由于使用基于时延的拥塞控制算法的流在与使用基于丢包的拥塞控制算法的流的竞争中，性能表现较差，本文提出了一种能够在两种拥塞控制算法间进行切换的拥塞控制协议Nimbus，该拥塞控制协议通过探测与其竞争的流的属性，来在两种工作模式间进行切换。

Elasticity Detection: A Building Block for Internet Congestion Control

Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Srinivas Narayana,
Mohammad Alizadeh, Hari Balakrishnan
MIT CSAIL
Email: nimbus@nms.csail.mit.edu

问题：本文提出在两种操作模式间进行转换，当竞争流是elastic流时，采取竞争模式，否则采取delay-control模式，那么当竞争流同时包含这两种流时，应该如何取舍呢？

Abstract— This paper develops a technique to detect whether the cross traffic competing with a flow is elastic or not, and shows how to use the elasticity detector to improve congestion control. If the cross traffic is elastic, i.e., made up of flows like Cubic or NewReno that increase their rate when they perceive available bandwidth, then one should use a scheme that competes well with such traffic. Such a scheme will not be able to control delays because the cross traffic will not cooperate to maintain low delays. If, however, cross traffic is inelastic, then one can use a suitable delay-controlled algorithm.

Our elasticity detector uses an asymmetric sinusoidal pulse pattern and estimates elasticity by computing the frequency response (FFT) of the cross traffic estimate; we have measured its accuracy to be over 90%. We present the design and evaluation of Nimbus, a congestion control protocol that uses the elasticity detector to switch between delay-control and TCP-competitive modes. Our results on emulated and real-world paths show that Nimbus achieves throughput comparable to or better than Cubic always, but with delays that are much lower when cross traffic is inelastic. Unlike BBR, Nimbus is fair to Cubic, and has significantly lower delay by 40-50 ms. Compared to Copa, which also switches between a delay-controlling and a TCP-competitive mode, Nimbus is more robust at correctly detecting the nature of cross traffic, and unlike Copa, it is usable by a variety of delay-based and TCP-competitive methods.

1 Introduction

Achieving high throughput and low delay has been a primary motivation for congestion control research for decades. Congestion control algorithms can be broadly classified as *loss-based* or *delay-based*. Loss-based methods like Cubic [13], NewReno [15], and Compound [30, 29] reduce their window only in response to packet loss or explicit congestion notification (ECN) signals, whereas delay-based algorithms like Vegas [3], FAST [33], LEDBAT [27], Sprout [34], and Copa [1] reduce their rates as delays increase to control packet delays and avoid "bufferbloat" [12].

There is, however, a major obstacle to deploying delay-based algorithms on the Internet: their throughput is dismal when competing against loss-based senders at a shared bottleneck. The reason is that loss-based senders increase their rates until they observe losses, which causes queuing delays to increase; in response to increasing delays, a competing delay-based flow will reduce its rate, hoping to reduce delay. The loss-based flow then uses this freed-up bandwidth. The throughput of the delay-based flow plummets, but delays don't reduce. Because most traffic on the Internet today uses loss-based algorithms, it is hard to justify deploying a delay-based scheme.

Is it possible to achieve the low delay of delay-based algorithms whenever possible, while ensuring that their throughput does not degrade in the presence of schemes like Cubic or NewReno? This question has received some attention recently, with Copa [1] proposing an algorithm with two modes—a "default" mode that controls delay but which competes poorly against schemes like Cubic, and a "TCP-competitive" mode that has no delay control but that competes more aggressively. Copa uses round-trip time (RTT) observations to determine if the RTT time-series is consistent with only other Copa flows sharing the bottleneck (Copa periodically empties network queues); if so, the sender uses the delay-controlled mode, but if not, the sender switches to a TCP-competitive mode.

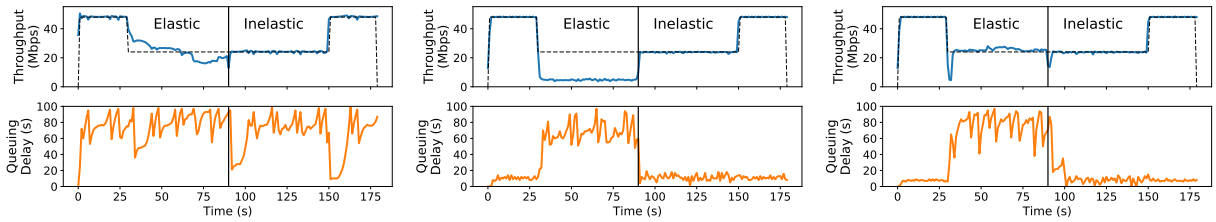
We introduce a new, more robust, approach to set the best operating mode. Our method explicitly characterizes whether the competing cross traffic is *elastic* or not. An elastic sender (or flow) is one that uses feedback from the receiver to increase its rate if it perceives that there is more available bandwidth, and slows down if it perceives an increase in cross traffic (e.g., by being "ACK clocked", by reacting to packet losses, etc.). Examples include backlogged Cubic, NewReno, Compound, and BBR [5] flows. When the cross traffic is elastic, delay-based control is susceptible to poor throughput. To cope, it is important to detect this situation and switch to a mode that mimics the behavior of the prevailing cross traffic.

By contrast, inelastic senders do not attempt to extract all available bandwidth, and run in open-loop fashion independent of cross traffic; examples include short TCP connections, application-limited flows, constant bit-rate streams, and flows that are rate-limited by an upstream bottleneck. When the cross traffic is inelastic, delay-based control can achieve high throughput while controlling delays.

Approach. We have developed an elasticity detector that uses only end-to-end observations to monitor the cross traffic. The sender continuously modulates its rate to create small traffic fluctuations at the bottleneck at a specific frequency (e.g., 5 Hz). It concurrently estimates the cross traffic rate using RTT samples

本文算法如何实现？以正弦函数来控制自己的流的速率，以此观察竞争流在这段时间的变化，计算出竞争流波动的频率，根据竞争流频率的变化来判断竞争流的elasticity，以此来切换流的工作模式（基于时延的或基于竞争的）

在该实验中，分别将cubic、delay-control、Nimbus流在相同的网络环境中进行实验，上图的吞吐量表示的是该流的吞吐量。可以看到cubic的时延非常高，因为它占据了队列。而delay-control的时延较低，但是它无法在与elastic流的竞争中胜出。最后Nimbus就很厉害。



(a) Cubic: High delay (b) Delay-control (§4.1): Throughput drops (c) Nimbus: Lower delay

Figure 1: **Example of mode switching.** In this experiment, we compare Cubic, a delay-control algorithm (§4.1), and Nimbus, which uses mode switching. In each experiment, the flow shares a 48 Mbit/s bottleneck link with one elastic long-running Cubic flow for 60 seconds (starting at $t=30$ sec.), followed by 60 seconds of inelastic traffic sending at 24 Mbit/s. Cubic has a large queuing delay throughout. The delay-control scheme achieves low delay when the cross traffic is inelastic but suffers significant throughput loss when it competes with Cubic. Using mode switching, Nimbus achieves the fair throughput against elastic traffic *and* low queuing delay when the cross traffic is inelastic.

and its own send and receive rates, and observes if the cross traffic fluctuates at the same frequency. If it does, then the sender concludes that it is elastic; otherwise, it is inelastic.

This technique works well because elastic and inelastic flows react differently to short-timescale traffic variations at the bottleneck. In particular, most elastic TCP flows are ACK clocked; therefore, fluctuations in the inter-packet arrival times at the receiver, reflected in ACKs, cause similar fluctuations in subsequent packet transmissions. By contrast, the rate of an inelastic flow is not sensitive to traffic variations at the bottleneck.

We present the design and evaluation of *Nimbus*, a congestion control protocol that uses our elasticity detector to switch between delay-control and TCP-competitive modes. Unlike Copa, Nimbus supports different algorithms in each mode. We report results with Vegas [3], Copa's "default mode" [1], FAST [33], and our Nimbus delay-control method as delay-based algorithms, and Cubic, Reno [15], and MulTCP [6] as TCP-competitive algorithms. Fig. 1 shows an example result with Nimbus.

We have implemented the elasticity detector and Nimbus in Linux using the congestion control plane [24]. Our experimental results show that:

1. On an emulated bottleneck link (96 Mbit/s, 50 ms delay, 100 ms buffering) with a WAN cross-traffic trace, Nimbus achieves throughput comparable to BBR and Cubic but with a significantly smaller (50 ms) median delay.
2. Nimbus's reduction in overall delays benefits the cross-traffic significantly: tail flow completion times for cross-traffic sharing the link with Nimbus are 3–4× smaller than with BBR both for short (< 15 KB) and long (> 150 MB) flows, and 1.25× smaller than Cubic for short flows.
3. Compared to Copa, Nimbus accurately classifies inelastic cross traffic whereas Copa's accuracy is always lower than 80%; Copa's classifier also fails when the inelastic cross traffic volume exceeds 80%. Moreover, with elastic cross traffic, Copa's accuracy degrades from 85% to 15% as the RTT ratio between the cross traffic and reference flow increases from 1 to 4. By contrast Nimbus's accuracy is close to 100%, degrading to 90% only when the RTT ratio is 4:1.
4. Our elasticity detection method detects the presence of elastic cross-traffic correctly more than 90% of the time across a wide range of network characteristics such as cross-traffic RTTs, buffer size, Nimbus RTTs, bottleneck link rates, and the share of the bottleneck link rate controlled by Nimbus.
5. On Internet paths, Nimbus achieves throughput comparable to or better than Cubic on most of the 25 real-world paths we tested, with lower delays in 60% of paths and similar delays in the other 40% of paths. Compared to BBR, Nimbus achieves 10% lower mean throughput, but at 40-50 ms lower packet delay.

