Elasticity Detection: A Building Block for Internet Congestion Control

Prateesh Goyal[1], Akshay Narayan[2], Frank Cangialosi[2], Srinivas Narayana[3], Mohammad Alizadeh[2], Hari Balakrishnan[2]
[1]Microsoft Research, [2]MIT Computer Science & Artificial Intelligence Lab, [3]Rutgers University
USA

Abstract

This paper introduces a new metric, "elasticity," which characterizes the nature of cross-traffic competing with a flow. Elasticity captures whether the cross traffic reacts to changes in available bandwidth. We show that it is possible to robustly detect the elasticity of cross traffic at a sender without router support, and that elasticity detection can reduce delays in the Internet by enabling delay-controlling congestion control protocols to be deployed without hurting flow throughput. Our results show that the proposed method achieves more than 85% accuracy under a variety of network conditions, and that congestion control using elasticity detection achieves throughput comparable to Cubic but with delays that are 50–70 ms lower when cross traffic is inelastic.

CCS Concepts

• **Networks → Transport protocols**.

Keywords

Congestion Control, Elasticity Detection.

ACM Reference Format:
Prateesh Goyal[1], Akshay Narayan[2], Frank Cangialosi[2], Srinivas Narayana[3], Mohammad Alizadeh[2], Hari Balakrishnan[2]. 2022. Elasticity Detection: A Building Block for Internet Congestion Control. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3544216.3544221

1 Introduction

Achieving high throughput and low delay has been a key goal of congestion control research for decades. An important category of proposals is *delay-controlling* congestion control protocols. To minimize delays while avoiding "bufferbloat" [16], these schemes (e.g., Vegas [2], FAST [45], LEDBAT [38], Sprout [46], Copa [1]) reduce their rates as delays increase, unlike *buffer-filling* methods like Cubic [19], NewReno [21], and Compound [39] that must fill buffers to elicit congestion signals (packet losses or ECN). Delay-controlling protocols offer a deployable path towards reducing queuing delay in the Internet; unlike active queue management [14, 36] or packet scheduling mechanisms [33, 40], they do not require changes to routers.

There is, however, a major obstacle to deploying delay-controlling protocols on the Internet: their throughput suffers when competing against flows that compete for bandwidth more aggressively (e.g., Cubic [19], NewReno [21], BBR [6], etc.) at a shared bottleneck. For example, a Cubic flow steadily increases its rate in the absence of packet loss or ECN, causing queuing delays to rise; in response to these increasing delays, a competing delay-controlling flow will reduce its rate. The Cubic flow then grabs this freed-up bandwidth. The throughput of the delay-controlling flow plummets, but delays don't reduce.

Is it possible to achieve the benefits of a delay-controlling protocol without lowering throughput? In this paper, we present a practical design to achieve precisely this goal. The key ingredient of our approach is a new method, **Nimbus**, to detect whether competing traffic at a bottleneck link is *elastic* or not using only end-to-end delay and rate measurements. We define a flow to be elastic if it increases its rate when it senses that more bandwidth is available at the shared bottleneck, and decreases it otherwise. All other flows are *inelastic*. Correspondingly, the cross traffic as a whole is elastic if it contains *any* elastic flows, and it otherwise inelastic.

A congestion-controlled flow backlogged at the transport layer is elastic. However, many flows on the Internet (even congestion-controlled flows) are not backlogged; examples include application-limited flows, short TCP flows that fit within the initial congestion window, constant bitrate (CBR) flows, and even video streams when the available bandwidth exceeds the maximum video bitrate. Such flows do not react to changes in available bandwidth and are thus inelastic.

Our key observation is that when Nimbus deems cross traffic to be inelastic, the sender can use a delay-controlling protocol to reduce delays for both the sender and the cross traffic without worrying about losing throughput. Otherwise, it can switch to a *TCP-competitive* protocol like Cubic (or whatever is considered dominant) to compete well without attempting to reduce delays.

Elasticity is a basic property of a backlogged congestion-controlled flow and does not depend on specifics such as its congestion control algorithm or round-trip time (RTT). We use this property to design a robust elasticity detector.A Nimbus sender measures elasticity by modulating its rate with sinusoidal pulses to create small fluctuations in traffic at the bottleneck at a specific frequency (e.g., 5 Hz). Concurrently, it estimates the rate of the cross traffic using measurements of its own send and receive rates, and computes the cross traffic's *frequency response* via the Fast Fourier Transform (FFT) to determine if its rate oscillates at the same frequency. If so, the sender concludes that the cross traffic contains elastic flows; if not, it is inelastic.

Many flows using delay-controlling algorithms are elastic. Thus, in the future, if delay-controlling protocols become widely deployed, Nimbus might miss out on some opportunities to control delays when competing against delay-controlling elastic cross-traffic flows. For example, if an elastic cross-traffic flow uses Copa (a delay-controlling scheme), then in principle it would be possible to achieve low delay and high throughput by also running Copa (or some other protocol compatible with Copa). However, this would require inferring the cross traffic's specific congestion control protocol; simply using a delay-controlling scheme like Vegas against Copa would lead to throughput loss. We sidestep this challenge by focusing on detecting elasticity, which suffices to ensure no throughput loss compared to the prevalent deployed algorithm(s). We leave detecting other properties of cross traffic (like the congestion control protocol), which could expand the set of scenarios where we can reduce delays, to future work.

It is also possible that one or more cross-traffic flows also use Nimbus. Because long-lived Nimbus traffic is elastic, in this case the Nimbus flows would switch to their TCP-competitive modes. To maintain Nimbus flows' ability to use delay-controlling algorithms in scenarios where *all* non-Nimbus traffic is inelastic, we extend Nimbus based on the insight that Nimbus flows can communicate with each other through frequency modulation to collectively determine which mode to operate in.

Key results. We demonstrate the benefits of using elasticity detection for congestion control with NimbusCC, a congestion controller that uses Nimbus to switch between TCP-competitive and delay-controlling modes. We implement NimbusCC using CCP [34] in Linux. NimbusCC can support various protocols in each mode. We report results using Vegas, Copa (default "delay" mode), and BasicDelay (a new method that uses our cross traffic rate estimator), as examples of delay-controlling protocols, and Cubic and NewReno as examples of TCP-competitive protocols. Our experimental results (§7) show that:

(1) Nimbus is robust to a variety of cross traffic conditions, achieving more than 85% detection accuracy even when the cross traffic is a highly dynamic mix of inelastic and elastic flows of different sizes(§7.1), and when it includes multiple flows with different RTTs or congestion control protocols. These results hold across a wide range of buffer sizes, RTTs, bottleneck link rates, active queue management (AQM) schemes, flow sizes, and fractions of cross traffic.
(2) NimbusCC achieves throughput within 10% of the ideal value against elastic traffic made up of a variable number of TCP flows, whereas Copa is 54% lower. NimbusCC also achieves 60 ms lower mean delay than Cubic against Poisson-distributed inelastic cross traffic.
(3) When cross traffic is modeled from a flow-size distribution measured at a WAN link [3], NimbusCC achieves throughput comparable to Cubic and BBR, but with 50 ms lower median delay. Copa has slightly better (5 ms) median delay but achieves 40% lower throughput than NimbusCC and Cubic whenever cross traffic is substantially elastic. Similar results hold when the cross traffic contains elastic flows using different congestion control protocols.
(4) On 25 Internet paths, NimbusCC achieved a throughput at least as high as Cubic with lower delays on 60% of the paths and similar delays on the other 40%. Compared to BBR, NimbusCC's throughput was 10% lower, but the mean packet delay was 40–50 ms lower.

Elasticity detection is a general technique and while we explore its use in congestion control, we believe it is further applicable to other use cases—e.g., for aggregate traffic control between sites [4] and in speed-testing tools to inform users not only of the rate and delay, but also the nature of the cross traffic on particular paths (and hence whether using a different congestion control protocol could improve throughput or delay).

2 Related Work

Copa [1] aims to maintain a bounded number of packets in the bottleneck queue. Copa induces a periodic pattern of sending rate, which nearly empties the queue once every five RTTs. This helps Copa flows obtain an accurate estimate of the minimum RTT and the queuing delay. In addition, Copa uses this pattern to detect the presence of non-Copa flows: it expects the queue to be nearly empty at least once every 5 RTTs, provided only Copa flows with similar RTTs share the bottleneck link. If this does not occur, Copa switches to a TCP-competitive mode.

This method is sensitive to variations in cross traffic (e.g., arrival/departure of flows), the control protocol used by the cross traffic flows, and even their RTTs. For these reasons, we find that Copa suffers from both false positives (increased delay) and, more importantly, false negatives (lower throughput) (see §5, §7.1 and §7.2). Unlike Copa, Nimbus does not look for a specific pattern in the RTTs. Instead it directly estimates elasticity by measuring whether the cross traffic reacts to rate fluctuations over a few seconds in the frequency domain. This method is more robust and can be applied to any combination of TCP-competitive and delay-controlling algorithms, whereas Copa's approach relies on the specific dynamics of its rate controller.

BBR [6] estimates the bottleneck bandwidth b and minimum RTT d. It paces traffic at b while capping the number of in-flight packets to $2\times b\times d$. To estimate the bottleneck, BBR periodically increases its rate over b for about one RTT and then reduces it for the following RTT. BBR uses this sending-rate pattern to obtain estimates of b; specifically, it tests if the observed rate exceeds the current estimate b in the rate-increase phase. However, BBR doesn't use these pulses to infer the nature of cross traffic.

PCC-Vivace [9] uses an online learning algorithm to adapt its sending rate to maximize a utility function that incorporates the achieved rate, delay, and loss rate. Our experiments (§5, §7.1) show that Vivace cannot achieve both low delay with inelastic cross traffic and compete fairly with elastic TCP flows.

3 Cross-Traffic Estimation

We present a simple new method to estimate the total rate of cross traffic at the sender (§3.1). Then, we show how to detect whether the cross traffic contains *any* elastic flows, describing the key principles (§3.2) and a practical method (§3.3).

Figure 1 shows our network model and introduces some notation. A sender communicates with a receiver over a single bottleneck link of rate μ. The bottleneck link is shared with cross traffic, consisting of an unknown number of flows, each of which is either elastic or inelastic. $S(t)$ and $R(t)$ denote the time-varying sending and receiving rates, respectively, while $z(t)$ is the total rate of the cross traffic.

Operating regime. Our technique requires some degree of traffic persistence. The sender must be able to create sufficient pulses and observe the impact on cross traffic over a period of time. Thus, it is best suited for long flows. Fortunately, it is for such transfers that delay-controlling schemes are useful, because short flows are

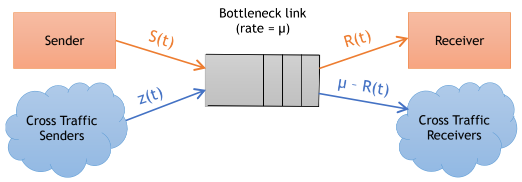


Figure 1: Network model. The time-varying total rate of cross traffic is $z(t)$. The bottleneck link rate is μ. The sender's transmission rate is $S(t)$, and the rate of traffic received by the receiver is $R(t)$.

unlikely to cause significant queueing delay [16]. The detector is designed for a single bottleneck link with a stable rate, and uses a link-rate estimator similar to BBR's. When these conditions do not hold, the detector can become inaccurate and have false positives. Our detector conservatively classifies cross traffic as elastic in these cases (with high likelihood, discussed in §6 and §7.3). When applied to congestion control, the detector will thus choose a TCP-competitive mode in these scenarios. Thus, while such false positives might cause the sender to miss out on opportunities to reduce delay, they will not cause it to lose throughput.

Our technique is most effective when the elastic flows react on a timescale of a few RTTs. If an elastic flow is slower to react, it can go undetected with short pulses. By using long pulses Nimbus can detect such sluggish elastic flows but at the cost of some congestion. Since the majority of elastic traffic on the Internet reacts on RTT timescales (e.g., ACK-clocked flows), we use short pulses. For ease of exposition, we describe Nimbus in the context of detecting ACK-clocked flows, but the technique applies more generally (e.g., correctly classifying fast-reacting rate-based flows).

3.1 Estimating the Rate of Cross Traffic

In Fig. 1, the total traffic into the bottleneck queue is $S(t)+z(t)$, of which the receiver sees $R(t)$. As long as the bottleneck link is busy (i.e., its queue is not empty), and the router treats all traffic the same way, the ratio of $R(t)$ to μ must be equal to the ratio of $S(t)$ and the total incoming traffic, $S(t)+z(t)$.[1] Using this property, we propose the following estimator for $z(t)$:

$$\hat{z}(t) = \mu \frac{S(t)}{R(t)} - S(t). \tag{1}$$

We estimate $S(t)$ and $R(t)$ by considering n packets at a time:

$$S_{i,i+n} = \frac{n_{bytes}}{s_{i+n}-s_i}, \qquad R_{i,i+n} = \frac{n_{bytes}}{r_{i+n}-r_i}, \tag{2}$$

where n_{bytes} is the number of bytes in the n packets, s_k is the time at which the sender sends packet k, r_k is the time at which the sender receives the ACK for packet k, and the units of the rates are bytes per second. $S(t)$ and $R(t)$ must be measured over the *same* n packets.

The above quantities can be calculated using the timestamps of the first and the last packet and hence are unaffected by delayed acknowledgements (delayed ACKs). Our implementation in CCP [34] uses the Linux kernel's measurements of $S(t)$ and $R(t)$ over the last RTT—the same method used by the BBR implementation.

[1]This property holds even when the bottleneck link is dropping packets as long as the drop rate is the same for the sender-to-receiver flow and the cross traffic.

