. 5 in [4]), whereas our controller balances the benefit of increasing pacing rate against predicted changes in RTT and RTT variability according to a predefined cost function. This leads to periodic increases in RTT for flows using BBR TCP, and more steady RTT for flows using our controller (e.g. Fig. 2b).

F. Impact of Parameter Setting

Here, we review the key parameters of our algorithm and explain the theoretical and empirical effect of each. Parameters are grouped by whether they pertain to the model or to the controller.

1) Model Parameters: Our model from Eq. 1 contains two user-specified parameters: p and q, which represent the number of past samples of RTT ℓ and control output r, respectively, with which to predict the next RTT. Increasing p and q give the model a longer memory, which may be important on systems with particularly long nominal RTTs. In practice, we set $p = 5$ and $q = 1$.

Additionally, the model update procedure from Eq. 2 contains a step size parameter α, which controls the convergence rate and volatility of the model. Increasing α may lead to faster model convergence, though at the expense of volatility due to the amplification of noisy RTT measurements. In practice, we set $\alpha = 0.5$.

2) Controller Parameters: The control procedure from Eq. 3 depends upon two key parameters, ψ and ξ, which control the importance of RTT variance and control output relative to predicted RTT, respectively. Setting both to unity implies that the controller should treat, for example, a 10% increase in normalized predicted RTT as equivalent to a 10% decrease in normalized RTT variance or a 10% increase in normalized control. In practice, we set $\psi = 1$ and $\xi = 100$.

The exponential moving averages of each of these three quantities, given in Eq. 4, depend upon a single parameter $\gamma \in (0, 1)$ which controls the memory of the moving average. Higher γ indicates a longer memory, meaning that the average is significantly impacted by measurements from the distant past. In practice, we set $\gamma = 0.5$.

V. Conclusions

In this paper we have described a novel approach to controlling RTT for modern high intensity internet applications. Our work differs from the state of the art in several regards, but mainly in its explicit incorporation of an adaptively-learned predictive model of RTT and receding horizon control scheme rather than on techniques from linear control theory [2] or on repeated probing of bottleneck bandwidth [4]. Our method is able to stabilize RTT and throughput under a variety of conditions, and qualitative analysis suggests a favorable comparison to the current state of the art.

Future work will examine the quantitative differences in behavior between techniques such as BBR and our method. We are also interested in designing an auto-tuning scheme for model and control parameters, and optimizing our algorithm more aggressively for high throughput. Eventually, we hope to release a qdisc for the Linux kernel.

References

[1] Amit Aggarwal, Stefan Savage, and Thomas Anderson. "Understanding the performance of TCP pacing". In: *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. IEEE. 2000, pp. 1157–1165.

[2] Takehito Azuma, Tsunetoshi Fujita, and Masayuki Fujita. "Congestion control for TCP/AQM networks using state predictive control". In: *Electrical Engineering in Japan* 156.3 (2006), pp. 41–47.

[3] Lawrence S. Brakmo and Larry L. Peterson. "TCP Vegas: End to end congestion avoidance on a global Internet". In: *IEEE Journal on selected Areas in communications* 13.8 (1995), pp. 1465–1480.

[4] Neal Cardwell et al. "BBR: Congestion-Based Congestion Control". In: *Queue* 14.5 (Oct. 2016), 50:20–50:53. ISSN: 1542-7730. DOI: 10.1145/3012426.3022184. URL: http://doi.acm.org/10.1145/3012426.3022184.

[5] Yuchung Cheng and Neal Cardwell. "Making Linux TCP Fast". In: *The Technical Conference on Linux Networking (NETDEV 1.2)*, pp. 73–80.

[6] Wu-chang Feng et al. "The BLUE active queue management algorithms". In: *IEEE/ACM Transactions on Networking* 10.4 (Aug. 2002), pp. 513–528. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.801399.

[7] S. Floyd and V. Jacobson. "Random early detection gateways for congestion avoidance". In: *IEEE/ACM Transactions on Networking* 1.4 (Aug. 1993), pp. 397–413. ISSN: 1063-6692. DOI: 10.1109/90.251892.

[8] Jim Gettys and Kathleen Nichols. "Bufferbloat: Dark buffers in the Internet". In: *Communications of the ACM* 55.1 (2012), pp. 57–65.

[9] Hassan Habibi Gharakheili, Arun Vishwanath, and Vijay Sivaraman. "Comparing edge and host traffic pacing in small buffer networks". In: *Computer Networks* 77 (2015), pp. 103–116. ISSN: 1389-1286. DOI: http://dx.doi.org/10.1016/j.comnet.2014.11.021. URL: http://www.sciencedirect.com/science/article/pii/S1389128614004393.

[10] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS Operating Systems Review* 42.5 (2008), pp. 64–74.

[11] T Hoeiland-Joergensen et al. *The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm*. draft-ietf-fq-codel-06 (work in progress). Mar. 2016.

[12] C. V. Hollot et al. "Analysis and Design of Controllers for AQM Routers Supporting TCP Flows". In: *IEEE Transactions on Automatic Control* 47.6 (2002), pp. 945–959.

[13] Bert Hubert et al. *Linux Advanced Routing & Traffic Control HOWTO*. Tech. rep. 2002.

[14] Vishal Misra, Wei-Bo Gong, and Don Towsley. "Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED". In: *ACM SIGCOMM Computer Communication Review*. Vol. 30. 4. ACM. 2000, pp. 151–160.

[15] Kathleen Nichols and Van Jacobson. "Controlling Queue Delay". In: *Communications of the ACM* 55.7 (2012), pp. 42–50.

[16] K Nichols et al. *Controlled Delay Active Queue Management*. draft-ietf-aqm-codel-04 (work in progress). June 2016.

[17] Rong Pan et al. "PIE: A lightweight control scheme to address the bufferbloat problem". In: *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*. IEEE. 2013, pp. 148–155.

[18] D Wei et al. "TCP pacing revisited". In: *Proceedings of IEEE INFOCOM*. 2006.

[19] Lixia Zhang, Scott Shenker, and Daivd D Clark. "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic". In: *ACM SIGCOMM Computer Communication Review* 21.4 (1991), pp. 133–147.