2 Related Work

The closest previous schemes to Nimbus are Copa [1] and BBR [5]. These schemes also periodically modulate their sending rates, but they do not infer the elasticity of cross traffic.

Copa. Copa aims to maintain a bounded number of packets in the bottleneck queue. Copa's control dynamics induces a periodic pattern of sending rate that nearly empties the queue once every 5 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: Copa expects the queue to be nearly empty at least once every 5 RTTs if only Copa flows with similar RTTs share the bottleneck link.[1] If the estimated queuing delay does not drop below a threshold in 5 RTTs, Copa switches to a TCP-competitive mode.

Unlike Copa, Nimbus does not look for the expected queue dynamics caused by its transmission pattern. Instead, it estimates the rate of the cross traffic using end-to-end rate and delay measurements, and it observes how the cross traffic reacts to the rate fluctuations it induces over a period of time. This enables Nimbus to directly estimate the elasticity of cross traffic. Although elasticity detection takes longer (e.g., a few seconds), our experiments show that is significantly more robust than Copa's method. For example, we find that Copa misclassifies cross traffic when the inelastic volume is high, or when it is

[1]Copa estimates if a queue is "nearly empty" using the observed short-term RTT variation (see §2.2 of [1]).

本文的假设：链路速率已知，然而这个假设在大多数时候并不成立

elastic but slowly increasing (§6.2). Moreover, since Nimbus's cross traffic estimation technique does not rely on low-level properties of the method's dynamics, it can be applied to any combination of delay-control and TCP-competitive control rules.

BBR. BBR maintains estimates of the bottleneck bandwidth (b) and minimum RTT (d). It paces traffic at a rate b while capping the number of in-flight packets to $2 \times b \times d$. 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 accurate estimates of b using the maximum achieved delivery rate; specifically, it tests for higher b in the rate-increase phase and subsequently drains the extra queuing this causes in the rate-decrease phase.

BBR does not compete fairly with other elastic TCP flows (e.g., Cubic). In particular, BBR's method to set b can be overly aggressive in the presence of other TCP flows. Depending on the bottleneck buffer size, either the BBR-induced losses limit the throughput of the other TCP flows (with shallow buffers), or BBR's hard cap on its in-flight data based on d causes it to get lower throughput than its fair share (with deep buffers).

Other related schemes. PCC [7] adapts the sending rate based on "micro-experiments" that evaluate how changing the sending rate (multiplicatively) impacts performance according to a specified utility function. PCC defines two utility functions: one targeting high throughput and low loss, and the other targeting low delay. Recently, PCC-Vivace [8] improved on PCC with a utility function framework and a rate control algorithm based on regret minimization. The behavior of these schemes depends on the utility function; unlike Nimbus, they do not achieve both low delay and compete well with elastic loss-based TCPs simultaneously. Vegas [3] and FAST [33] are delay-based algorithms that aim to maintain small queues at the bottleneck using different control rules. Other delay-based algorithms include TCP Nice [32] and LEDBAT [28], which aim to use spare bandwidth without hurting "foreground" transfers. Timely [23] is designed for RDMA in datacenters. These schemes generally perform poorly when competing with loss-based elastic algorithms. Compound TCP [30] maintains both a loss-based window and a delay-based window, and transmits data based on the sum of the two windows. Compound achieves high throughput as soon as the link utilization drops because of its delay window, but does not control self-inflicted delays because of its loss window.

3 Cross-Traffic Estimation

如何对竞争流的速率进行测量

We first show how to estimate the total rate of cross traffic (§3.1) from measurements at the sender. Then, we show how to detect whether elastic flows are a significant contributor to the cross traffic, describing the key ideas (§3.2) and a practical method (§3.3).

Figure 2 shows the 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

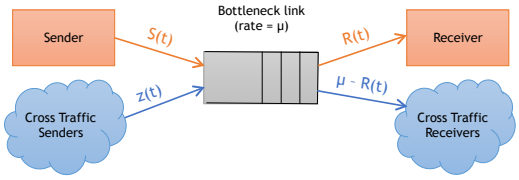


Figure 2: **Network model.** The time-varying total rate of cross traffic is $z(t)$. The bottleneck link rate is μ. The sender's transmission rate is $S(t)$, and the rate of traffic received by the receiver is $R(t)$.

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. We assume that the sender knows μ, and use prior work to estimate it (§4.3).

Our cross traffic estimation technique requires some degree of traffic persistence. The sender must be able to create sufficient traffic variations and observe the impact on cross traffic over a period of time. Hence it is best suited for control of large flows, such as large file downloads, data backups to the cloud, etc. Fortunately, it is precisely for such transfers that delay-control congestion control can provide the most benefit, since short flows are unlikely to cause bufferbloat [12].

3.1 Estimating the Rate of Cross Traffic

We estimate $z(t)$ using the estimator

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

To understand why this estimator works, see Figure 2. 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, $S(t)+z(t)$. Thus, any protocol that keeps the bottleneck link always busy can estimate $z(t)$ using Eq. (1).

In practice, we can estimate $S(t)$ and $R(t)$ by considering n packets at a time:

$$S_{i,i+n} = \frac{n}{s_{i+n} - s_i}, \qquad R_{i,i+n} = \frac{n}{r_{i+n} - r_i} \quad (2)$$

where 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 rate are packets per second. Note that measurements of $S(t)$ and $R(t)$ must be performed over the *same* n packets.

We have conducted several tests with various patterns of cross traffic to evaluate the effectiveness of this $z(t)$ estimator. The overall error is small: the 50th and 95th percentiles of the relative error are 1.3% and 7.5%, respectively.

3.2 Elasticity Detection: Principles

We seek an online estimator to determine if the cross traffic includes elastic flows using only measurements at the sender.[2]

[2]Receiver modifications might improve accuracy by avoiding the need to estimate $R(t)$ from ACKs at the sender, but would be a little harder to deploy.

问题：为什么inelastic流会对ack的变化无动于衷？

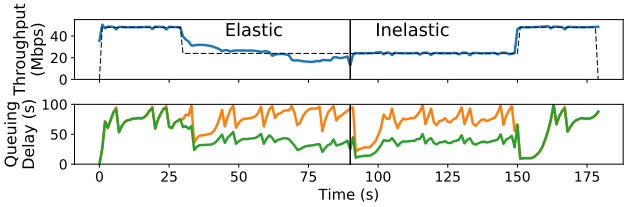


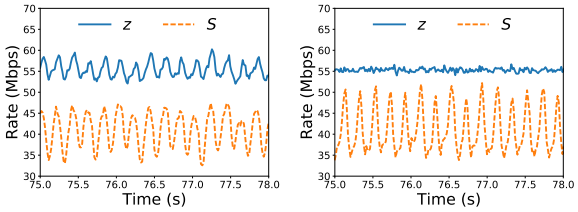
Figure 3: Delay measurements at a single point in time do not reveal elasticity. The bottom plot shows the total queuing delay (orange line) and the self-inflicted delay (green line). The experiment setup is the same as Figure 1a.

A strawman approach might attempt to detect elastic flows by estimating the contribution of the cross traffic to queuing delay. For example, the sender can estimate its own contribution to the queuing delay—i.e., the "self-inflicted" delay—and if the total queuing delay is significantly higher than the self-inflicted delay, conclude that the cross traffic is elastic.

This simple scheme does not work. To see why, consider again the experiment in Figure 1a, where a Cubic flow shares a link with both elastic and inelastic traffic in two separate time periods. Figure 3 plots the self-inflicted queuing delay for the Cubic flow in the same experiment. The self-inflicted delay looks nearly identical in the elastic and inelastic phases of the experiment. The reason is that a flow's share of the queue occupancy is proportional to its throughput, independent of the elasticity of the cross traffic. This example suggests that no measurement at a single point in time can be used to reliably distinguish between elastic and inelastic cross traffic. In this experiment, the Cubic flow gets roughly 50% of the bottleneck link; therefore, its self-inflicted delay is roughly half of the total queuing delay at all times.

To detect elasticity, tickle the cross traffic! Our method detects elasticity by monitoring how the cross traffic responds to induced traffic variation 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. Because elastic flows are ACK-clocked, if a new ACK is delayed by a time duration δ seconds, then the next packet transmission will also be delayed by δ. The sending rate depends on this delay: if the mean inter-arrival time between ACKs is d, adding an extra delay of δ on each ACK would reduce the flow's sending rate from $1/d$ to $1/(d+\delta)$ packets per second. By contrast, inelastic traffic does not respond like this to mild increases in delay.