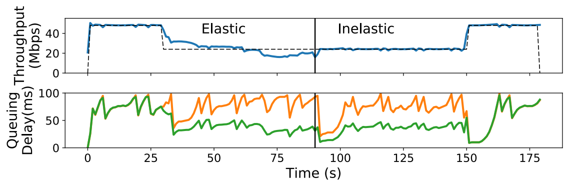


Figure 2: Instantaneous delay measurements do not reveal elasticity. The bottom plot shows the total queueing delay (orange) and the self-inflicted delay (green). The experiment contains one background Cubic flow in the elastic region (30–90 s) and CBR cross traffic in the inelastic region (90–150 s).

Like BBR [6], we use the maximum received rate to estimate μ, taking care to avoid incorrect estimates due to ACK compression.[2]

We have conducted several tests with various patterns of cross traffic to evaluate the effectiveness of this $z(t)$ estimator (including scenarios with packet drops and delayed ACKs). The overall error is small: the 50th and 95th percentiles of the relative error are 1.3% and 7.5%, respectively. Unlike prior work on estimating cross traffic rate [23, 25, 41], our method is in-band and does not use any probe packets; however, it relies on the property that the sender is persistently backlogged.

3.2 Elasticity Detection: Principles

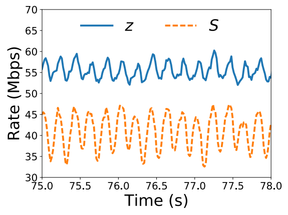
We now turn to designing an online estimator for a sender to determine if the cross traffic includes *any* elastic flows.[3] A strawman approach might attempt to detect elastic flows by estimating the contribution of the cross traffic to queueing delay. For example, the sender can estimate its own contribution to the queueing delay—i.e., the "self-inflicted" delay—and if the total delay is significantly higher than the self-inflicted delay, conclude that the cross traffic is elastic.

This scheme does not work. To see why, consider the experiment in Figure 2, where a Cubic flow shares a link with elastic and inelastic traffic in two separate time periods. The self-inflicted queueing delay for the Cubic flow (green, bottom figure) looks the same in the elastic and inelastic phases. The reason is that a flow's share of the queue occupancy is proportional to its throughput, which is roughly the same in the two phases (top figure). Because the Cubic flow gets 50% of the bottleneck link, its self-inflicted delay is roughly half of the total queueing delay always (orange, bottom figure). This example suggests that instantaneous measurements cannot be used to distinguish between elastic and inelastic cross traffic.

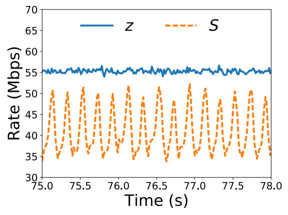
To detect elasticity, tickle the cross traffic! Our method detects elasticity by monitoring how the cross traffic responds to induced traffic variations at the bottleneck link over a period of time. The key observation is that elastic flows react in a predictable way to rate fluctuations at the bottleneck. Consider, for example, long-running Cubic or NewReno flows, which are ACK-clocked. For these flows, if an ACK is delayed by a time duration δ, then the next packet transmission will also be delayed by δ. Therefore changes in the rate of packet arrivals at the receiver cause similar changes in the sending rate after one RTT via the ACKs. By contrast, the sending rate of inelastic flows does not depend on the receive rate.

We induce changes in the inter-packet spacing of cross traffic at the bottleneck link by sending packets in *pulses*. We take the

[2]A variety of other techniques [10, 11, 22, 24, 28, 29, 31] could also be used to estimate μ.
[3]Receiver participation will improve accuracy by avoiding the need to estimate $R(t)$ from ACKs at the sender, but would be a little harder to deploy.



(a) Elastic cross traffic



(b) Inelastic cross traffic

Figure 3: Cross traffic's reaction to pulses. The pulses change the inter packet spacing for cross traffic. Elastic traffic reacts to these changes after a RTT, while inelastic traffic does not.

desired sending rate, $S(t)$, and alternate between sending at rates higher and rates lower than $S(t)$, ensuring that the mean rate is $S(t)$. Sending in such pulses (e.g., modulated on a sinusoid) changes the inter-packet spacing of the cross traffic departing the bottleneck link in a controlled manner. If the cross traffic contains elastic flows, then because of the induced changes in the ACK clocks of those flows, their rates will react to our pulses. When we increase our rate, the elastic cross traffic will reduce its rate in the next RTT, and vice versa. If enough of the cross traffic is elastic, then our sender can measure and detect these fluctuations in the cross traffic rate.

Fig. 3a and Fig. 3b compare the responses of elastic (Cubic) and inelastic (constant bit rate) cross traffic when the sender transmits packets in sinusoisal pulses at frequency $f_p = 5$ Hz. $S(t)$ is the sender's rate and $z(t)$ is the estimated cross traffic rate computed using Eq. (1). The path has a minimum RTT of 50 ms and a buffer size of 100 ms (2× the bandwidth-delay product). The elastic flow's sending rate after one RTT is inversely correlated with the pulses in the sending rate, while the inelastic flow's sending rate is unaffected.

3.3 Elasticity Detection: Practice

To produce a practical method to detect cross traffic using this idea, we must address three challenges:

(1) Pulses in the sending rate must induce a measurable change in z, but not congest the bottleneck link.
(2) Because there is natural variation in cross traffic, and noise in $\hat{z}$, it is not easy to perform a robust comparison between the predicted change in z and the measured z.
(3) Because the sender does not know the RTTs of cross traffic flows, it does not know when to look for the predicted response in the cross traffic rate.

The first method we developed to solve these problems measured the *cross-correlation* between $S(t)$ and $z(t)$. A cross-correlation near zero would be considered inelastic cross traffic, whereas a significant non-zero value would indicate elastic cross traffic. We found that this approach works well (with square-wave pulses) if the cross traffic is substantially elastic and has a similar RTT to the flow trying to detect elasticity, but not otherwise. The trouble is that because elastic cross traffic will react after *its* RTT, $S(t)$ and $z(t)$ must be aligned using the cross traffic's RTT, which is not easy to infer. Moreover, the elastic flows in the cross traffic may have different RTTs, making the alignment even more challenging.

From time to frequency domain. We have developed a method, Nimbus, that overcomes the challenges stated above. It uses two ideas. First, the sender modulates its packet transmissions using *sinusoidal pulses* at a known frequency f_p, with amplitude equal to a modest

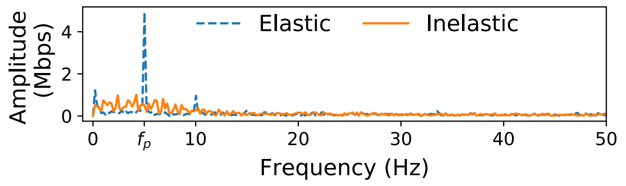


Figure 4: Cross traffic FFT for elastic and inelastic traffic. Only the FFT for elastic traffic has a pronounced peak at f_p (5 Hz).

fraction (e.g., 25%) of the bottleneck link rate. These pulses induce a noticeable change in inter-packet times at the link without causing congestion, because the queues created in one part of the pulse are drained in the subsequent part, and the period of the pulses is short (e.g., $f_p = 5$ Hz). By using short pulses, we ensure that the total burst of data sent in a pulse is a small fraction of the typical bottleneck queue size.

Second, the sender looks for periodicity in the cross traffic rate at frequency f_p, using a frequency domain representation of the cross-traffic rates. We use the Fast Fourier Transform (FFT) of the time series of the cross traffic estimate $\hat{z}(t)$ over a short time interval (e.g., 5 seconds). Detecting periodicity in the frequency domain is more robust than the time-domain, for the same reason that frequency modulation provides better signal-to-noise ratio than amplitude modulation [37]: it is less affected by variations in the cross traffic rate and measurement noise. Further, observing the cross traffic's response at a known frequency, f_p, yields a method that is robust to the presence of multiple elastic flows with different RTTs, and even, different congestion control protocols, because all elastic flows (irrespective of RTT and protocol) will exhibit rate oscillations at the frequency f_p. As a result, there will be an overall response at frequency f_p in the cross traffic, equal to superposition of the responses of the individual elastic flows at frequency f_p.[4]

Fig. 4 shows the FFT of the $\hat{z}(t)$ time-series produced using Eq. (1) for examples of elastic and inelastic cross traffic, respectively. Elastic cross traffic exhibits a pronounced peak at f_p compared to the neighboring frequencies, while for inelastic traffic the FFT magnitude is spread across many frequencies. The magnitude of the peak depends on how much of the cross traffic is elastic; the more elastic the cross traffic, the sharper the peak at f_p. Therefore, rather than compare the peak at f_p to a pre-determined threshold, we compare it to the magnitude of the nearby frequencies.

We define the *elasticity metric*, η, as follows:

$$\eta = \frac{|FFT_z(f_p)|}{\max_{f \in (f_p+\epsilon, 2f_p-\epsilon)} |FFT_z(f)|} \quad (3)$$

Eq. (3) compares the magnitude of the FFT at frequency f_p to the peak magnitude in the range from just above f_p to just below $2f_p$. We use $\epsilon = 0.5$ Hz (with $f_p = 5$ Hz) in our implementation. If η is less than a threshold $\eta_{thresh} (\geq 1)$, then the cross traffic is deemed inelastic; otherwise, it is elastic.

3.4 Setting Parameters for Elasticity Detection

Detection threshold. In practice, cross traffic can be a mix of elastic and inelastic flows. In such scenarios, we want our detector to be sensitive to the presence of any elastic flows, since even one elastic flow

[4]In theory, the response of flows with different RTTs may cancel each other out, but this is very unlikely since it requires specific combinations of RTTs. We have not seen this problem occur in our experiments (§7.2).

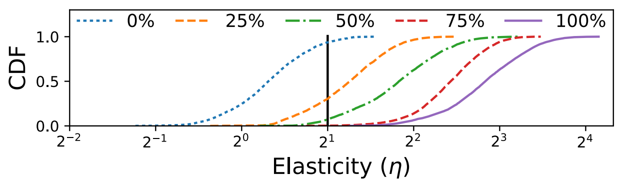


Figure 5: Distribution of elasticity with varying elastic fraction of cross traffic. The cross traffic consists of an elastic Cubic flow and inelastic Poisson-distributed traffic with different rates. Completely inelastic cross traffic has η close to one, while completely elastic cross traffic exhibits a high η. Cross traffic with some elastic fraction also exhibits high elasticity ($\eta > 2$).

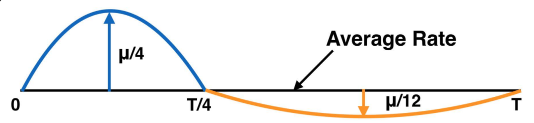


Figure 6: Asymmetric sinusoidal pulse. The pulse has period $T = 1/f_p$. The positive half-sine lasts for $T/4$ with amplitude $\mu/4$, and the negative half-sine lasts for the remaining duration, with amplitude $\mu/12$. The two half-sines cancel out each other over one period.

can eventually grab all the link bandwidth from a delay-controlling flow. Fig. 5 shows the CDF of elasticity (η) as the fraction of bytes belonging to elastic flows in the cross traffic varies. η varies due to variations in the cross traffic but its value generally increases as more of the cross traffic becomes elastic: the median values range from $\eta = 1$ for purely inelastic traffic to $\eta = 10$ for purely elastic traffic.

The value of η_{thresh} dictates which type of traffic is detected more reliably. A large η_{thresh} will ensure that inelastic traffic is always classified correctly, but it increases the chance that cross traffic with a small elastic component is misclassified as inelastic (potentially hurting throughput). With a small η_{thresh}, on the other hand, elastic traffic will be classified correctly, but we may occasionally classify inelastic traffic as elastic, losing an opportunity to reduce delays. To balance these concerns, we choose a small fixed threshold $\eta_{thresh} = 2$, which in Fig. 5 corresponds to classifying cross traffic with a 25% elastic component correctly 75% of the time. We evalutate the impact of η_{thresh} on congestion control performance in D.2.

Pulse shaping. Rather than a pure sinusoid, we use an *asymmetric* sinusoidal pulse, as shown in Fig. 6. In the first one-quarter of the pulse cycle, the sender adds a half-sine of a certain amplitude (e.g., $\mu/4$) to $S(t)$; in the remaining three-quarters of the cycle, it subtracts a half-sine with one-third of the amplitude used in the first quarter of the cycle (e.g., $\mu/12$). The reason for this asymmetric pulse is that it enables senders with low sending rates, $S(t)$, to generate pulses. For example, for a peak amplitude of $\mu/4$, a sender with $S(t)$ as low as $\mu/12$ can generate the asymmetric pulse shown in Fig. 6; a symmetric pulse with the same peak rate would require $S(t) > \mu/4$.

Our pulses produce an observable pattern in the FFT when the cross traffic is elastic. Using asymmetric sinusoidal pulses creates harmonics at multiples of the pulse frequency f_p. However, these harmonics do not affect η (see Eq. (3)), which only uses the FFT in the frequency band $[f_p, 2f_p - \epsilon)$.

Pulse duration. What should the duration, T, of the pulse be? The answer depends on two factors: first, the interval over which S and R are measured (with which the sender computes $\hat{z}$), and second, the amount of data we are able to send in excess of the mean rate without causing congestion. If T were smaller than the measurement interval of S and R, the perturbation to the cross traffic rate during one part of the pulse will be averaged out during the rest of the pulse, resulting in no impact on $\hat{z}(t)$. But T cannot be too large because the sender transmits in excess of the mean rate $S(t)$ for $T/4$. In particular, the size of the burst sent in a pulse is $\frac{2}{\pi}\frac{\mu}{4}\frac{T}{4} = \frac{T\mu}{8\pi} \approx 0.04\mu T$. If T is equal to the RTT, this is 4% of the bandwidth-delay product (BDP) at the peak. Moreover, since pulsing doesn't increase the average sending rate, there is no increase in the average queuing delay (§5).

We set T to a large RTT value observed on the Internet, for example $T = 200$ ms, with the rationale that router buffers are typically provisioned to avoid packet losses for one such RTT, and because our implementation measures S and R over one RTT. We measure rates over one RTT because sub-RTT measurements are confounded by burstiness in packet transmissions (e.g., caused by ACK compression [26]).

If the cross traffic reacts slower than the pulse duration, Nimbus might misclassify those flows. Using longer pulses could improve detection accuracy in such scenarios but it might cause congestion. We evaluate this alternative for detecting PCC-Vivace, a rate-based scheme (not ACK-clocked), in Appendix B.3.

FFT duration. Computing FFTs over a small duration allows quick responses to changes in cross traffic, but it increases errors due to noise. For example, natural variations in inelastic cross traffic over small periods can cause false peaks at f_p in the FFT, resulting in a misclassification. The FFT duration also impacts the frequency resolution of the FFT; in particular, ϵ in Eq. (3) must be larger than 1/`FFT_Duration`. We choose an FFT duration of 5 seconds (corresponding to 25 pulses) and $\epsilon = 0.5$ Hz to balance these concerns. While Nimbus is robust in detecting the presence of long-lived elastic flows, it might still missclassify transient elastic cross-traffic. Typically, elastic flows take multiple RTTs to ramp up their rates and grab bandwidth from competing flows. As a result, even when the Nimbus sender is operating in the delay-controlling mode, such transient elastic traffic is unlikely to adversely affect throughput. In §7.1, we show that even when the cross-traffic is a highly dynamic mix of inelastic and elastic flows of different sizes, Nimbus achieves high detection accuracy and congestion control performance.

4 NimbusCC

NimbusCC is a congestion control system that uses mode switching. It has a TCP-competitive mode in which the sender transmits using a TCP-competitive congestion control algorithm (e.g., Cubic), and a delay-control mode that uses a delay-controlling algorithm (e.g., Copa). NimbusCC switches between the two modes using our elasticity detector, Nimbus.

4.1 Mode Switching

At any given time, NimbusCC transmits data at the time-varying rate dictated by the congestion control algorithm running at that time. It modulates this rate with asymmetric sinusoidal pulses (Fig. 6). NimbusCC uses the pulsing parameters described in §3.4, calculating S and R over one window's worth of packets. It computes the FFT for the z measurements reported in the last 5 seconds to calculate elasticity (η) using Eq. (3), and it picks the mode by comparing η to $\eta_{thresh} = 2$ (§3.4).

We support Cubic and NewReno for the TCP-competitive mode and Copa's default mode and Vegas for the delay-control mode.

We also implemented a simple delay-controlling algorithm, called BasicDelay, which relies on our cross traffic rate estimator to calculate the spare capacity at the sender.

BasicDelay uses a typical control loop inspired by prior explicit control protocols [17, 27, 42]. Let S be the sending rate and $\hat{z}$ be the estimated cross traffic rate, both measured over the last window of packets. Also, let x be the current RTT, and x_{min} be the minimum observed RTT. Upon receiving an ACK, BasicDelay sets its current rate to:

$$\text{Rate} \leftarrow S+\alpha(\mu-S-\hat{z})+\beta\frac{\mu}{x}(x_{\min}+d_t-x), \tag{4}$$

where α and β are constants smaller than 1, and d_t is a target queuing delay. The term $(\mu - S - z)$ is the sender's estimate of the spare capacity in the last RTT. By adding an α-fraction of the spare capacity to $S(t)$, BasicDelay tries to get closer to the ideal rate. The second term in the above rule seeks to maintain a specified queuing delay, d_t, to prevent the queue from both growing too large or going empty. Recall that our cross traffic estimator, Eq. (1), requires a non-empty queue to estimate z.

To safeguard against losing throughput in the first FFT duration (when there is not enough history to determine the elasticity), NimbusCC flows start in the TCP-competitive mode. NimbusCC takes special care in initializing the rate when switching to TCP-competitive mode. NimbusCC sets the rate (and equivalent window) to the rate that was used 5 seconds ago because the elasticity detector takes 5 seconds (FFT Duration) to detect elastic cross traffic. During this time, the elastic traffic could cause a reduction in the delay-control mode's rate. Hence, NimbusCC resets its rate to the rate at the beginning of the 5-second detection period.

4.2 Multiple NimbusCC Flows

What happens when a bottleneck is shared by multiple NimbusCC flows? If all the NimbusCC flows pulse at the same frequency (f_p), then they will all detect a peak in the FFT at that frequency and stay in the TCP-competitive mode (regardless of the other cross traffic). Thus they will achieve the same throughput as the TCP-competitive protocol and compete fairly with each other, but will not maintain low delays when there is no elastic cross traffic.

We can, however, expand the set of scenarios where Nimbus can reduce delays. Ideally, we want all the NimbusCC flows to remain in delay-control mode when there is no elastic cross traffic, and use TCP-competitive mode otherwise. One approach is for different NimbusCC flows to pulse at different frequencies. But this approach cannot scale to more than a few flows, because the set of distinguishable frequencies is limited (recall that the pulse period T cannot be too small).

The pulser and the watchers. We propose a different approach. One of the NimbusCC flows assumes the role of the *pulser*, while the others are *watchers*. They coordinate without explicit communication; each NimbusCC flow is unaware of the identities, or even existence, of the others.

The pulser sends data by modulating its rate with asymmetric sinusoids. The pulser uses two different frequencies, f_{pc} in TCP-competitive mode, and f_{pd} in delay-control mode. The values of these frequencies are fixed and agreed upon beforehand; we use $f_{pc} = 5$ Hz and $f_{pd} = 6$ Hz in our experiments.[5]

[5]These values are in accordance with bounds on T and f described in §4.

A watcher infers whether the pulser is pulsing at frequency f_{pc} or frequency f_{pd} by computing the FFT of its receive rate, R, at these two frequencies. It then picks the mode corresponding to the larger peak to match the pulser's mode. Note that since a watcher is not pulsing, it can detect the pulser's pulses in its own receive rate, R; i.e., it does not even need to estimate z. The pulser, on the other hand, cannot look at its own R to detect pulses in the cross traffic, since it will end up detecting its own pulses.

For multiple NimbusCC flows to maintain low delays during times when there is no elastic cross traffic on the link, the pulser must classify watcher traffic as inelastic. Note that from the pulser's perspective, the watcher flows are part of the cross traffic; thus, to avoid confusing the pulser, the rate of watchers must not react to the pulses of the pulser. To achieve this goal, a watcher applies an exponentially weighted moving average (EWMA) filter to its transmission rate before sending data. The EWMA filter mutes all frequencies in the sending rate that exceed $\min(f_{pc}, f_{pd})$.

Pulser election. A distributed and randomized election decides which flow is the pulser and which are watchers. If a NimbusCC flow determines that there is no pulser (by seeing that there is no peak in the FFT at the two potential pulsing frequencies), then it decides to become a pulser with a probability proportional to its transmission rate:

$$p_i = \frac{\kappa\tau}{\text{FFT Duration}} \times \frac{R_i}{\mu}. \tag{5}$$

Each flow makes decisions periodically, e.g., every $\tau = 10$ ms, κ is a constant, and R_i is the receive rate of the i^{th} flow. This rule ensures that the expected number of flows that become pulsers over the FFT duration is at most κ. To see why, note that the expected number of pulsers is equal to the sum of the probabilities in Eq. (5) over all the decisions made by all flows in the FFT duration. Since $\sum_i R_i \leq \mu$ and each flow makes (FFT Duration$/\tau$) decisions, these probabilities sum up to at most κ.

It is also not difficult to show that the number of pulsers within an FFT duration has approximately a Poisson distribution with a mean of κ [12]. Thus the probability that after one flow becomes a pulser, a second flow also becomes a pulser before it can detect the pulses of the first flow in its FFT measurements is $1-e^{-\kappa}$. Therefore, κ involves a tradeoff: a smaller κ will lead to fewer conflicts but will take longer to elect a pulser. In our experiments, we use $\kappa = 1$.

For any value of κ, there is a non-zero probability of more than one concurrent pulser.[6] In such cases, all the NimbusCC flows will stay in TCP-competitive mode: they could miss opportunities to reduce delay but will *not* lose throughput relative to the status quo. As a further optimization, if there are multiple pulsers, then each pulser will observe that the cross traffic has more variation than the variations it creates with its pulses. This can be detected by comparing the magnitude of the FFT of the cross traffic $z(t)$ at f_p with the FFT of the pulser's receive rate $R(t)$ at f_p. If the cross traffic's FFT has a larger magnitude at f_p, the NimbusCC pulser concludes that there must be multiple pulsers and switches to a watcher with a fixed probability.

Deployment scenarios. This protocol for coordinating multiple NimbusCC flows is only necessary if it is likely that NimbusCC flows will often compete with each other (§6). In other situations, NimbusCC will achieve low delays against inelastic traffic and

[6]In scenarios outside Nimbus's intended operating regime, multiple concurrent pulsers are more likely.

this extension is unnecessary. Nevertheless, recall that to ensure fail-safe operation, NimbusCC starts in TCP-competitve mode before switching to delay-controlling mode if it is safe to do so. If the protocol for multiple NimbusCC flows is needed, then it is reasonable to expect elastic cross-traffic flows to use NimbusCC; thus, in this case most flows would be capable of delay-control, and NimbusCC should start in delay-controlling mode instead.

Remark. This scheme for coordinating pulsers is similar to receiver-driven layered multicast (RLM) congestion control [32]. In RLM, a sender announces to the multicast group that it is conducting a probe experiment at a higher rate, so any losses incurred during the experiment should not be heeded by the other senders. In contrast, in Nimbus, there is no explicit coordination channel, and the pulsers and watchers coordinate via their independent observations of cross traffic patterns. The pulser election also shares similarities with carrier sense multiple access (CSMA) protocols. Similar to a CSMA sender, a watcher looks for the absence of any pulser (free channel) on the shared bottleneck, and switches probabilistically to a pulser to try to avoid multiple pulsers (collisions). However, unlike CSMA protocols, collisions are harder to detect in Nimbus.

5 Visualizing NimbusCC

We illustrate NimbusCC on a synthetic workload with time-varying cross traffic. We emulate a bottleneck link in Mahimahi [35], a link emulator. The network has a bottleneck rate of 96 Mbit/s, a minimum RTT of 50 ms, and 100 ms (2 BDP) of buffering. We compare two mode-switching protocols, NimbusCC (Cubic+BasicDelay) and NimbusCC (Cubic+Copa), with Cubic, BBR, Vegas, and PCC-Vivace (all from Linux), and Copa (from Copa's authors).

The cross traffic varies over time between elastic, inelastic, and a mix of the two. We generate inelastic cross traffic using Poisson packet arrivals at the specified mean rate. Elastic cross traffic uses Cubic, via `iperf` [43].

Fig. 7 shows the throughput and queuing delays for the various protocols, as well as the correct fair-share rate. Table 1 summarizes the deviation from fair-share throughput in the elastic (20–120 s) and inelastic (0–20 and 120–180 s) regions, and the mean queuing delay in the inelastic region. The delay in the elastic region is similar for all schemes.

Throughout the experiment, both NimbusCC variants achieve throughput close to the fair-share rate and low (≤15 ms) queuing delays in the presence of inelastic cross traffic. With elastic cross traffic, both variants switch to TCP-competitive mode within 5 seconds and achieve close to their fair share. The delays during this period approach the buffer size because the competing traffic is buffer-filling; the delays return to their previous low value (15 ms) within 5 seconds after the elastic flows complete. NimbusCC stays in the correct mode throughout the experiment, except for one interval in the elastic period. The deviation from fair-share in the elastic region is because Cubic is not perfectly fair to itself over short time periods.

Cubic achieves its fair-share rate but experiences high delays (80 ms) throughout. BBR's throughput is often much higher than its fair share with high delays even against inelastic cross traffic, which prior work has also observed [1, 20]. Vegas suffers from low throughput in the presence of elastic cross traffic as it reacts to packet delays.

While Copa generally uses the correct mode it frequently switches mode unnecessarily; Copa makes 28 switching errors in the elastic region, while NimbusCC only switches once. In the elastic period,

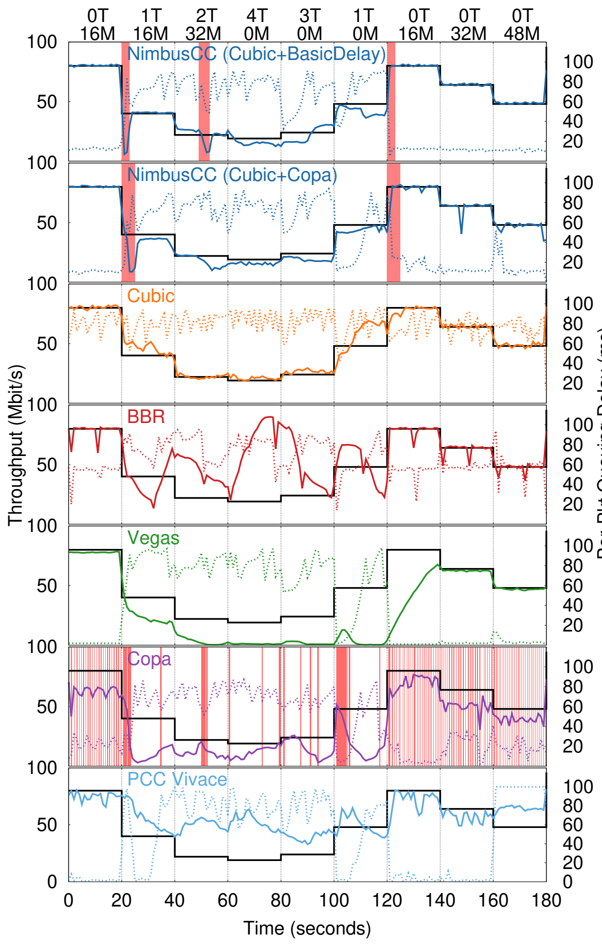


Figure 7: Performance on a 96 Mbit/s Mahimahi link with 50 ms delay and 2 BDP of buffering while varying the rate and type of cross traffic as denoted at the top of the graph. xM denotes x Mbit/s of inelastic Poisson cross traffic. yT denotes y long-running Cubic cross-flows. The solid black line indicates the correct time-varying fair-share rate that the protocol should achieve given the cross traffic. For each scheme, the solid line shows throughput and the dotted line shows queuing delay. The cross traffic contains elastic flows from 20–120 s. For Nimbus and Copa, the red shaded regions indicate times spent in the wrong mode (e.g., delay-controlling with elastic cross traffic).