We use this observation to detect elasticity by inducing changes in the inter-packet spacing of cross traffic at the bottleneck link. To achieve this, we transmit data in *pulses*, taking the desired sending rate, $S(t)$, and sending at a rate first higher, then lower than $S(t)$, while ensuring that the mean rate remains $S(t)$. Sending in such pulses (e.g., modulated on a sinusoid) modulates the inter-packet spacing of the cross traffic in the queue in a controlled way. If enough of the cross-traffic flows are elastic, then because of the explicitly induced changes in the ACK clocks of those flows, they will react to the changed inter-packet time. In particular, when we increase our rate and transmit a burst, the elastic cross traffic will reduce its rate in the next RTT; the opposite will happen when we decrease our rate.



(a) Elastic traffic (b) Inelastic traffic

Figure 4: **Cross traffic's reaction to sinusoidal pulses.** The pulses cause changes in the inter packet spacing for cross traffic. Elastic traffic reacts to these changes after a RTT. Inelastic cross traffic is agnostic to these changes.

Fig. 4a and Fig. 4b show the responses of elastic and inelastic cross-traffic flows (z), when the sender transmits data in sinusoisal pulses ($S(t)$) at frequency $f_p = 5$ Hz. The elastic flow's sending rate after one round-trip time of delay 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. First, pulses in our sending rate must induce a measurable change in z, but must not be so large as to congest the bottleneck link. Second, because there is natural variation in cross-traffic, as well as noise in the estimator of z, it is not easy to perform a robust comparison between the predicted change in z and the measured z. Third, 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 measured the *cross-correlation* between $S(t)$ and $z(t)$. If the cross-correlation was close to zero, then the traffic would be considered inelastic, but 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 cross traffic will react after its RTT, and thus we must align $S(t)$ and $z(t)$ 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, and rendering the method impractical.

From time to frequency domain. We have developed a method that overcomes the three challenges mentioned 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 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, and the period of the pulses is short (e.g., $f_p = 5$ Hz). By using short

是否可以考虑在第三点结合SDN和CCP

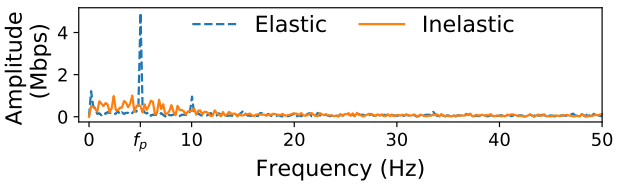


Figure 5: **Cross traffic FFT for elastic and inelastic traffic.** Only the FFT of elastic cross traffic has a pronounced peak at f_p (5 Hz).

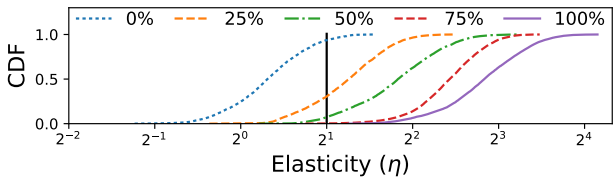


Figure 6: Distribution of elasticity with varying elastic fraction of cross traffic. Completely inelastic cross traffic has elasticity values close to zero, while completely elastic cross traffic exhibits high elasticity values. Cross traffic with some elastic fraction also exhibits high elasticity ($\eta > 2$).

pulses, we ensure that the total burst of data sent in a pulse is a small fraction of the bottleneck's 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 $z(t)$ over a short time interval (e.g., 5 seconds). Observing the cross-traffic's response at a known frequency, f_p, yields a method that is robust to different RTTs in cross traffic.

Fig. 5 shows the FFT of the $z(t)$ time-series estimate 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 elastic the cross traffic is; for example, the more elastic the cross traffic, the sharper the peak at f_p.

Because the peak magnitude depends on the proportion of elastic flows in cross traffic, we found that a more robust indicator of elasticity is to compare the magnitude of the f_p peak to the next-best peak at nearby frequencies, rather than use a pre-determined absolute magnitude threshold. We define the *elasticity metric*, η as follows:

$$\eta = \frac{|FFT_z(f_p)|}{\max_{f \in (f_p, 2f_p)} |FFT_z(f)|} \tag{3}$$

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 $2f_p$. In Fig. 5, η for elastic traffic is 10, whereas for inelastic traffic it is close to 1.

In practice, the cross traffic is less likely to be either only elastic or only inelastic, but will be a mix. Fig. 6 shows elasticity of the cross traffic when we vary the percentage of bytes belonging to elastic flows in the cross traffic. Based on this data, we propose a hard-decision rule: if $\eta \leq 2$, then the cross traffic is considered inelastic; otherwise, it is elastic.

Pulse shaping. Rather than a pure sinusoid, we use an *asymmetric sinusoidal pulse*, as shown in Fig. 7. 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 $\mu/4$,

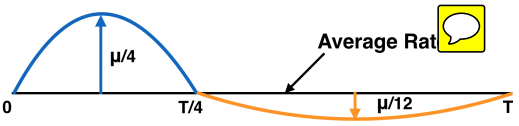


Figure 7: **Example of an asymmetric sinusoidal pulse.** The pulse has period $T = 1/f_p$. The positive half-sine lasts for $T/4$ with amplitude $\mu/4$, and the negative half-sine lasts for the remaining duration, with amplitude $\mu/12$. The two half-sines are designed to cancel out each other over one period.

a sender with $S(t)$ as low as $\mu/12$ can generate the asymmetric pulse shown in Fig. 7; a symmetric pulse with the same peak rate would require $S(t) > \mu/4$. A peak pulse rate of $\mu/4$ causes a noticeable change to inter-packet times by transmitting a fraction of BDP worth of packets over a short time period (less than an RTT).

What should the duration, T, of the pulse be? The answer depends on two factors: first, the duration over which S and R are measured (with which the sender estimates z), and second, the amount of data we are able to send in excess of the mean rate without causing excessive congestion. If T were smaller than the measurement interval of S and R, then the pulse would have no measurable effect, because the excess in the high part and low part would cancel out over the measurement interval. But T cannot be too large because the sender sends in excess of the mean rate $S(t)$ for $T/4$.

Based on these considerations, we set T so that $T/4$ is on the order of the current RTT or a little less than that to avoid packet losses (e.g., $T/4$ could be the minimum observed RTT), and measure S and R (and hence, z) using Eq. (2) over the duration of the current RTT (i.e., over exactly one window's worth of packets). As a concrete example, a flow with minimum RTT 50 ms would use $T/4 = 50$ ms, giving a pulse frequency of $1/0.2 = 5$ Hz.

Our pulses are designed to produce an observable pattern in the FFT when the cross traffic is elastic. Using asymmetric sinusoidal pulses creates extra harmonics at multiples of the pulse frequency f_p. However, these harmonics do not affect the elasticity metric in Eq. (3), which only considers the FFT in the frequency band $[f_p, 2f_p)$.

4 Nimbus Protocol

This section describes a protocol that uses mode switching. It has a TCP-competitive mode in which the sender transmits using Cubic's congestion avoidance phase, and a delay-control mode that uses a method, described below. The sender switches between the two modes using the elasticity detector described in the previous section, transmitting data at the time-varying rate dictated by the congestion control, and modulating its transmissions on the asymmetric sinusoid of Fig. 7.

Nimbus需要将带宽占满直到出现队列，这是否是本文的缺点之一？

4.1 Delay-Control Rule

Nimbus uses a delay-control control rule inspired by ideas in XCP [18], TIMELY [23], and PIE [26]. It seeks to achieve high throughput while maintaining a specified threshold queuing delay, $d_t > 0$. A positive d_t ensures that the link is rarely under-utilized, and allows us to estimate $z(t)$. The protocol seeks to deliver an ideal rate of $\mu - z(t)$. The reason we designed our own method rather than use an existing one like Vegas [3] is because our ability to estimate z yields tighter controls on delay than prior protocols.

The control rule has two terms. The first seeks to achieve the ideal rate, $\mu - z$. The second seeks to maintain a specified threshold queuing delay, d_t, to prevent the queue from both emptying and growing too large.

Denote the minimum RTT by $x_{\min}$ and the current RTT by x_t. The Nimbus delay-control rule is

$$S(t+\delta) = (1-\alpha)S(t) + \alpha(\mu - z(t)) + \beta\frac{\mu}{x_t}(x_{\min} + d_t - x_t). \quad (4)$$

Prior work (XCP [18] and RCP [2]) has established stability bounds on α and β for nearly identical control laws. Our implementation uses $\alpha = 0.8$, and $\beta = 0.5$.

4.2 Mode Switching

Nimbus uses the pulsing parameters described in §3.3, calculating S and R over one window's worth of packets and setting $T/4$ to the minimum RTT. It computes the FFT over multiple pulses and uses the z measurements reported in the last 5 seconds to calculate elasticity (η) using Eq. (3).