Copa's frequent mode switches lower its throughput (14 Mbit/s) compared to NimbusCC (27.5 Mbit/s) and fair-share rate (e.g., see 100–120 s). Further, by draining queues periodically, Copa incurs some underutilization against inelastic traffic (e.g., 140–160 s).

Vivace competes unfairly with elastic traffic. At times, Vivace fails to maintain low delays against inelastic cross traffic and incurs heavy packet loss (e.g., 160–180s).

6 Discussion

Rate-based protocols. Table 2 summarizes how Nimbus classifies different types of cross traffic. Recall that our method relies on the cross traffic responding to variations induced by pulses on an RTT timescale. This is true of all ACK-clocked protocols, which are classifed as elastic.

Scheme	Throughput Δ Elastic	Throughput Δ Inelastic	QDelay Inelastic
NimbusCC Cubic+BasicDelay	−10%	0%	12 ms
NimbusCC Cubic+Copa	−15%	−1%	14 ms
Cubic	+12%	0%	78 ms
BBR	+61%	−2%	56 ms
Vegas	−79%	−15%	3 ms
Copa	−54%	−19%	18 ms
PCC-Vivace	+61%	−2%	27 ms

Table 1: Average queuing delay (in ms) in the inelastic region, and deviation from fairshare throughput in elastic and inelastic regions from Fig. 7. NimbusCC is the only scheme to achieve close to fair-share throughput and low delays.

Cross Traffic	Elastic	ACK-Clocked	Classification
Cubic	Yes	Yes	Elastic
NewReno	Yes	Yes	Elastic
Copa	Yes	Yes	Elastic
Vegas	Yes	Yes	Elastic
BBR	Yes	If CWND-limited	Elastic*
PCC-Vivace	Yes	No	Inelastic*
Fixed window	Yes	Yes	Elastic
App. limited	No	No	Inelastic
Const. stream	No	No	Inelastic

Table 2: Classification by Nimbus.

For BBR, recent work has showed that it is ACK-clocked when competing with other flows [44]; Nimbus thus classifies it as elastic. We therefore find that NimbusCC (with Cubic as the TCP-competitive protocol) achieves similar throughput to Cubic when competing against BBR (Appendix §B.2).

Some rate-based protocols may not react on RTT timescales. For example, Nimbus in its default configuration classifies PCC-Vivace as inelastic because it does not react quickly enough to Nimbus's pulses. Increasing the pulse duration helps Nimbus to correctly classify such flows as elastic (Appendix B.3). Increasing the pulse duration, however, may also increase queuing delays. Since most elastic traffic today is ACK-clocked, we use a small pulse duration by default. In the future, if rate-based protocols become widely deployed, the pulse duration could be adjusted accordingly.

Error in link rate estimation. Nimbus requires an estimate of the link rate. If this estimate has too much error, the detector classifies traffic as elastic even if it is inelastic. NimbusCC will then operate in TCP-competitive mode, achieving similar throughput and delay as the status quo. To understand why, define $\hat{z}(t)$ as the estimate of the cross traffic rate, $z^*(t)$ as the actual cross traffic rate, $\hat{\mu}$ as the estimate of the link rate, and μ^* as the actual link rate. Then, from Eq. (1):

$$\hat{z}(t) = \hat{\mu}\frac{S(t)}{R(t)} - S(t), \qquad z^*(t) = \mu^*\frac{S(t)}{R(t)} - S(t) \tag{6}$$

Combining the equations above, we get

$$\hat{z}(t) = \frac{\hat{\mu}}{\mu^*}z^*(t) + \left(\frac{\hat{\mu}}{\mu^*} - 1\right)S(t) \tag{7}$$

When the link rate estimate is inaccurate, $\hat{z}(t)$ is a linear combination of the cross traffic rate and the sending rate. As the error in the link rate estimate increases, the contribution of the sending rate to $\hat{z}(t)$ increases. Since the sending rate oscillates at the pulse frequency, $\hat{z}(t)$ also oscillates, and all cross traffic (regardless of its nature) is classified as elastic.

Time-varying links and multiple bottlenecks. On time-varying links (e.g., wireless links), the elasticity detector cannot obtain an accurate estimate of the link rate and will therefore tend to classify traffic as elastic, for the same reason described above. In scenarios with multiple bottleneck links,[7] the elasticity detector breaks, but again in a predictable way. Similar to the incorrect μ case, the cross traffic estimate is a combination of the actual cross traffic rate and the sending rate in such scenarios, and Nimbus will tend to classify traffic as elastic (see Appendix D.4). We evaluate Nimbus under such conditions in §7.3.

Insufficient share of the bottleneck. To generate pulses, the detector must control a fraction of the traffic at the bottleneck link ($\geq \mu/12$). If the sender's rate is not high enough, NimbusCC does not modulate the sending rate and switches to the TCP-competitive mode to guard against losing throughput from misclassification. Therefore when there are a large number of flows competing at the bottleneck, each with a tiny share of the link bandwidth, NimbusCC is similar to the status quo. Note that in such scenarios, the cross traffic is more likely to be elastic.

Prevalence of inelastic and elastic cross-traffic. NimbusCC's effectiveness in improving delay on Internet paths depends on how often cross traffic is inelastic (and whether elastic cross traffic flows use NimbusCC as in §4.2). While our experiments on paths between different cloud regions and a small number of residential hosts suggest that scenarios where cross traffic is predominantly inelastic might be common (§7.5), a full understanding would require a large-scale measurement study. We do not attempt to provide a verdict on this question.

NimbusCC is most useful if elastic cross-traffic is neither rare or dominant, since it can identify the appropriate operating mode on a flow-by-flow basis. If the majority of cross-traffic is either elastic or inelastic in practice, NimbusCC would mostly operate in the corresponding mode, though its mode switching would still be useful as a safeguard when the cross traffic is not of the predominant class.

7 Evaluation

We have implemented NimbusCC using CCP [34], which provides a convenient way to express the signal processing operations in user-space code. It uses estimates of S, R, the RTT, and packet losses from the Linux kernel every 10 ms.

We evaluate our elasticity detection method and NimbusCC. We use the Mahimahi emulator and investigate the performance benefits (§7.1) of elasticity detection with realistic workloads, its robustness (§7.2), and its behavior in scenarios where the assumptions underlying the method are not met (§7.3). We also evaluate the effectiveness of the pulser-watcher extension (§7.4). Unless specified otherwise, the topology in these experiments consists of a single bottleneck link with a stable link rate and we do not use the pulser-watcher extension. Finally, we evaluate the performance of NimbusCC on real Internet paths (§7.5).

7.1 Performance Benefits from Elasticity Detection

We evaluate the delay and throughput benefits of mode switching via elasticity detection using trace-driven emulation. We generate

[7] We expect such scenarios to be rare, since congestion in the Internet typically occurs at the network edge [15, 30].

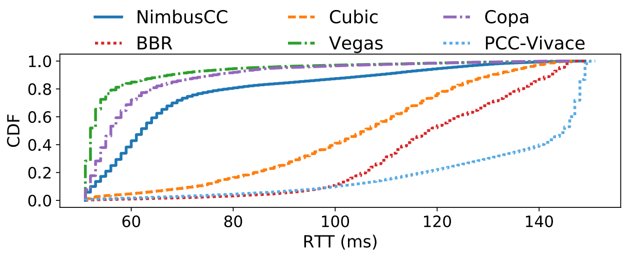
cross traffic from an empirical distribution of flow sizes derived from a wide-area packet trace from CAIDA [3]. This packet trace was collected at an Internet backbone router on January 21, 2016 and contains over 30 million packets recorded over 60 seconds. We generate Cubic cross traffic flows with flow sizes drawn from this data, with flow arrival times generated by a Poisson process to offer a fixed average load to fill 50% of the link (48 Mbit/s). The experiment duration is 360 s and consists of 100,000 cross traffic flows.

The cross traffic consists of a highly dynamic mix of short and long flows, with a heavy-tailed distribution of flow sizes, ranging from 10 KB to 100 MB (average flow size is 22 KB). Very short flows with size less than the initial congestion window (< 15 KB) are inelastic as they transmit all data at once and don't react to the fluctuations in the available bandwidth, whereas long (backlogged) flows are elastic. The traffic trace consists of periods with *a mix of elastic and inelastic cross traffic,* along with periods with only inelastic cross traffic flows. There is high churn in the number of flows, and the cross traffic exhibits periods of high load that span from a few RTTs to several minutes.

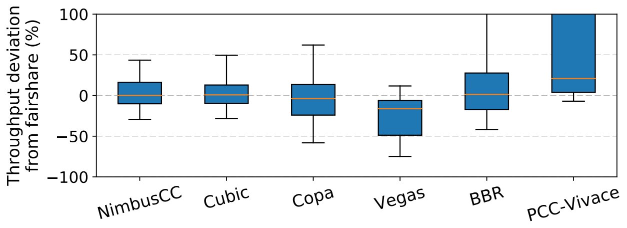
We start one backlogged flow using different congestion control algorithms (NimbusCC, Cubic, Copa, Vegas, PCC-Vivace or BBR) and sharing a 96 Mbit/s bottleneck link with the cross traffic flows. The propagation RTT is 50 ms and the buffer size is 1.2 Mbytes (2 bandwidth-delay products). NimbusCC uses Cubic in the TCP competitive mode and BasicDelay (§4.1) in the delay-controlling mode. For BasicDelay we used $\alpha = 0.8$, $\beta = 0.5$ and $d_t = 12.5$ ms.

NimbusCC reduces delays while achieving fair-share throughput. Fig. 8a shows the distribution of per-packet RTT and Fig. 8b shows the deviation from fair-share throughput (over 5-second intervals) for various schemes. NimbusCC and Cubic achieve the lowest deviation from fair share among these schemes. NimbusCC's deviation profile is comparable to Cubic (note that both NimbusCC and Cubic deviate from the fair share since Cubic is not perfectly fair to itself over short time periods). The reason is that NimbusCC correctly switches to Cubic mode in the presence of elastic flows. Additionally, by switching to delay-controlling mode in the absence of elastic flows, NimbusCC achieves lower RTTs, with a median delay only 10 ms higher than Vegas and >50 ms lower than Cubic and BBR.

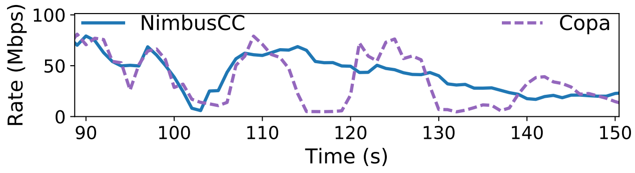
Cost of incorrect mode-switching. Copa has a slightly lower median delay than Nimbus, but at a high cost: its throughput is significantly lower than the fair-share at the 10th and 25th percentiles (corresponding to times with significant elastic traffic). Fig. 8c shows this more clearly, comparing the rates of NimbusCC and Copa during a 60-second interval. Because of the high variations in the cross traffic rate, the queuing delay can drop below Copa's detection threshold even in the presence of elastic flows (e.g., due to departure of other cross flows). Copa often incorrectly operates in its default delay-control mode against elastic cross traffic (e.g., 115–120, 130–140 s). These incorrect switches cause Copa to nearly stop sending. Since the elastic flows competing with Copa in such periods achieve a higher throughput than the same flows against NimbusCC (which attains the fairshare rate), they complete more quickly, freeing up bandwidth that Copa grabs subsequently. This is why in periods like 120–130 s, which immediately follow a low-rate period for Copa, it achieves a higher rate than NimbusCC. Also, since NimbusCC competes fairly with elastic flows rather than yielding bandwidth, it has a slightly higher delay than Copa at the tail.



(a) NimbusCC reduces delay relative to Cubic, BBR and PCC-Vivace. It has higher delays than Copa and Vegas, but those two schemes have lower throughput than the fair share against elastic cross traffic (see figure below).



(b) Deviation in throughput from fair-share. NimbusCC and Cubic achieve highest fairness. Vegas and Copa deviate from fair-share at lower percentiles and lose throughput; BBR and PCC-Vivace are significantly higher than fair share. Midline is the median, the box edges are the 25%ile and 75%ile, the whisker notches are 10%ile and 90%ile.



(c) Copa incorrectly switches to its default delay-control mode even when competing against elastic traffic, unlike NimbusCC.

Figure 8: Performance of NimbusCC on a cross traffic workload derived from a packet trace collected at a WAN router.