We found earlier (Fig. 6) that a good threshold for η is 2. To prevent frequent mode switches, Nimbus applies a hysteresis to this threshold before switching modes. When in delay-control mode, η must exceed 2.25 to switch to TCP-competitive mode, and when in TCP-competitive mode, η must be lower than 2 to switch to delay-control mode.

It is important that the rate be initialized carefully on a mode switch. When switching to TCP-competitive mode, Nimbus sets the rate (and equivalent window) after switching to the rate that was being used five seconds ago (five seconds is the duration over which we calculate the FFT). The reason is that the elasticity detector takes five seconds to detect the presence of elastic cross traffic, and the arrival of elastic traffic over the past five seconds would have reduced the delay mode's rate. We set the new congestion window to the inflection point of the Cubic function, so the rate over the next few RTTs will rise faster in each successive RTT, and we reset `ssthresh` to this congestion window to avoid entering slow start.

While switching to delay mode, Nimbus resets the delay threshold to the current value of $x_t - x_{\min}$, rather than to the desired threshold d_t, and linearly decreases the threshold used in the control rule to the desired d_t over several RTTs. The reason is that $x_t - x_{\min}$ is likely to have grown much larger than d_t in TCP-competitive mode, and using d_t would only cause the sender to reduce its rate and go down in throughput. The approach we use ensures instead that the delay mode will drain the queues gradually and won't lose throughput instantly, and also provides a safeguard against incorrect switches to delay mode.

4.3 Implementation

We implemented Nimbus using the congestion control plane (CCP) [24], which provides a convenient way to express the signal processing operations in user-space code while achieving high rates. The implementation runs at up to 40 Gbit/s using the Linux TCP datapath. All the Nimbus software is in user-space. The Linux TCP datapath uses a CCP library to report to our Nimbus implementation the estimates of S, R, the RTT, and packet losses every 10 ms. S and R are measured using the methods built in TCP BBR over one window's worth of packets. Our implementation of Nimbus is rate-based, and sets a cap on the congestion window to prevent uncontrolled "open-loop" behavior when ACKs stop arriving.

On every measurement report, Nimbus (1) updates the congestion control variables of the current mode and calculates a pre-pulsing sending rate, (2) superimposes the asymmetric pulse in Fig. 7 to obtain the actual sending rate, and (3) generates the FFT of z and makes a decision to switch modes using a 5-second measurement period for the FFT.

We note that calculating z requires an estimate of the bottleneck link rate (μ). There has been much prior work [16, 9, 10, 19, 17, 20, 21] in estimating μ, any of which could be incorporated in Nimbus. Like BBR, our current implementation uses the maximum received rate, taking care to avoid incorrect estimates due to ACK compression and dilation.

4.4 Visualizing Nimbus and Other Schemes

We illustrate Nimbus on a synthetic workload with time-varying cross traffic. We emulate a bottleneck link in Mahimahi [25], a link emulator. The network has a bottleneck link rate of 96 Mbit/s, a minimum RTT of 50 ms, and 100 ms (2 BDP) of router buffering. Nimbus sets a target queuing delay threshold of 12.5 ms in delay-control mode. We compare Nimbus with Linux implementations of Cubic, BBR, and Vegas, our empirically-validated implementation of Compound atop CCP, and implementations of Copa and PCC provided by the respective 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 a specified mean rate. Elastic cross-traffic uses Cubic. We generate all elastic traffic (Nimbus and cross-traffic) using `iperf` [31].

Fig. 8 shows the throughput and queuing delays for the various algorithms, as well as the correct fair-share rate over time. Throughout the experiment, Nimbus achieves both its fair share rate and low ($\leq$20 ms) queuing delays in the presence of inelastic cross-traffic. With elastic cross-traffic, Nimbus switches to competitive mode within 5 s and achieves close to its fair-share rate. The delays during this period approach the buffer size because the competing traffic is buffer-filling; the delays return to their previous low value (20 ms) within 5 s after the elastic cross-flows complete. Nimbus stays in the correct mode throughout the experiment, except for one interval in the competitive period.

Cubic experiences high delays close to the buffer size (100 ms) throughout the experiment.

BBR与其他流竞争的缺点

BBR's throughput that is often *significantly higher* than its fair share and suffers from high delays even with inelastic cross-traffic; this is consistent with a prior result [1]. Setting its sending rate to the estimated link rate is problematic in the presence of cross-traffic. Furthermore, BBR's use of the maximum achieved delivery rate as the "right" sending rate has been shown [14] to cause BBR to unnecessarily inflate queuing delays.

Vegas suffers from low throughput in the presence of elastic cross-traffic, as it reduces its sending rate in the presence of large delays. Compound ramps up its rate quickly whenever it detects low delays, but behaves like TCP Reno otherwise. Consequently, it attains lower than its fair-share rate in the presence of Cubic flows, and suffers from high delays even with inelastic cross-traffic.

Copa mostly uses the correct mode but it has frequent incorrect mode switches. This increases variability and causes Copa to lose throughput (55 Mbit/s versus 68 Mbit/s for Nimbus) and occasionally suffer high delay fluctuations in the inelastic periods. Against Cubic flows, Copa achieves lower throughput than its fair share mainly because it emulates Reno.

PCC optimizes delivery rates using an online objective function, but this local optimization results in significant unfairness to TCP cross-traffic as well as high queuing delays.

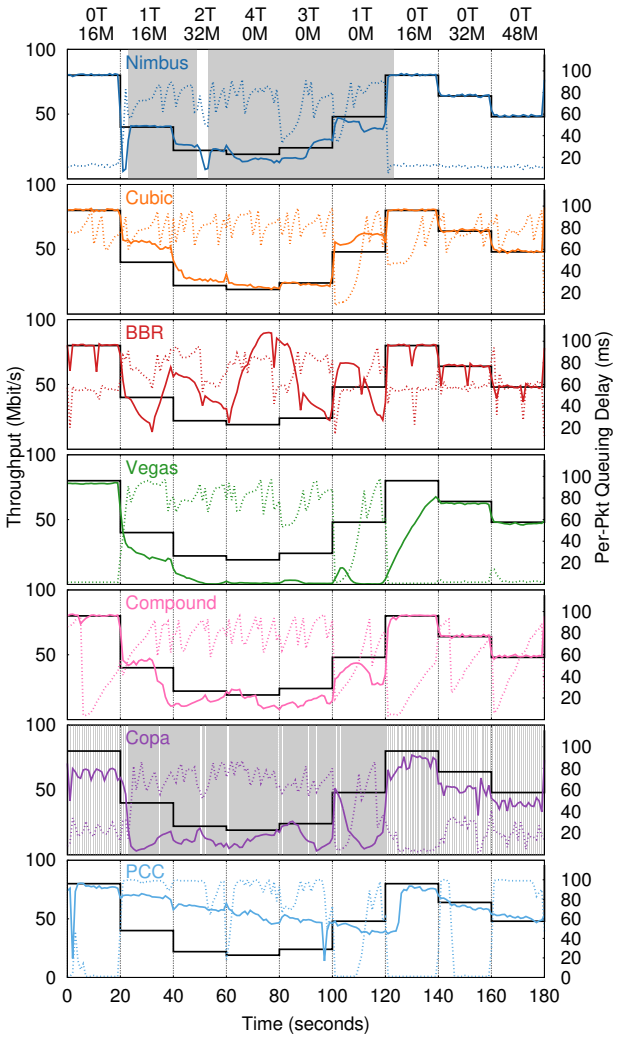


Figure 8: Performance on a 96 Mbit/s Mahimahi link with 50 ms delay and 2 BDP of buffering while varying the rate and type of cross traffic as denoted at the top of the graph. xM denotes x Mbit/s of inelastic Poisson cross-traffic. yT denotes y long-running Cubic cross-flows. The solid black line indicates 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 dash line shows queuing delay. For Nimbus and Copa, the shaded regions indicate TCP-competitive periods.

5 Multiple Nimbus Flows

What happens when a bottleneck is shared by multiple Nimbus flows running on different senders? The goal is for all the Nimbus flows to remain in delay-control mode when there is no other elastic cross traffic, and compete well with elastic cross traffic otherwise. The problem is that pulsing may create adverse interactions that confuse the different Nimbus instances.

One approach is for the Nimbus flows to all pulse at the same frequency. However, in this case, they will all detect a peak in the FFT at the oscillation frequency. They will all then stay in TCP-competitive mode and won't be able to maintain low delays, even when there is no elastic cross traffic. A second approach is for different Nimbus flows to pulse at different frequencies. The problem with this approach is that it cannot scale to more than a few flows, because the set of possible frequencies is too small (recall that we require $T/4 \approx$RTT).

Watchers, meet Pulser. We propose a third approach. One of the Nimbus flows assumes the role of the *pulser*, while the others are *watchers*. The coordination between them involves no explicit communication; in fact, each Nimbus flow is unaware of the identities, or even existence, of the others.