While both schemes achieve the same overall average throughput[8], NimbusCC is better suited to applications that value stable bandwidth, e.g., video streaming, interactive web browsing, online gaming, etc. In such applications, sending at a very low rate for several seconds when cross traffic has elastic flows is unacceptable. Note that elastic traffic was present for only about 25% of the duration of this experiment with Copa, which is why we see Copa under-performing only at the lower percentiles.

NimbusCC helps cross traffic. The 95th percentile flow completion time (FCT) of cross traffic flows reduces by 3-4× compared to BBR, and 1.3× compared to Cubic for short (≤ 15 KB) flows (Appendix A). In contrast, PCC-Vivace is unfair to the background flows (positive deviation from fairshare). It grabs significantly more bandwidth than all the other schemes and keeps the buffer near-full more than half the time. The result is that many background flows do not complete, and their completion times are over 100× worse than with other schemes. PCC-Vivace also shows higher delays that any other scheme; the median delay is 90 ms higher than NimbusCC.

[8]Any work-conserving scheme will achieve the same throughput in this experiment because the cross traffic sends a fixed number of bytes.

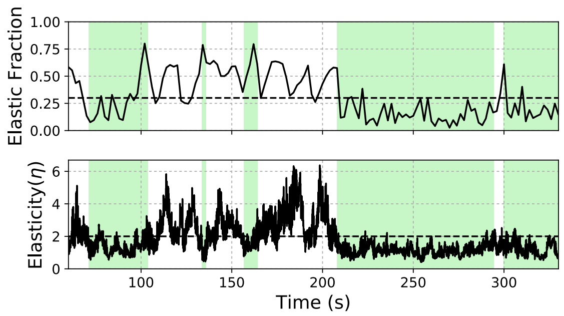


Figure 9: The elasticity metric closely tracks elastic cross traffic (ground truth measured independently from the rate of ACK-clocked flows). Green-shaded regions indicate inelastic periods.

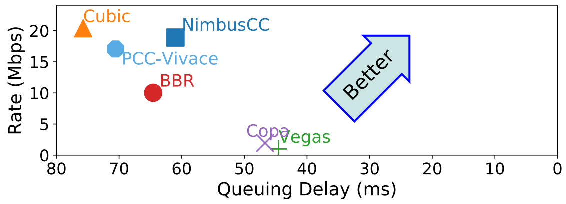


Figure 10: Mean throughput and queuing delay (lower delay on the right) with video cross traffic. NimbusCC achieves similar throughput as Cubic but reduces delays and performs better than the other schemes. Copa and Vegas achieve low throughput.

Elasticity detection is accurate. To define ground truth, we note that short flows (< 10 packets) transmit all data at once, without any rate adjustments. We thus classify a cross traffic flow as elastic if it is larger than the initial congestion window of 10 packets, finishing in greater than a RTT.

The top chart in Fig. 9 shows the fraction of bytes belonging to elastic flows as a function of time. The bottom chart shows the output of the elasticity detector with the dashed threshold line at $\eta = 2$. The shading corresponds to periods when NimbusCC is in delay-control mode. Shaded regions correlate well with the periods when the true fraction of elastic traffic is low (e.g., < 0.3), while white regions correlate well with periods when the elastic fraction is high. Unlike Copa, our elasticity detector observes fluctuations in cross traffic *over a period of time in the frequency domain*, and the accuracy is less susceptible to variations in the cross traffic rate. Despite the churn in cross traffic flows, the overall accuracy of our elasticity detector is over 90% when the fraction of the elastic traffic is high (> 30%). When the fraction of elastic traffic is low, NimbusCC operates primarily in the delay-controlling mode. In this case, the elastic flows in the cross traffic are relatively short. Such short elastic flows do not last long enough to grab bandwidth from the NimbusCC flow.

Performance against different congestion control protocols. We repeat the experiment in Fig. 8 but with cross traffic consisting of an equal (on average) mix of Cubic, NewReno and BBR flows. The results are similar to the previous experiment: NimbusCC achieves lower delays than Cubic for a similar throughput profile, while Copa and Vegas lose throughput when cross traffic is elastic. The reason is that, regardless of the congestion control protocol, the elastic cross traffic flows react to Nimbus's pulses, and can therefore be classified correctly (Appendix B.1).

Performance with video cross traffic. Video streams can be application-limited (e.g., when the client playback buffer is full) or network-limited (e.g., when the client is downloading a high-bitrate chunk) at different points in time. Therefore video traffic can exhibit both inelastic and elastic behavior. We compare the performance of congestion control algorithms running against cross traffic consisting of a 4k DASH [8] video stream using Cubic on a 48 Mbit/s link with 50 ms RTT for 80 seconds. Fig. 10 shows the throughput and delay of the various schemes. Because of effective mode switching, NimbusCC achieves similar throughput as Cubic at 15 ms lower delay. Nimbus recognizes application-limited video traffic as inelastic, allowing the sender to control delays in those cases; it rarely recognizes network-limited elastic traffic as inelastic, so does not wrongly reduce its rate as Copa does. Note that the figure shows the rate of a backlogged flow competing against video cross traffic; the total link utilization with all the schemes was at least 90%.

7.2 Robustness of Elasticity Detection

We evaluate the robustness of Nimbus under a variety of network and traffic conditions. Unless specified otherwise, we run NimbusCC as a backlogged flow on a 96 Mbit/s bottleneck link with a 50 ms propagation RTT and a 100 ms drop-tail buffer (2 BDP). We consider three categories of synthetic cross traffic sharing the link with NimbusCC: (i) inelastic Poisson-distributed traffic; (ii) fully elastic traffic (backlogged NewReno flows); and (iii) an equal mix of inelastic and elastic traffic. The duration of each experiment is 120 seconds. We evaluate *accuracy*: the fraction of time Nimbus correctly detects the presence of elastic cross traffic. For each experiment, we report the mean accuracy of the detector across 5 runs.

Impact of cross traffic RTT. We vary the cross traffic's minimum RTT from 10 ms to 200 ms (0.2 – 4× NimbusCC's RTT). We find that varying cross traffic RTT does not reduce accuracy. For purely inelastic and purely elastic traffic, the accuracy is more than 98% in all cases, while for mixed traffic, the accuracy is more than 85% in all cases (a random guess would have only achieved 50%). Regardless of the cross traffic RTT, the elastic flows respond to fluctuations created by Nimbus, generating a peak in the cross traffic FFT at the oscillation frequency. The cross traffic's RTT *affects the phase, but not the amplitude* of the peak in the FFT.

A mix of RTTs in the cross traffic. We vary the number of elastic cross traffic flows from 1 to 5, where the RTT of n^{th} flow in $20 \cdot n$ ms. In case the cross traffic contains elastic flows, all the elastic flows oscillate at Nimbus's pulse frequency. As a result, the sum of the rates of these elastic flows also oscillates,[9] and the traffic is correctly classified as elastic. For purely elastic and inelastic traffic, Nimbus achieves an average accuracy of 98% across 5 runs, while for mixed traffic, the mean accuracy is greater than 90% in all cases. In other words, heterogeneity in RTTs of cross-flows does not degrade the accuracy of elasticity detection.

Pulse size, link rate, and offered cross traffic load. We perform a multi-factor experiment varying Nimbus's pulse size from 1/16 to 1/2 the link rate, the fair share of the bottleneck link rate from 12.5%—75% (by varying the cross traffic load), bottleneck link rates set to 96, 192, and 384 Mbit/s. The accuracy for purely elastic cross traffic is always higher than 95%. while the average accuracy over all the points for the other two traffic mixes is more than 90%. Fig. 11

[9]Since the RTTs are different, the elastic flows' oscillations will differ in phase and the oscillations could in theory cancel each other out leading to mis-classification, but it requires specific combinations of RTT and is unlikely.

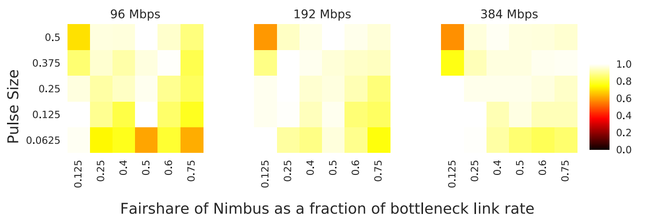


Figure 11: Nimbus is robust to variations in link bandwidth and fraction of traffic controlled by it. The accuracy is high even when the fraction of traffic under control is small. Increasing pulse size increases robustness.

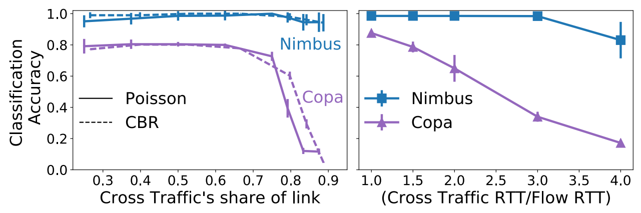


Figure 12: Nimbus is more accurate than Copa when (i) inelastic cross traffic occupies a large fraction of the link (left); (ii) elastic cross traffic has higher RTT than the flow's RTT (right).

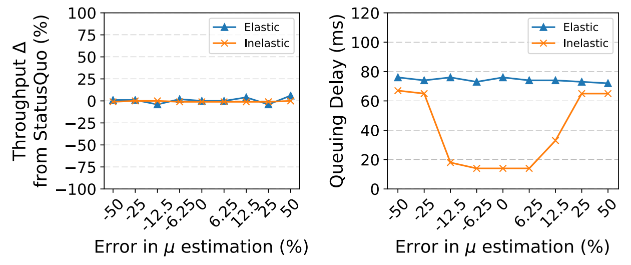
shows the average detection accuracy over the other two categories of cross traffic (mix + purely inelastic). The classification accuracy is not sensitive to cross traffic load. Nimbus's *use of asymmetric pulses* enables a sender to create fluctuations in the cross traffic even when the sending rate is low. As a result, the detection accuracy remains high under high cross traffic load.

In general, increasing the pulse sizes improves accuracy because the elasticity detector can create a more easily observable change in the cross traffic sending rates. An increase in the link rate results in higher accuracy for a given pulse size and Nimbus link share because the variance in the rates of inelastic Poisson cross traffic reduces with increasing cross traffic sending rate, reducing the number of false peaks in the cross traffic FFT. However, the elasticity detector has low accuracy (~60%) when it uses high pulse sizes and controls a low fraction of the link rate. We believe that this is due to a quirk in the way the Linux networking stack reports round-trip time measurements under sudden sending rate changes.

Comparison with Copa. We now compare the classification accuracy of Nimbus with Copa. First, we generate inelastic cross traffic at different rates and measure the accuracy. We consider both constant-bit-rate (CBR) and Poisson cross traffic.

Fig. 12 (left) shows that Nimbus has high accuracy in all cases, but Copa's accuracy drops sharply when the cross traffic occupies over 80% of the link. This result highlights a pitfall of Copa's approach: setting an operating mode based on the absolute value of queueing delays is problematic. With a high inelastic cross traffic load, Copa is unable to drain the queue quickly enough (i.e., every 5 RTTs), which throws off its detector. In contrast, the elasticity detector estimates elasticity through delay variations caused by its pulses, and is more robust.

Next, we ran a backlogged NimbusCC or Copa flow competing against a backlogged NewReno flow. We vary the RTT of the NewReno flow between $1-4\times$ the RTT of the NimbusCC/Copa flow. Fig. 12 (right) shows that Copa's accuracy degrades as the RTT of the cross traffic increases; Nimbus's accuracy is much higher, dropping only slightly when the cross traffic RTT is $4\times$ larger than NimbusCC.



(a) Throughput Δ **(b) Queuing Delay**

Figure 13: Impact of incorrect μ. When the error is high, all the traffic is classified as elastic and NimbusCC operates in TCP-competitive mode.

An elastic cross-flow with a large RTT increases its rate slowly enough to evade detection by Copa. Therefore, Copa drains the queue as it expects and concludes the absence of non-Copa cross traffic. This behavior continues until the cross-flow has grown to offer a load close to the link rate, when it starts interfering with Copa's queue draining. By contrast, Nimbus is more robust since it is based on the time series of variations of the cross traffic rate. Moreover, even when the classification accuracy for Copa is higher, it makes frequent mode-switches and is suspectible to lose throughput against elastic traffic. Appendix C shows the throughput and queueing delay dynamics of Copa and NimbusCC.

Further robustness results and impact of parameters. In Appendix D.1, we explore variations in buffer size, RTT of the Nimbus flow, and presence of active queue management schemes, and we show that Nimbus is robust to these settings. In Appendix D.2, we evaluate the sensitivity of Nimbus's performance to the detection threshold parameter (η_{thresh}). We repeat the experiment in Fig. 8 for different values of η_{thresh} (from 1 to 6), and find that although the performance is similar for a range of values, increasing η_{thresh} generally reduces delays but can cause NimbusCC to lose throughput against elastic cross traffic. In Appendix D.3, we demonstrate the versatility of NimbusCC in supporting different combinations of algorithms for its delay-controlling and TCP-competitive modes.

7.3 Performance Outside Intended Operating Regime

The elasticity detector is designed for a single bottleneck link with stable rate and relies on an estimate of the bottleneck link rate. What happens when these conditions do not hold? As explained in §6, NimbusCC is designed to classify the cross traffic as elastic in such scenarios, thereby switching to the TCP-competitive protocol and achieving similar throughput to the status quo. In this section, we evaluate how effective this mechanism is by comparing the throughput of NimbusCC with the baseline TCP-competitive protocol (referred to as "status quo") in several settings, and also report the per-packet queuing delay. We evaluate the performance for two cross traffic scenarios: (i) inelastic Poisson-distributed traffic, and (ii) fully elastic traffic (backlogged NewReno flows).