The pulser sends data by modulating its transmissions on the asymmetric sinusoid. The pulser uses two different frequencies,

f_{pc} in TCP-competitive mode, and f_{pd} in delay-control mode; e.g., 5 Hz and 6 Hz. The values of these frequencies are fixed and agreed upon beforehand. A watcher infers whether the pulser is pulsing at frequency f_{pc} or frequency f_{pd} by observing the FFT at those two frequencies and uses it to set its own mode to match the pulser's mode. The variation in the queuing due to pulsing will cause a watcher flow's receive rate to pulse at the same frequency. The watcher computes the FFT at the two possible pulsing frequencies and picks the mode corresponding to the larger value. With this method, a watcher flow can determine the mode without estimating the bottleneck link rate or controlling a significant portion of the link rate.

For multiple Nimbus flows to maintain low delays during times when there is no elastic cross traffic on the link, the pulser flow must classify watcher traffic as inelastic. Note that from the pulser's perspective, the watchers 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 a low pass filter to its transmission rate before sending data. The low pass filter cuts off all frequencies in the sending rate that exceed $\min(f_{pc}, f_{pd})$.

Pulser election. A decentralized and randomized election process decides which flow is the pulser and which are watchers. If a Nimbus 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:

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

Each flow makes decisions periodically, e.g., every $\tau = 10$ ms, and κ is a constant. Since the FFT duration is 5 seconds, each p_i is small (note that $\sum_i R_i \leq \mu$), but since flows make decisions every τ seconds, eventually one will become a pulser.

If the estimates $\hat{\mu}_i$ are equal to the true bottleneck rate μ, then 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 the number of pulsers within an FFT duration has approximately a Poisson distribution with a mean of κ [11]. 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 controls the tradeoff between fewer conflicts vs. longer time to elect a pulser.

For any value of κ, there is a non-zero probability of more than one concurrent pulser. 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, Nimbus concludes that there must be multiple pulsers and switches to a watcher with a fixed probability.

Remark. The multiple-Nimbus scheme for coordinating pulsers bears resemblance to receiver-driven layered multicast (RLM) congestion control [22]. 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. By contrast, in Nimbus, there is no explicit coordination channel, and the pulsers and watchers coordinate via their independent observations of cross traffic patterns.

6 Evaluation

We answer the following questions in this section.

§6.1: Does Nimbus achieve low delay and high throughput?
§6.2: Is Nimbus's mode switching more accurate than Copa?
§6.3: Does Nimbus need to control a large link share?
§6.4: Does the elasticity detector track the elastic fraction?
§6.5: How robust is Nimbus's elasticity detection?
§6.6: Can multiple Nimbus flows co-exist well?
§6.7: Does Nimbus perform well on real Internet paths?
§6.8: Can we use other delay-based and buffer-filling algorithms?

We evaluate the elasticity detection method and Nimbus using the Mahimahi emulator with realistic workloads, and on Internet paths. Our Internet experiments are over paths between Amazon's EC2 machines around the world, well-connected university networks, and residential hosts.

6.1 Does Nimbus Achieve Low Delays and High Throughput?

We evaluate the delay and throughput benefits of mode switching using trace-driven emulation. We generate cross-traffic from an empirical distribution of flow sizes derived from a wide-area packet trace from CAIDA [4]. This packet trace was collected at an Internet backbone router on January 21, 2016 and contains over 30 million packets recorded over 60 seconds. The maximum rate over any 100 ms period is 2.2 Gbit/s. We generate Cubic cross-flows with flow sizes drawn from this data, with flow arrival times generated by a Poisson process to offer a fixed average load.

One backlogged flow running a fixed algorithm (Nimbus, Cubic, Vegas, Copa, or BBR) and the WAN-like cross-flows share a 96 Mbit/s Mahimahi bottleneck link with a propagation RTT of 50 ms and a bottleneck buffer of 100 ms. We generate cross-traffic to fill 50% of the link (48 Mbit/s) on average. Nimbus uses a queuing delay threshold of 12.5 ms in delay-mode and emulates Cubic in competitive-mode.

Fig. 9 shows throughput (over 1-second intervals) and packet delays of Nimbus, Vegas, Cubic, Copa, and BBR. Nimbus achieves a throughput distribution comparable to Cubic and BBR. Unlike Cubic and BBR, however, Nimbus also achieves low RTTs, with a median only 10 ms higher than Vegas and >50 ms lower than Cubic and BBR. Vegas suffers from low throughput.

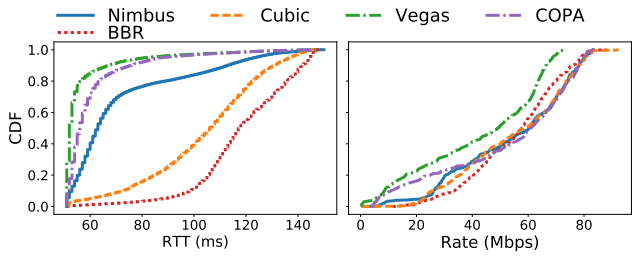


Figure 9: Nimbus reduces delay relative to Cubic and BBR while achieving comparable throughput on a cross-traffic workload derived from a packet trace collected at a WAN router. Vegas and Copa also reduce delay but lose throughput.

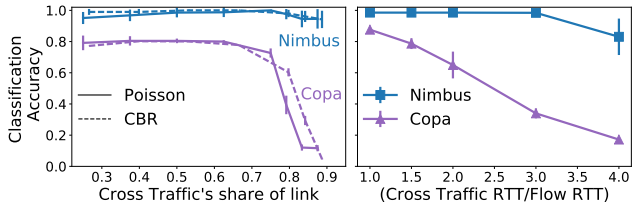


Figure 10: Nimbus achieves higher classification accuracy than Copa when (i) inelastic cross traffic occupies a large fraction of the link (left), and (ii) elastic cross traffic with higher RTT then the flow's RTT (right).

Because of mode switching, Nimbus and Copa both achieve low delays while maintaining an overall high throughput. However, Copa has lower throughput than Nimbus about 20% of the time (lowest 20th percentile in the rate CDF). The reason is that Copa makes some incorrect mode switches in periods where the cross traffic includes large elastic flows.

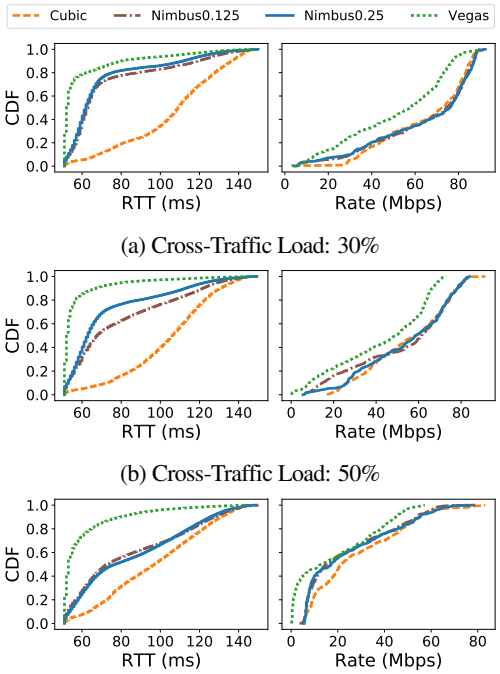
We also measured the flow completion time (FCT) of cross-flows and found that Nimbus benefits cross traffic: it reduces 95th percentile completion times for cross traffic flows by 3-4× compared to BBR, and 1.25× compared to Cubic for short ($\leq$ 15 KB) flows. Appendix A provides details.

6.2 Is Nimbus More Accurate than Copa?

The experiment in §4.4 showed that Nimbus makes fewer mistakes in selecting its mode compared to Copa. We compare the classification accuracy of Nimbus to Copa in two scenarios. First, we generate inelastic cross traffic of different rates and measure the fraction of the time a backlogged Nimbus and Copa flow operate in the correct mode. This experiment uses a 96 Mbit/s bottleneck link with a 50 ms propagation RTT and a 100 ms drop-tail buffer (2 BDP). We consider both constant-bit-rate (CBR) and Poisson cross traffic, generated as described in §4.4.

Fig. 10 (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.

In the second scenario, a backlogged Nimbus or Copa flow competes against a backlogged NewReno flow, and we vary the RTT of the NewReno flow between $1-4\times$ the RTT of the Nimbus/Copa flow. Fig. 10 (right) shows that Copa's accuracy



(a) Cross-Traffic Load: 30%

(b) Cross-Traffic Load: 50%

(c) Cross-Traffic Load: 90%

Figure 11: At low cross-traffic loads, Nimbus's queuing delay approaches that of Vegas while its throughput approaches that of Cubic. At high loads, Nimbus behaves like Cubic. Increasing pulse size improves switching accuracy and performance.

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 Nimbus.

These two results highlight the probems with Copa's approach of setting the operating mode using an expected pattern of queuing delays. We investigated the reasons for Copa's errors in these experiments. In the first case with inelastic cross traffic, Copa is unable to drain the queue quickly enough (every 5 RTTs); this throws off Copa's detection rule and it switches to competitive mode. In the second case, because a NewReno cross traffic flow with a high RTT increases its rate slowly, it has a small impact on the queuing delay during Copa's 5 RTT period. Therefore, Copa can drain the queue as it expects and it concludes that there isn't any non-Copa cross traffic. This error continues until the rate of the cross traffic gets large enough to interfere with Copa's expected queuing delay behavior.

By contrast, Nimbus directly estimates the volume and elasticity of cross traffic and is more robust. Appendix D provides examples of the throughput and queuing delay dynamics of Copa and Nimbus in the two scenarios.

6.3 Does Nimbus Need to Control a Large Link Share?

Must Nimbus control a significant fraction of the link rate to reap the benefits of mode switching? We evaluate Nimbus's delay and

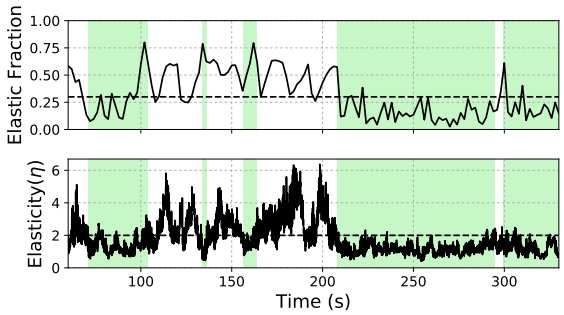


Figure 12: The elasticity metric, and hence Nimbus's mode, closely tracks the prevalence of elastic cross-traffic (ground truth measured independently using the volume of ACK-clocked flows). Green-shaded regions indicate inelastic periods.

throughput relative to other algorithms when Nimbus controls varying shares of the link rate. We generate cross-traffic as described in §6.1 but vary the offered load of the cross-traffic at three levels (30%, 50%, and 90% of the link rate). We measure throughput and delay for two pulse sizes: 0.125μ and 0.25μ.

Fig. 11 shows our findings. First, in all cases, Nimbus lowers delay without hurting throughput, with the delay benefits most pronounced when cross traffic is low. Second, as cross traffic increases, Nimbus's delay improvements decrease, because it must stay in TCP-competitive mode more often. Third, Nimbus's behavior is better at the larger pulse size, but its benefits are generally robust even at 0.125μ.

Elasticity detection is less accurate with smaller pulse amplitude (0.125μ), causing more errors in mode switching. With this pulse size, Nimbus does not lower its delays or maintain its throughput as effectively at medium load (50%) when switching correctly matters more for good performance.

In the next two sections, we evaluate the applicability of Nimbus's elasticity detection method to real workloads, and test the method's robustness to various network-level characteristics.

6.4 Does η Track the True Elastic Fraction?

We use the setup of §6.1 to present a mix of elastic and inelastic cross traffic, and evaluate how well the detector's decision correlates with the true proportion of elastic traffic. We define a cross-flow generated from the CAIDA trace as "elastic" if it is guaranteed to have ACK-clocked packet transmissions over its lifetime, i.e., flows with sizes higher than TCP's default initial congestion window (10 packets in Linux 4.10, which is our transmitter).

The top chart in Fig. 12 shows the fraction of bytes belonging to elastic flows as a function of time. The bottom chart shows the output of the elasticity detector along with the dashed threshold line at $\eta = 2$. The green shaded regions are times when Nimbus was in delay-control mode. These correlate well with the times when the elastic fraction was low; when the elasticity fraction is < 0.3, the elasticity detector correctly outputs "inelastic" over 90% of the time, when one accounts for the 5 second estimation delay. Even counting that delay, the accuracy is over 80%.

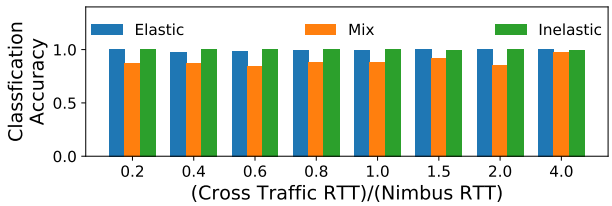


Figure 13: Nimbus classifies purely elastic and inelastic traffic with accuracy greater than 98%. For a mix of elastic and inelastic traffic, the average accuracy is greater than 80% in all cases.

6.5 How Robust is Elasticity Detection?

We now evaluate the robustness of our elasticity detection method to cross-traffic and Nimbus's RTTs, bottleneck link rate, the share of the bottleneck controlled by Nimbus, bottleneck buffer size, and Nimbus's pulse size. Unless specified otherwise, we run Nimbus as a backlogged flow on a 96 Mbit/s bottleneck link with a 50 ms propagation delay and a 100 ms drop-tail buffer (2 BDP). We supply Nimbus with the correct link rate for these experiments, allowing us to study the robustness of elasticity detection with respect to the properties mentioned above. We consider three categories of synthetic cross-traffic sharing the link with Nimbus: (i) fully inelastic traffic (Poisson); (ii) fully elastic traffic (backlogged NewReno flows); and (iii) an equal mix of inelastic and elastic. The duration of each experiment is 120 seconds. The main performance metric is *accuracy*: the fraction of time that Nimbus correctly detects the presence of elastic cross-traffic.

Impact of cross-traffic RTT. We vary the cross-traffic's propagation round-trip time from 10 ms (0.2× that of Nimbus) to 200 ms (4× that of Nimbus). Fig. 13 shows the mean accuracy across 5 runs of each category of cross-traffic. We find that varying cross-traffic RTT does not impact detection accuracy. For purely inelastic and purely elastic traffic, Nimbus achieves an average accuracy of more than 98% in all cases, while for mixed traffic, Nimbus achieves a mean accuracy of 85% in all cases (a random guess would've achieved only 50%).

Impact of pulse size, link rate, and link share controlled by Nimbus. We perform a multi-factor experiment varying Nimbus's pulse size from 0.0625×—0.5× the link rate, Nimbus's fair share of the bottleneck link rate from 12.5%—75%, and bottleneck link rates set to 96, 192, and 384 Mbit/s. The accuracy for purely elastic cross-traffic was always higher than 95%.

Fig. 14 shows the average detection accuracy over five runs of the other two categories of cross-traffic. Nimbus achieves an accuracy of more than 90% averaged over all the points. In general, increasing the pulse sizes improves accuracy because Nimbus 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. For the same reason, decreasing Nimbus's share of the link also results in higher

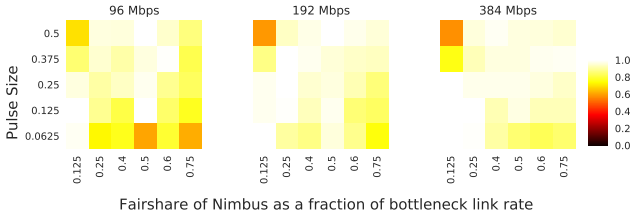


Figure 14: Nimbus is robust to variations in link bandwidth and fraction of traffic controlled by it. Increasing pulse size increases robustness.

accuracy in general. However, at low link rates, Nimbus 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.

Impact of buffer size and RTT. We vary the buffer size from 0.25 BDP to 4 BDP for each category of cross traffic, at propagation round-trip times of 25 ms, 50 ms, and 75 ms. With purely elastic or inelastic traffic, Nimbus has an average accuracy (across 5 runs) of 98% or more in all cases but one (see below), while with mixed traffic, the accuracy is always 80% or more. With shallow buffers, when the buffer size is less than the product of the delay threshold x_t and the bottleneck link rate (i.e., 0.25 BDP when the round-trip time is 50 ms), Nimbus classifies all traffic as elastic. However, this low accuracy does not impact the performance of Nimbus, as Nimbus achieves its fair-share throughput and low delays (bounded by the small buffer size). Further, 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 are hard to observe.

6.6 Can Multiple Nimbus Flows Co-Exist?

Does mode switching permit low delays in the presence of multiple mode switching flows (and possibly other cross-traffic)? Can multiple such flows share a bottleneck link fairly with each other and with cross-traffic? We run Nimbus with Vegas as its delay-mode algorithm and supply it with the correct link rate. (We use Vegas because Nimbus's rule in its present form is not fair to other flows with the same rule.)

Fig. 15 demonstrates how Nimbus flows react as other Nimbus 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 ($t = 120,240,360$ 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 ($t = 480,720$ s). The pulser can be identified visually by its rate variations.