Error in link rate estimation. We explicitly supply an incorrect link rate estimate to NimbusCC. We vary the error in the link rate estimate from −50% to +50% of the real link rate value (96 Mbit/s). Fig. 13 reports the average results across 5 runs (120 s each). NimbusCC achieves throughput similar to status quo in all the scenarios. The classification accuracy is high (> 95%) for elastic cross traffic in all

Cross Traffic	Throughput Δ	Q Delay	Classification Accuracy
Elastic	-1%	103 ms	95%
Inelastic	-1%	70 ms	39%

Table 3: Performance on time-varying links.

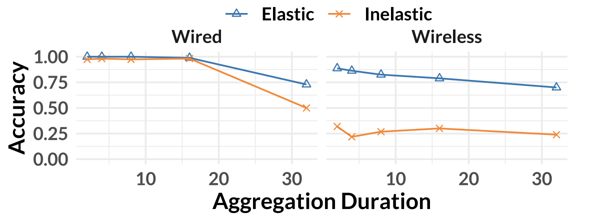


Figure 14: Classification accuracy degrades at higher aggregation durations.

the cases. When the error rate is high (> 12.5%), inelastic cross traffic is also classified as elastic and NimbusCC fails to control delays.

Performance on time-varying links. The experiment consists of a single bottleneck with a time-varying link rate. We model the rate of the bottleneck link as a random walk; the rate can change by ± 20 Mbit/s every second. Table 3 summarizes the results across 5 such traces. NimbusCC achieves throughput within 1% of the status quo. However, the classification accuracy is low against inelastic traffic and the queuing delay is high (though no higher than the status quo). On time-varying links, inferring the bottleneck link rate is hard in an end-to-end manner [17, 18]. When NimbusCC's estimate of the link rate[10] differs substantially from the bottleneck link rate, the cross traffic estimator fails, and NimbusCC simply operates in the TCP-competitive mode regardless of the cross traffic (§6). Thus, on time-varying bottleneck links NimbusCC is safe to run and will not lose throughput relative to the status quo, but it might lose opportunities to control delays.

Links with frame aggregation. We evaluate NimbusCC's performance on links with frame aggregation in two scenarios (Figure 14). In the first scenario (left), the path of the NimbusCC flows consists of a fixed capacity wired bottleneck link and a non-bottleneck link that emulates wireless frame aggregation by aggregating packets in a certain duration and delivering them in a burst. We vary the aggregation duration from 2 to 32 ms and evaluate the classification accuracy against elastic and inelastic traffic in each case. We find that classification accuracy is high for small values of aggregation duration. When the aggregation duration is increased to 32 ms, the noise in the FFT causes the detection accuracy to drop drastically. In the second scenario (right), the bottleneck link is the same time-varying link from the previous experiment. Here, we find that accuracy of Nimbus is governed by errors in link rate estimation and Nimbus tends to classify all traffic as elastic. However, we find that when aggregation duration is large, the classification accuracy for elastic traffic is also poor, which could degrade throughput.

Multiple bottleneck links. The topology consists of two links. Link 1's bandwidth is 192 Mbit/s and link 2's bandwidth is 96 Mbit/s. The experiment consists of a single NimbusCC flow going through the two links. Each link has either elastic or inelastic cross traffic (a cross traffic flow only traverses one of the two links). NimbusCC achieves throughput comparable to the status quo (within 15%) in all the cases (see Appendix D.4).

[10]We use the average bandwidth as the link rate estimate for these experiments.

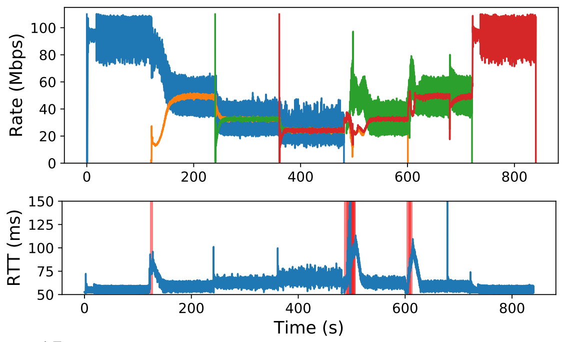


Figure 15: Multiple competing NimbusCC flows. Multiple NimbusCC flows achieve fair sharing of a bottleneck link (top graph). There is at most one pulser flow at any time; identified by its rate variations. Together, the flows achieve low delays by staying in delay mode for most of the duration (bottom graph). The red background shading shows when a NimbusCC flow was (incorrectly) in competitive mode

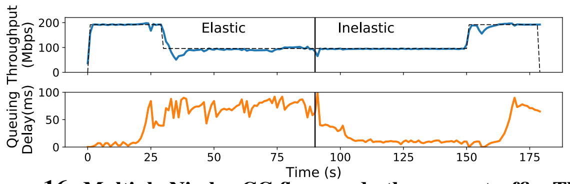


Figure 16: Multiple NimbusCC flows and other cross traffic. There are 3 NimbusCC flows throughout. Cross traffic in 30-90 s is elastic and made up of 3 Cubic flows. Cross traffic in 90-150 s is inelastic and made up of a 96 Mbit/s constant bit-rate stream. NimbusCC flows achieve their fair share (top) while achieving low delays in the absence of elastic cross traffic (bottom).

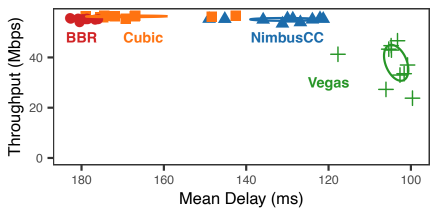
7.4 Multiple NimbusCC

We now evaluate performance when multiple NimbusCC flows share a bottleneck link in scenarios within Nimbus's operating regime.[11] We run NimbusCC with Vegas as its delay-control algorithm. Fig. 15 demonstrates how NimbusCC flows react as other NimbusCC flows arrive and leave (there is no other cross traffic). Four flows arrive at a link with rate 96 Mbit/s and round-trip time 50 ms. Each flow begins 120 s after the last one began and lasts for 480 s. The top half shows the rates achieved by the four flows over time. Each new flow begins as a watcher. If the new flow detects a pulser ($t = 120$, 240, and 360 s), it remains a watcher. If the pulser goes away or a new flow fails to detect a pulser, one of the watchers becomes a pulser ($t = 480$ and 720 s). The pulser can be identified visually by its rate variations.

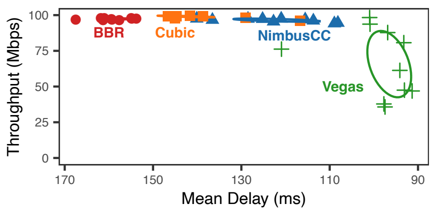
The flows share the link equally. The bottom half of the figure shows the achieved delays with red shading to indicate when one of the flows is (incorrectly) in TCP-competitive mode. The flows maintain low RTTs and stay in delay-control mode most of the time.

Fig. 16 demonstrates multiple NimbusCC flows switching in the presence of cross traffic. We run three NimbusCC flows on an emulated 192 Mbit/s link with a propagation delay of 50 ms. In the first 90 s, the cross traffic is elastic (three Cubic flows), and for the rest of the experiment, the cross traffic is inelastic (96 Mbit/s constant bit-rate). The top graph shows the total rate of the three NimbusCC flows, along with a reference line for the fair-share rate of the aggregate. The graph at the bottom shows the measured queuing delays.

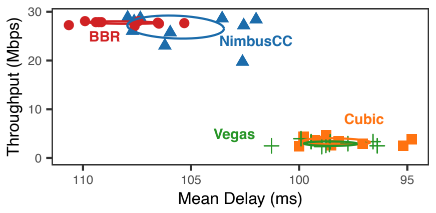
[11]In other scenarios, it is possible for multiple pulsers to coexist. In such cases, multiple NimbusCC will not be able to maintain low delays.



(a) EC2 California to Host A

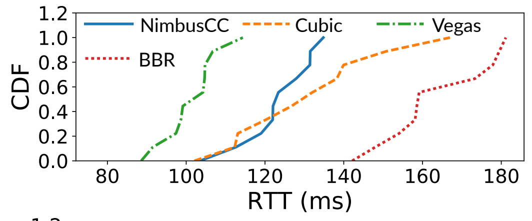


(b) EC2 Ireland to Host B



(c) EC2 London to Host C

Figure 17: Performance on three example Internet paths. The x axis is inverted; better performance is up and to the right. On paths with buffering and no drops, ((a) and (b)), NimbusCC achieves the same throughput as BBR and Cubic but reduces delays significantly. On paths with significant packet drops (c), Cubic suffers but NimbusCC achieves high throughput.



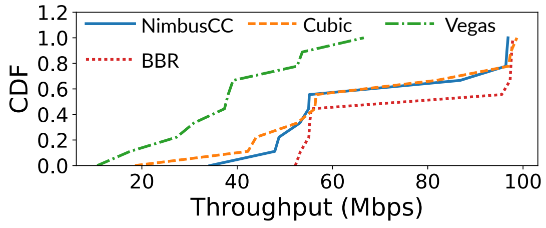


Figure 18: Paths with queuing. NimbusCC reduces the RTT compared to Cubic and BBR (upto 50ms), at similar throughput.

NimbusCC shares the link fairly with other cross traffic, and achieves low delays by staying in the delay-controlling mode in the absence of elastic cross traffic for most of the experiment.

Impact of κ. Increasing κ reduces the time to elect a pulser, but can lead to multiple pulsers. We evaluate the impact of κ in Appendix D.5.

7.5 Testing on Internet Paths

We ran NimbusCC on 25 paths between five senders and five receivers. The servers were EC2 instances located in California, London, Frankfurt, Ireland, and Paris, all with 10 Gbit/s access links.[12] The receivers were residential hosts, each connected directly to its Internet router via a 1 Gbit/s Ethernet link. We verified that the bottleneck in each case was not either access link on the path.

We initiated bulk data transfers using NimbusCC, Cubic, BBR, and Vegas. We ran one-minute experiments over five hours on each path, and measured the throughput and mean packet delay. Fig. 17 shows the throughput and mean delays over three of the paths. The x (delay) axis is inverted; thus, better performance is up and to the right. NimbusCC achieves high throughput comparable to BBR in all cases, but with lower packet delays. Cubic attains high throughput on paths with deep buffers (Fig. 17a and Fig. 17b), but not on paths with packet drops or policers (Fig. 17c).[13] Vegas attains poor throughput on these paths because it does not keep the bottleneck link busy and is unable to compete with elastic cross traffic. These trends show the utility of elasticity detection on Internet paths: it is possible to achieve high throughput and low delays over the Internet using delay-control algorithms with the ability to switch to a different competitive mode when required.

Fig. 18 summarizes the results on paths with larger buffers. NimbusCC's throughput is similar to Cubic's and 10% lower than BBR's but with much lower delays (40–50 ms lower than BBR). NimbusCC's lower mean delay indicates that the cross traffic at the bottleneck link often did not contain long backlogged elastic flows. We believe that during these periods, the cross traffic was application-limited (e.g., video streams where the available bandwidth exceeded the maximum video bitrate). It is in such cross-traffic scenarios that NimbusCC provides the most benefits in terms of delay reduction while still achieving high throughput.

[12]We also ran experiments between pairs of cloud servers but we observed no congestion on any such path.

[13]For each path, we also ran experiments in the night (when the cross traffic load was likely close to 0) and compared the throughput of Cubic and BBR. On the paths where the Cubic throughput was lower consistently across runs, we observed frequent packet drops without much variation in RTT. We inferred that the drops either occur at a shallow-buffered bottleneck link or a policer, both of which are known to hurt the throughput of Cubic [6, 13].

8 Conclusion

We showed that characterizing the elasticity of cross traffic is a useful building block for improving congestion control. We introduced Nimbus, a method for detecting and quantifying the elasticity of cross traffic. Nimbus uses asymmetric sinusoidal pulses to modulate the sending rate and observes the frequency response of the cross traffic rate, taking advantage of the property that elastic cross traffic can be made to oscillate at a pulsing frequency set by sender. Nimbus relies only on end-to-end rate and delay measurements and requires no changes to the routers. We presented several experiments to demonstrate the robustness and accuracy of our proposed method. We also showed that elasticity detection enables transport protocols to combine the best aspects of delay-control methods while being competitive with elastic flows when necessary.

This work does not raise any ethical issues.

Acknowledgements

We thank Justine Sherry for helpful comments and Deepti Raghavan for helping with experiments. This work was supported by NSF grants 1407470, 1563826, 1526791, 2006346, 2006827, and DARPA contract HR001117C0048.

References

[1] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *NSDI*.
[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*.

[3] CAIDA. 2016. The CAIDA Anonymized Internet Traces 2016 Dataset - 2016-01-21. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[4] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Site-to-site Internet Traffic Control. In *EuroSys*.

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR Congestion Control. https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-02.pdf.

[6] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-Based Congestion Control. *CACM* (Jan. 2017).

[7] Jon Crowcroft and Philippe Oechslin. 1998. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM CCR* 28, 3 (July 1998), 53–69.

[8] DASH Industry Forum. 2019. Dynamic Adaptive Streaming over HTTP. https://github.com/Dash-Industry-Forum/dash.js.

[9] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*.

[10] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. 2001. What do Packet Dispersion Techniques Measure?. In *INFOCOM*. IEEE.

[11] Allen B Downey. 1999. Using pathchar to Estimate Internet Link Characteristics. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. ACM, 241–250.

[12] Willliam Feller. 2008. *An Introduction to Probability Theory and its Applications*. Vol. 2. John Wiley & Sons.