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

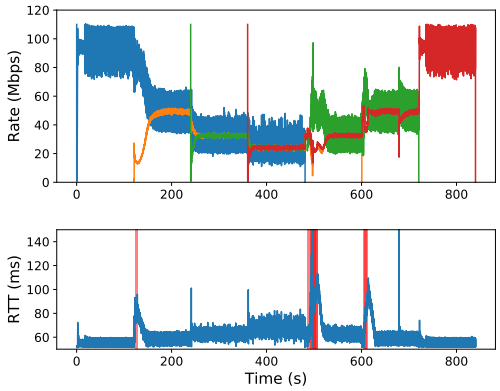


Figure 15: **Multiple competing Nimbus flows.** Multiple Nimbus flows achieve fair sharing of a bottleneck link (top graph). There is at most one pulser flow at any time, which can be identified by its rate variations. Together, the flows achieve low delays by staying in delay mode for most of the duration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

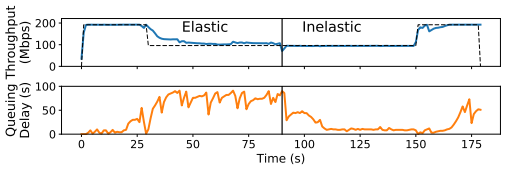


Figure 16: **Multiple Nimbus flows and other cross-traffic.** There are 3 Nimbus flows throughout. Cross traffic in duration 30-90 s is elastic and made up of three Cubic flows. Cross traffic in duration 90-150 s is inelastic and made up of a 96 Mbit/s constant bit-rate stream. Multiple Nimbus flows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

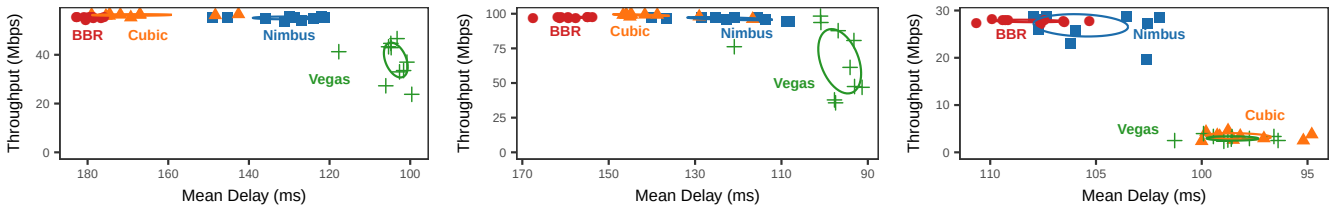
delay-mode for most of the time.

Fig. 16 demonstrates multiple Nimbus flows switching in the presence of cross-traffic. We run three Nimbus flows on an emulated 192 Mbit/s link with a propagation delay of 50 ms. The cross traffic is synthetic. 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 Nimbus flows, along with a reference line for the fair-share rate of the aggregate. The graph at the bottom shows the measured queuing delays. Nimbus shares the link fairly with other cross-traffic, and achieves low delays by staying in the delay mode in the absence of elastic cross-traffic.

6.7 Real-World Internet Paths

We ran Nimbus on the public Internet with a test-bed of five servers and five clients, 25 paths in all. The servers are Amazon EC2 instances located in California, London, Frankfurt, Ireland, and Paris, and are rated for 10 Gbit/s. We verified that the bottleneck in each case was not the server's Internet link. We use five residential hosts in different ASes as clients.

To understand the nature of cross traffic on these paths, we ran experiments with Nimbus's delay-control algorithm (without



(a) EC2 California to Host A (b) EC2 Ireland to Host B (c) EC2 Frankfurt 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)), Nimbus achieves the same throughput as BBR and Cubic but reduces delays significantly. On paths with significant packet drops (c), Cubic suffers but Nimbus achieves high throughput.

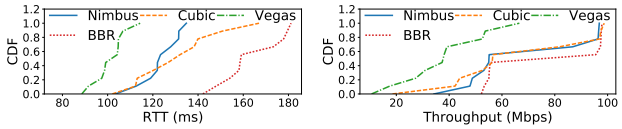


Figure 18: **Paths with queuing.** Nimbus reduces the RTT compared to Cubic and BBR (40-50 ms lower), at similar throughput.

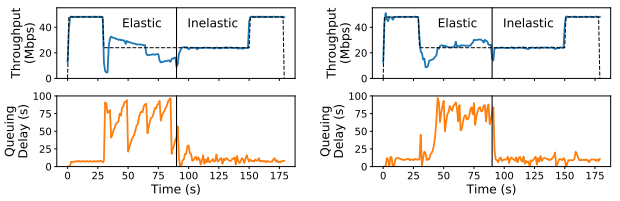
mode switching) and Cubic each performing bulk transfers over a three-day period. The results showed that scenarios where cross traffic is predominantly inelastic and thus delay-control algorithms can be effective are common; see Appendix B for details.

Next, on each path, we initiated bulk data transfers using Nimbus, Cubic, BBR, and Vegas. We ran one minute experiments over five hours on each path, and measured the achieved average throughput and mean delay. Fig. 17 shows throughput and delays over three of the paths. The x (delay) axis is inverted; better performance is up and to the right. We find that Nimbus achieves high throughput comparable to BBR in all cases, at noticeably lower 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). Vegas attains poor throughput on these paths due to its inability to compete with elastic cross-traffic. These trends illustrate the utility of mode switching 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 the paths with queuing. Nimbus obtained similar throughput to Cubic and 10% lower than BBR but at significantly lower delay (40-50 ms lower than BBR) on these paths.

6.8 How General Is Mode Switching?

We hypothesize that mode switching is a useful building block: a mode switching algorithm can use a variety of congestion control algorithms for its delay-based and TCP-competitive modes, switching using our elasticity detection method. We have implemented Cubic, Reno, and MulTCP [6] as competitive-mode algorithms, and Nimbus delay (§4.1), Vegas, FAST [33], and COPA [1] as delay-mode algorithms. In Fig. 19, we illustrate two combinations of delay and competitive mode algorithms



(a) Reno + Nimbus delay (b) Cubic + COPA

Figure 19: **Nimbus's versatility.** Mode switching with different combinations of delay-based and TCP-competitive 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 Reno+Nimbus-delay-mode (Fig. 19a) and Cubic+COPA (Fig. 19b) achieve their fair-share rate while keeping the delays low in the absence of elastic cross-traffic.

7 Limitations

The detector does not reliably characterize BBR as elastic because BBR responds on time scales longer than an RTT; here, Nimbus uses delay-based control more often than desired with BBR cross traffic. However, we find that Nimbus achieves similar throughput to Cubic in practice: with more than 1 BDP of router buffering, BBR becomes window-limited and is detected as elastic, but with shallow buffers, BBR obtains a disproportionate share of the link compared to both Cubic and Nimbus (Appendix §C).

The detector correctly characterizes delay-based schemes like Vegas as elastic, but in response runs a Cubic-like method, causing delays to grow when high throughput could have been achieved with lower delays. Determining if elastic cross traffic is also delay-controlled is an open question.

The detector assumes that the flow has a single bottleneck. Multiple bottlenecks can add noise to Nimbus's rate measurements, preventing accurate cross-traffic estimation. The challenge is that the spacing of packets at one bottleneck is not preserved when traversing the second bottleneck.

8 Conclusion

This paper showed a method for detecting the elasticity of cross traffic and showed that it is a useful building block for congestion control. The detection technique uses a carefully constructed asymmetric sinusoidal pulse and observes the frequency response of cross traffic rates at a sender. We presented several controlled experiments to demonstrate its robustness and accuracy. Elasticity detection enables protocols to combine the best aspects of delay-control methods with TCP-competitiveness. We found that our proposed methods are beneficial not only on a variety of emulated conditions that model realistic workloads, but also on a collection of 25 real-world Internet paths.

References

[1] V. Arun and H. Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *NSDI*, 2018.

[2] H. Balakrishnan, N. Dukkipati, N. McKeown, and C. J. Tomlin. Stability analysis of explicit congestion control protocols. *IEEE Communications Letters*, 11(10), 2007.

[3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

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

[5] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):50:20–50:53, Oct. 2016.

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

[7] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[8] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. PCC vivace: Online-learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA, 2018. USENIX Association.

[9] C. Dovrolis, P. Ramanathan, and D. Moore. What do packet dispersion techniques measure? In *INFOCOM*. IEEE, 2001.

[10] A. B. Downey. Using pathchar to estimate internet link characteristics. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 241–250. ACM, 1999.

[11] W. Feller. *An introduction to probability theory and its applications*, volume 2. John Wiley & Sons, 2008.

[12] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40, 2011.

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

[14] M. Hock, R. Bless, and M. Zitterbart. Experimental Evaluation of BBR Congestion Control. In *ICNP*, 2017.

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

[16] N. Hu and P. Steenkiste. Estimating available bandwidth using packet pair probing. Technical report, DTIC Document, 2002.

[17] V. Jacobson. Pathchar: A tool to infer characteristics of internet paths, 1997.

[18] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[19] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *ACM SIGCOMM Computer Communication Review*, volume 30, pages 283–294. ACM, 2000.

[20] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USITS*, volume 1, pages 11–11, 2001.

[21] B. Mar. pchar: A tool for measuring internet path characteristics. *http://www. employees. org/˜ bmah/Software/pchar/*, 2000.

[22] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven Layered Multicast. In *SIGCOMM*, 1996.

[23] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[24] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. Restructuring Endpoint Congestion Control. In *SIGCOMM*, 2018.

[25] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX Annual Technical Conference*, 2015.

[26] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *Intl. Conf. on High Performance Switching and Routing (HPSR)*, 2013.