[13] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An Internet-Wide Analysis of Traffic Policing *(SIGCOMM)*. https://doi.org/10.1145/2934872.2934873

[14] Sally Floyd and Van Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking* 1, 4 (1993), 397–413.

[15] Daniel Genin and Jolene Splett. 2013. Where in the Internet is Congestion? *arXiv preprint arXiv:1307.3696* (2013).

[16] Jim Gettys and Kathleen Nichols. 2011. Bufferbloat: Dark Buffers in the Internet. *ACM Queue* 9, 11 (2011), 40.

[17] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2020. ABC: A Simple Explicit Congestion Controller for Wireless Networks *(NSDI)*. https://www.usenix.org/conference/nsdi20/presentation/goyal

[18] Prateesh Goyal, Mohammad Alizadeh, and Hari Balakrishnan. 2017. Rethinking Congestion Control for Cellular Networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, Sujata Banerjee, Brad Karp, and Michael Walfish (Eds.). ACM, 29–35. https://doi.org/10.1145/3152434.3152437

[19] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review* 42, 5 (July 2008), 64–74.

[20] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental Evaluation of BBR Congestion Control. In *ICNP*.

[21] J. C. Hoe. 1996. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*.

[22] Ningning Hu and Peter Steenkiste. 2002. *Estimating Available Bandwidth Using Packet Pair Probing*. Technical Report. DTIC Document.

[23] Ningning Hu and Peter Steenkiste. 2003. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE JSAC* 21, 6 (2003), 879–894.

[24] Van Jacobson. 1997. Pathchar: A Tool to Infer Characteristics of Internet Paths.

[25] Manish Jain and Constantinos Dovrolis. 2002. Pathload: A Measurement Tool for End-to-End Available Bandwidth *(PAM)*.

[26] Hao Jiang and Constantinos Dovrolis. 2003. Source-level IP Packet Bursts: Causes and Effects. In *IMC*.

[27] Dina Katabi, Mark Handley, and Chalrie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*.

[28] Kevin Lai and Mary Baker. 2000. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *ACM SIGCOMM Computer Communication Review*, Vol. 30. ACM, 283–294.

[29] Kevin Lai and Mary Baker. 2001. Nettimer: A Tool for Measuring Bottleneck Link Bandwidth.. In *USITS*, Vol. 1. 11–11.

[30] Jacob B Malone, Aviv Nevo, Jonathan W Williams, et al. 2016. *The Tragedy of the Last Mile: Congestion Externalities in Broadband Networks*. Technical Report.

[31] BA Mar. 2000. pchar: A tool for measuring internet path characteristics. *http://www. employees. org/~bmah/Software/pchar/* (2000).

[32] Steven McCanne, Van Jacobson, and Martin Vetterli. 1996. Receiver-driven Layered Multicast. In *SIGCOMM*.

[33] Paul E McKenney. 1990. Stochastic Fairness Queueing. In *INFOCOM*.

[34] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *SIGCOMM*.

[35] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*.

[36] Rong Pan, P. Natarajan, C. Piglione, M.S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. 2013. PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem. In *Intl. Conf. on High Performance Switching and Routing (HPSR)*.

[37] Theodore S Rappaport et al. 1996. *Wireless Communications: Principles and Practice*. Vol. 2. Prentice Hall.

[38] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. 2010. LEDBAT: The New BitTorrent Congestion Control Protocol. In *ICCCN*.

[39] M. Sridharan, K. Tan, D. Bansal, and D. Thaler. 2008. *Compound TCP: A New TCP Congestion Control for High-Speed and Long Distance Networks*. Technical Report. Internet-draft draft-sridharan-tcpm-ctcp-02.

[40] Ion Stoica, Scott Shenker, and Hui Zhang. 2003. Core-Stateless Fair Queueing: a Scalable Architecture to Approximate Fair Bandwidth Allocations in High-Speed

Networks. *IEEE/ACM Trans. Netw.* 11, 1 (2003), 33–46. https://doi.org/10.1109/TNET.2002.808414

[41] Jacob Strauss, Dina Katabi, and Frans Kaashoek. 2003. A Measurement Study of Available Bandwidth Estimation Tools. In *IMC*.

[42] C.H. Tai, J. Zhu, and N. Dukkipati. 2008. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*.

[43] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. 2005. Iperf: The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects.

[44] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's Interactions with Loss-Based Congestion Control. In *IMC*. https://doi.org/10.1145/3355369.3355604

[45] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. 2006. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking* 14, 6 (2006), 1246–1259.

[46] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*.

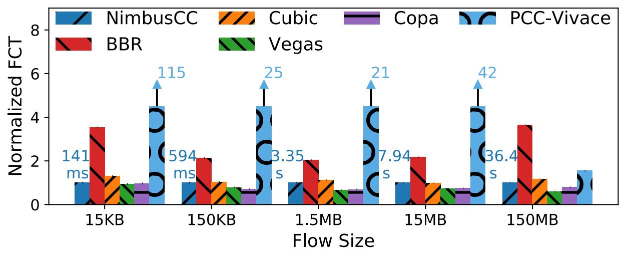


Figure 19: **Using NimbusCC reduces the p95 FCT of cross-flows relative to BBR at all flow sizes, and relative to Cubic for short flows. Vegas provides low cross-flow FCT, but its own rate is low.**

Appendices are supporting material that has not been peer-reviewed.

A Nimbus Helps Cross Traffic

In the setup from §7.1, we measure the flow completion time (FCT) of cross traffic flows. Fig. 19 compares the 95th percentile (p95) FCT for flows of different sizes. The FCTs are normalized by the corresponding value for NimbusCC at each flow size (i.e., NimbusCC is always 1).

BBR and PCC-Vivace exhibits much higher FCT at all cross traffic flow sizes compared to the other protocols, consistent with the unfairness seen in the experiment in §5.

For small flows (≤15 KB), the p95 FCT with NimbusCC and Copa are comparable to Vegas and lower than Cubic. With NimbusCC, p95 FCT of cross traffic at higher flow sizes are slightly lower than Cubic because of small delays in switching to TCP-competitive mode. At all flow sizes, Vegas provides the best cross traffic flow FCTs, but its own flow rate is dismal; Copa is more aggressive than Vegas but less than NimbusCC, but at the expense of its own throughput (§7.1).

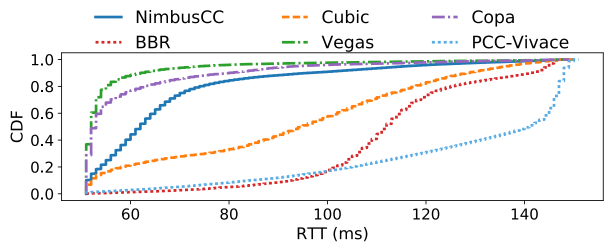
B Cross-traffic Congestion Control Protocols

B.1 Multiple Elastic Flows using different Congestion Control Protocols.

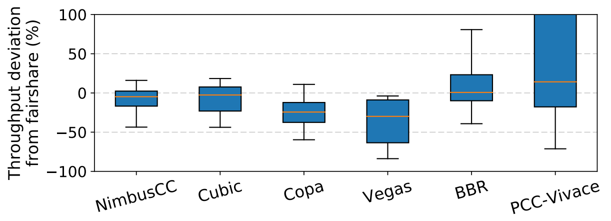
We repeat the experiment in Fig. 8 but with cross traffic consisting of an equal (on average) mix of Cubic, NewReno and BBR flows. Whenever a new cross traffic flow starts, with an equal probability it chooses one of the three congestion control protocols. Fig. 20 shows performance of various schemes. The results are similar to the experiment in Fig. 8: NimbusCC achieves lower delays than Cubic for a similar throughput profile, while Copa and Vegas lose throughput when cross traffic is elastic. The reason is that, regardless of the congestion control protocol, the elastic cross traffic flows react to Nimbus's pulses, and can therefore be classified correctly.

B.2 NimbusCC & Cubic v. BBR

We now evaluate how well a NimbusCC (Cubic + BasicDelay) flow competes with a BBR flow. In this experiment, the cross traffic is 1 BBR flow and the bottleneck link bandwidth is 96 Mbit/s. We vary the buffer size from 0.5 BDP to 4 BDP. Fig. 21 shows the mean throughput of NimbusCC and Cubic flows while competing with BBR over a 2-minute experiment. NimbusCC achieves same throughput as Cubic for all buffer sizes, which matches Cubic's expected behavior against BBR [5].



(a) Per-packet RTT



(b) Deviation from fairshare throughput

Figure 20: **Performance with WAN cross traffic consisting of an equal mix of Cubic, NewReno and BBR flows. The deviation profile of NimbusCC is similar to that of Cubic, however, NimbusCC reduces delays.**

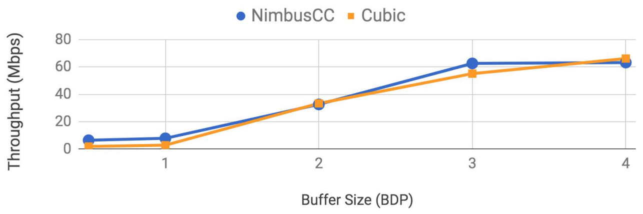


Figure 21: **NimbusCC's performance against BBR is similar to that of Cubic. Both NimbusCC and Cubic compete against 1 BBR flow on a 96 Mbit/s link. For various buffer sizes, NimbusCC achieves the same throughput as Cubic.**

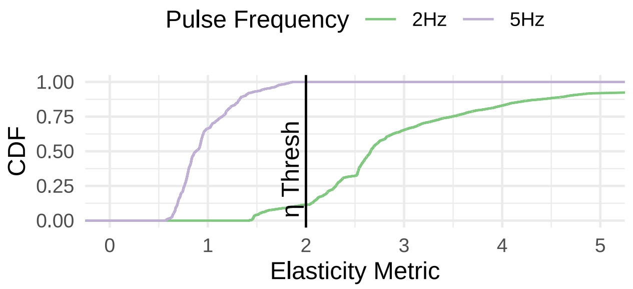


Figure 22: **By modifying the pulse frequency, Nimbus correctly classifies PCC-Vivace, a rate-based elastic protocol, as elastic.**

B.3 Elastic Flows, No ACK Clocking

Nimbus aims to detect ACK-clocked elastic flows that react quickly to changes in available bandwidth on RTT timescales. This experiment demonstrates Nimbus's ability to also detect slow-reacting elastic cross traffic by tuning the pulse frequency. We ran a NimbusCC flow against a PCC-Vivace flow on a 96 Mbit/s link with 100ms of buffering. Fig. 22 shows the CDF of the elasticity metric, η, for two different pulse frequencies, f_p. PCC-Vivace is not ACK-clocked and does not react to Nimbus's pulses at $f_p = 5$ Hz. As a result η is below the threshold most of the time. Reducing the pulse frequency to 2 Hz creates pulses with a longer duration. PCC-Vivace reacts to these slower variations in available bandwidth, and is correctly classified as elastic ($\eta > \eta_{thresh}$).

Changing the pulse frequency involves a trade-off. Increasing the pulse duration will increase queuing delays and congestion. But if slowly-reacting elastic protocols become widely deployed, competing with them using Nimbus for delay-control opportunities will require an increase in pulse duration.

C Copa Mode Switching Errors

We explore the dynamics of NimbusCC and Copa's mode switching in experiments from the scenarios in §7.2.

C.1 CBR Cross Traffic

Fig. 23 shows throughput and delay profile for Copa and NimbusCC while competing against inelastic CBR traffic. We consider two scenarios: (i) CBR occupies a small fraction of the link (24 Mbits/s, 25%) and (ii) CBR occupies majority of the link (80 Mbit/s, 83%). When the CBR traffic is low (Fig. 23 a and Fig. 23 b), both Copa and Nimbus identify it as non-buffer-filling and inelastic, respectively, and achieve low queuing delays.

When the CBR's share of the link is high (Fig. 23 c), Copa incorrectly classifies the cross traffic as buffer-filling and stays in competitive mode, leading to high queuing delays. Copa relies on a pattern of emptying queues to detect whether the cross traffic is buffer-filling or not. However, when the rate of cross traffic is z, the fastest possible rate at which the queue can drain is $\mu - z$, even if Copa reduces its rate to zero. If the cross traffic occupies x fraction of the link (i.e., $z = x\mu$), then

$$\max(-\frac{dQ}{dt}) = \mu - z = (1-x)\mu = (1-x)\frac{BDP}{RTT}. \tag{8}$$

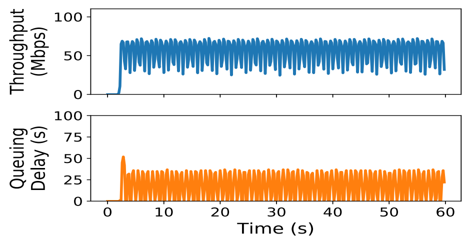
Hence, if the queue size exceeds $5 \times (1-x)BDP$, Copa won't be able to drain the queue in 5 RTTs, and it will mis-classify the cross traffic as buffer-filling. The queue size can grow large due to a transient burst or if Copa incorrectly switches to competitive mode. Once Copa is in competitive mode, it will drive the queues higher, and may get stuck in that mode.

Nimbus doesn't rely on emptying queues and correctly classifies cross traffic as inelastic, achieving low delays (Fig. 23 d).

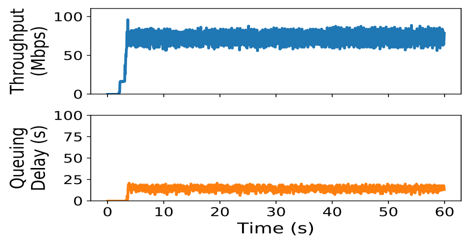
C.2 Elastic Cross Traffic

Fig. 24 shows throughput and delay over time for Copa and NimbusCC while competing against an elastic NewReno flow. We consider two scenarios: (1) both flows have the same propagation RTT, and (2) the cross traffic's propagation RTT is 4× higher than the Copa or NimbusCC flow. When the RTTs are the same (Fig. 24 a and Fig. 24 b), both Copa and Nimbus correctly classify the cross traffic, achieving their fair share.

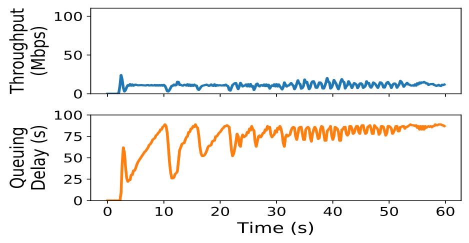
When the cross traffic RTT is higher (Fig. 24 c), NewReno ramps up its rate slowly, causing Copa to mis-classify the traffic and achieve less than its fair share. Here, Copa achieves 27 Mbit/s but its fair share is at least 48 Mbit/s (in fact, 77 Mbit/s considering the RTT bias). In contrast, (Fig. 24 d), Nimbus correctly classifies the cross traffic as elastic, and NimbusCC achieves its RTT-biased share of throughput.



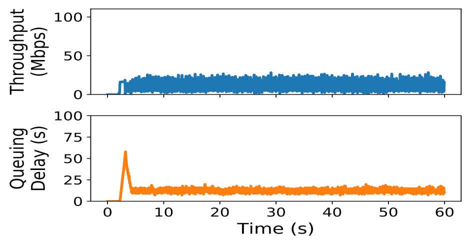
(a) Copa: 24 Mbit/s CBR



(b) NimbusCC: 24 Mbit/s CBR



(c) Copa: 80 Mbit/s CBR



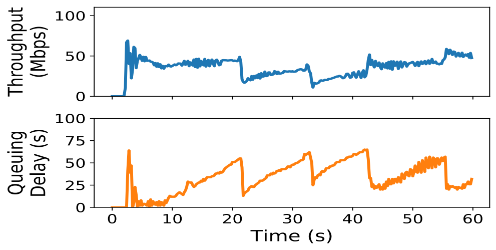
(d) NimbusCC: 80 Mbit/s CBR

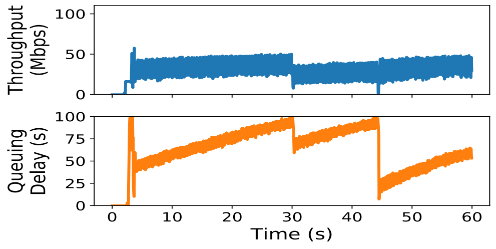
Figure 23: When the CBR traffic is low (a), Copa classifies the traffic as non buffer-filling and is able to achieve low queuing delays. But when the CBR traffic occupies a high fraction (c), Copa incorrectly classifies the traffic as buffer-filling, resulting in higher queuing delays. In both the situations (b and d), the elasticity detector correctly classifies the traffic as inelastic and NimbusCC achieves low queuing delays.

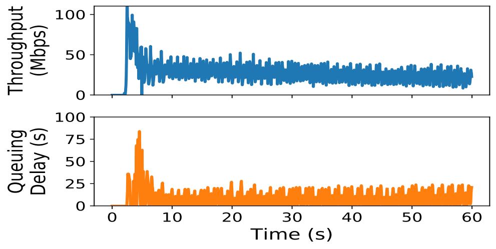
D Other Results

D.1 Buffer size, RTT, and AQM

We vary the bottleneck drop-tail buffer size from 0.25 BDP to 4 BDP for three categories of cross traffic as in the earlier experiments, with propagation delays of 25 ms, 50 ms, and 75 ms. We also measured classification accuracy when the bottleneck link implements PIE [36] at two target delays (0.25 BDP and 1 BDP) with a propagation delay of 50 ms. With purely elastic or inelastic traffic, Nimbus has a mean accuracy (across five runs) of 98% or more in all cases but two, while with mixed traffic, the accuracy is always 85% or more. In all cases


(a) Copa: Cross Traffic RTT = 1 × Flow RTT


(b) NimbusCC: Cross Traffic RTT = 1 × Flow RTT


(c) Copa: Cross Traffic RTT = 4 × Flow RTT

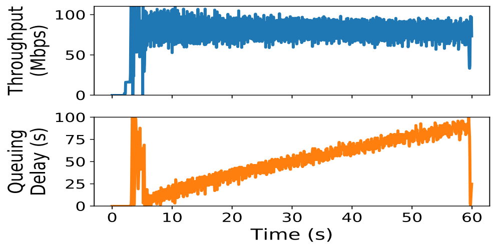
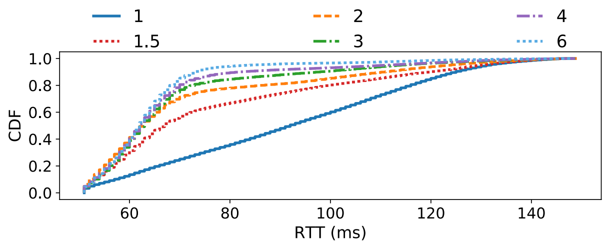

(d) NimbusCC: Cross Traffic RTT = 4 × Flow RTT

Figure 24: Queuing delay and throughput dynamics for elastic cross traffic. When the elastic cross traffic increases fast enough (a), Copa classifies it as buffer-filling and is able to achieve its fair share. But when the elastic cross traffic increases slowly (c), Copa incorrectly classifies the traffic as non-buffer-filling, achieving less than its fair share. In both the situations (b and d), Nimbus correctly classifies the traffic as elastic and NimbusCC achieve its fair share.

(including low accuracy ones), NimbusCC achieves its fair-share throughput and low delays.

Now we discuss the cases with low classification accuracy. First, with shallow buffers of size less than the product of the delay threshold x_t and the bottleneck link rate (e.g., 0.25 BDP when the round-trip time is 50 ms), Nimbus classifies all traffic as elastic. Second, with the bottleneck link implementing PIE with small target delay (e.g., corresponding to 0.25 BDP), Nimbus classifies all traffic as elastic. In both cases, NimbusCC can incur heavy losses in delay-control mode as NimbusCC's target queuing delay of 0.25 BDP


(a) Per-packet RTT

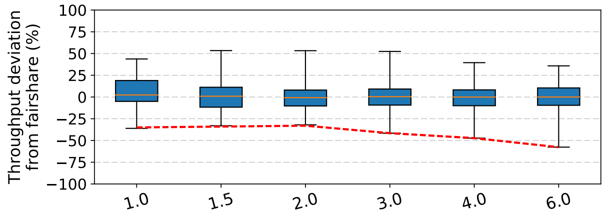

(b) Deviation from fairshare throughput

Figure 25: Impact of η_{thresh}. With a high η_{thresh}, NimbusCC operates in delay-controlling mode more often, reducing delays but losing throughput against elastic cross traffic (see the 10th percentile in the throughput profile, shown in red)

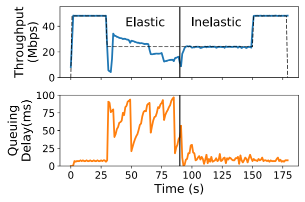
is comparable to the drop-tail buffer size or target delay of PIE. These losses interfere with the cross traffic estimator leading to classification errors (in delay-control mode). However, low accuracy does not impact the performance of NimbusCC as it achieves its fair-share throughput and low delays (bounded by the small buffer size for a drop-tail queue and the delay control threshold of PIE). Further, classification accuracy decreases when Nimbus's RTT exceeds its pulse period. Since Nimbus's measurements of rates are over one RTT, any oscillations over a smaller period cannot be observed.

D.2 Impact of η_{thresh}

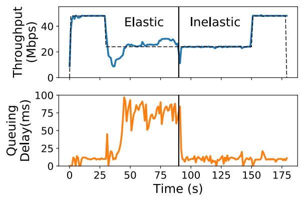
We repeat the experiment in Fig. 8 but vary the detection threshold from 1 to 6. Fig. 25 shows the performance as a function of η_{thresh}. η_{thresh} presents a performance trade-off. With a high η_{thresh}, Nimbus classifies traffic as inelastic more frequently. NimbusCC operates in delay-controlling mode a higher fraction of the time reducing delays. However, at times NimbusCC operates in the delay-controlling mode incorrectly against elastic cross traffic, losing throughput. This affect can be seen prominently at the lowest percentiles in the throughput profile. Similarly, a small η_{thresh} causes NimbusCC to miss opportunities for controlling delays against inelastic traffic, but NimbusCC doesn't lose throughput against elastic traffic.

D.3 Using different CC Algorithms with NimbusCC

NimbusCC can employ a variety of congestion control algorithms for its delay-controlling and TCP-competitive modes. We have implemented Cubic, NewReno, and MulTCP [7] as competitive-mode algorithms, and BasicDelay, Vegas, FAST [45], and COPA [1] as delay-controlling algorithms. In Fig. 26, we illustrate two combinations of delay and competitive mode algorithms sharing a bottleneck link with synthetic elastic and inelastic cross traffic active at different periods during the experiment. The fair-share rate over time is shown as a reference. Both NewReno+BasicDelay



(a) NewReno + BasicDelay



(b) Cubic + COPA

Figure 26: NimbusCC's versatility. NimbusCC with different combinations of delay-controlling and TCP-competitive algorithms.

(Fig. 26a) and Cubic+COPA (Fig. 26b) achieve their fair-share rate while keeping the delays low in the absence of elastic cross traffic.

D.4 Multiple Bottleneck Links

In this section we analyze how multiple bottleneck links affect the Nimbus elasticity detector. Consider a scenario where a NimbusCC flow is going through two bottleneck links, with rates μ_1^* and μ_2^*, in series. Let $z_1^*(t)$ and $z_2^*(t)$ be the cross traffic rate on the two links respectively. Let $\hat{z}(t)$ be the cross traffic estimate, and $\hat{\mu}$ be the bottleneck link rate estimate provided to the elasticity detection algorithm. We define $R_1(t)$ and $R_2(t)$ as the rate at which Link 1 and Link 2 dequeue packets from the NimbusCC flow respectively. Assuming that both links are fully utilized, we have:

$$R_1(t) = \frac{\mu_1^* \cdot S(t)}{z_1^*(t) + S(t)},$$
$$R_2(t) = \frac{\mu_2^* \cdot R_1(t)}{z_2^*(t) + R_1(t)} = \frac{\mu_1^* \cdot \mu_2^* \cdot S(t)}{z_1^*(t) \cdot z_2^*(t) + z_2^*(t) \cdot S(t) + \mu_1^* \cdot S(t)}. \quad (9)$$

Since the receive rate of the NimbusCC flow is $R_2(t)$, the cross traffic estimate given by Eq. (1) is:

$$\hat{z}(t) = \hat{\mu} \cdot \frac{S(t)}{R_2(t)} - S(t) = \hat{\mu} \cdot \frac{z_1^*(t) \cdot z_2^*(t)}{\mu_1^* \cdot \mu_2^*} + S(t) \cdot \left(\frac{\hat{\mu} \cdot z_2^*(t)}{\mu_1^* \cdot \mu_2^*} + \frac{\hat{\mu}}{\mu_2^*} - 1 \right). \quad (10)$$

The cross traffic estimate is thus a combination of the real cross traffic rate and the sending rate. Since the sending rate component oscillates at the pulse frequency, Nimbus will detect oscillations at the pulsing frequency in $\hat{z}(t)$ and classify cross traffic as elastic.

Experiment. We evaluate NimbusCC on a topology with multiple bottleneck links. The topology consists of two links. Link 1's bandwidth is 192 Mbit/s and link 2's bandwidth is 96 Mbit/s. The experiment consists of a single NimbusCC flow going through the two links. The propagation RTT is 50ms. Each link has either elastic or inelastic cross traffic (a cross traffic flow only traverses one of the two links). Depending on the instantaneous rate of the cross traffic at each link, the bottleneck could either be both links or one of the links (the bottleneck can change within an experiment). Table 4 shows the throughput delta relative to status quo and the total queuing delay across both links, averaged over 5 runs of each scenario. NimbusCC achieves throughput comparable to the status quo (within 15%) in all the cases. In scenarios where either of the links contained elastic cross traffic NimbusCC stayed in the TCP-competitive mode majority of the time (> 85%). When both links had inelastic cross traffic, NimbusCC uses the delay-controlling mode and is able to reduce delays; note that in this case the slower link (link 2) is the bottleneck.

Cross Traffic (Link 1)	Cross Traffic (Link 2)	Throughput Δ	Q Delay
Elastic	Elastic	-3%	146 ms
Inelastic	Elastic	5%	76 ms
Elastic	Inelastic	-14%	91 ms
Inelastic	Inelastic	0%	14 ms

Table 4: Performance on a topology with multiple bottlenecks.

κ	Time to elect a pulser	Fraction of time with multiple pulsers
0.5	19.8 s	0%
0.75	15.3 s	4.6%
1	8.0 s	9.3%
1.5	5.4 s	15.4%
2	2.7 s	29.3%

Table 5: Impact of κ.

D.5 Impact of κ

We evaluate the impact of κ on pulser election. In the experiment eight NimbusCC flows start simultaneously, the bottleneck link is 96 Mbits/s and the base RTT is 50ms. We report the time it takes to elect a pulser and the fraction of time there were multiple pulsers. Table 5 summarizes the average values across 20 runs (45 s each). As expected, increasing κ reduces the time to elect a pulser, but also increases the chances of multiple pulsers being elected.