[27] D. Rossi, C. Testa, S. Valenti, and L. Muscariello. Ledbat: The new bittorrent congestion control protocol. In *ICCCN*, pages 1–6, 2010.

[28] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT), 2012. RFC 6817, IETF.

[29] M. Sridharan, K. Tan, D. Bansal, and D. Thaler. Compound TCP: A New TCP congestion control for high-speed and long distance networks. Technical report, Internet-draft draft-sridharan-tcpm-ctcp-02, 2008.

[30] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[31] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects, 2005.

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

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

[34] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

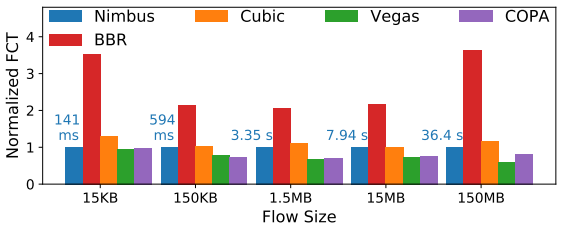


Figure 20: Using Nimbus reduces the p95 FCT of cross-flows relative to BBR at all flow sizes, and relative to Cubic for short flows. Vegas provides low cross-flow FCT, but its own rate is low.

A Does Mode Switching Help Cross-Traffic?

Using the same setup as §6.1, we measure the flow completion time (FCT) of cross-flows. Fig. 20 compares the 95th percentile (p95) cross-flow FCT for cross-flows of different sizes. The FCTs are normalized by the corresponding value for Nimbus at each flow size.

BBR exhibits much higher cross-flow FCT at all sizes compared to all the other protocols, consistent with the observation of unfairness (§4.4).

For small cross-flows ($\leq$ 15 KB), the p95 FCT with Nimbus and Copa are comparable to Vegas and lower than Cubic. With Nimbus 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-flow FCTs, but its own flow rate is dismal; Copa is more aggressive than Vegas but less than Nimbus, but at the expense of its own throughput (§6.1).

B Is Cross Traffic Ever Inelastic?

Our experiments on more than 25 Internet paths show that scenarios where cross traffic is predominantly inelastic are common. Figure 21 shows the average throughput and delay for 100 runs of a loss-based scheme (Cubic) compared to a delay-based scheme (Nimbus delay, described in §4) on one of these paths. The delay-based scheme generally achieves much lower delays than Cubic, with similar throughput. This shows that there is an opportunity to significantly improve delays using delay-based algorithms, provided we can detect loss-based TCPs and compete with them fairly when needed.

C How Well Does Nimbus Compete with BBR?

We now evaluate how well a Nimbus 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. 12 shows the average throughput of Nimbus and Cubic flows while competing with BBR over a 2-minute experiment. Nimbus achieves the same throughput as Cubic for all buffer sizes.

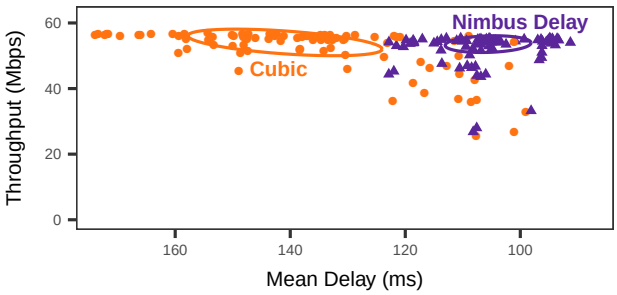


Figure 21: **Loss-based vs. delay-based congestion control.** The plot shows the average throughput and delay for 100 experiments with Cubic and the Nimbus delay-control algorithm (§4). The experiments were run between a residential client (location redacted for anonymity) and an EC2 server in California. Each experiment lasted one minute.

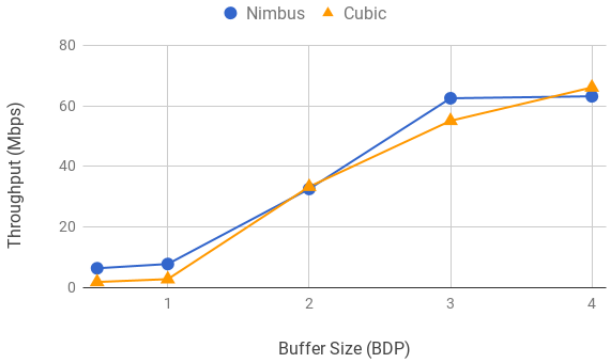


Figure 22: **Nimbus's performance against BBR is similar to that of Cubic.** Both Nimbus and Cubic compete against 1 BBR flow on a 96 Mbit/s link. For various buffer sizes, Nimbus achieves the same throughput as Cubic.

When the buffer size is $\leq$ 1 BDP, BBR is not ACK-clocked, and Nimbus classifies it as inelastic traffic. As a result, Nimbus gets a relatively small fraction of the link bandwidth. In this scenario, Cubic also gets a small fraction of the link, because BBR sends traffic at its estimate of bottleneck link and is too aggressive.

When the buffer size is $\geq$ 1 BDP, BBR becomes ACK-clocked because of the cap on its congestion window. Nimbus now classifies BBR as elastic traffic. Nimbus stays in competitive mode, behaving like Cubic and achieving the same throughput as Cubic.

D Understanding Scenarios where Copa's Mode Switching Makes Errors

We explore the dynamics of Nimbus and Copa in a few experiments from the scenarios described in §6.2. Recall that the link capacity is 96 Mbit/s, the propagation RTT is 50 ms, and the buffer size is 2 BDPs.

D.1 Constant Bit Rate Cross Traffic

Fig. 23 shows throughput and delay profile for Copa and Nimbus while competing against inelastic Constant Bit Rate (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 b), both Copa and Nimbus correctly 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 as a result 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. For 80 Mbit/s of cross traffic, this implies:

$$\begin{aligned} \max(-\frac{dQ}{dt}) &= \mu - z \\ &= 0.17 \times \mu \\ &= 0.17 \times \frac{BDP}{RTT}. \end{aligned} \tag{6}$$

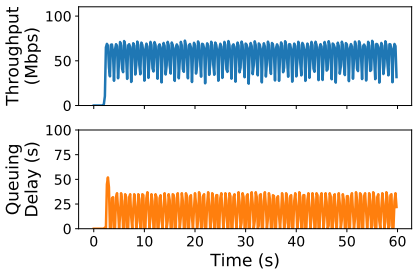
Therefore, if the queue size ever exceeds $5 \times 0.17 BDP$, then Copa won't be able to drain the queue in 5 RTTs, and it will misclassify the cross traffic as buffer-filling. The queue size can grow large due to a transient burst or if Copa incorrectly switches to competitive mode. Once Copa is in competitive mode, it will drive the queues even higher and can therefore get stuck in competitive mode.

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

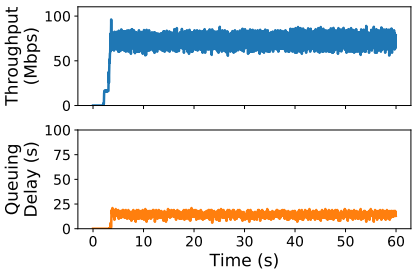
D.2 Elastic cross traffic

Fig. 24 shows throughput and delay over time for Copa and Nimbus 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 $4\times$ higher than the Copa or Nimbus flow. When the RTTs are the same (Fig. 24 a and b), both Copa and Nimbus correctly classify the cross traffic, achieving a fair share of throughput.

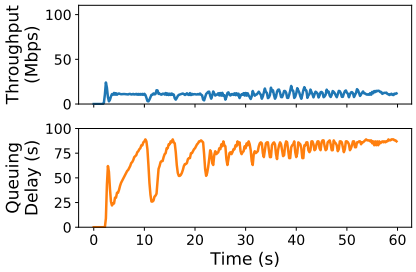
When the cross traffic RTT is higher (Fig. 24 c), the NewReno flow ramps up its rate slowly, causing Copa to misclassify the traffic and achieve less than its fair share of the throughput. Here, Copa achieves 27 Mbit/s but its fair share is at least 48 Mbit/s (and 77 Mbit/s if one considers the RTT bias).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



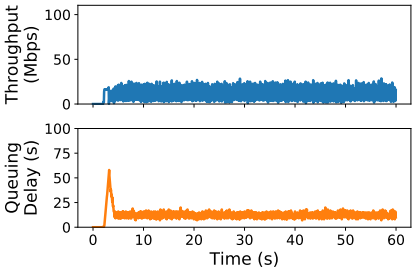
(a) Copa: 24 Mbit/s CBR



(b) Nimbus: 24 Mbit/s CBR

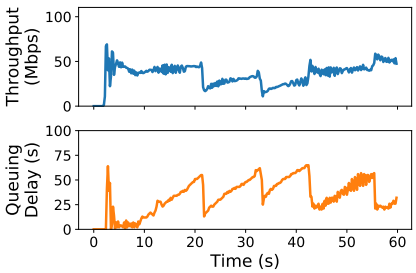


(c) Copa: 80 Mbit/s CBR

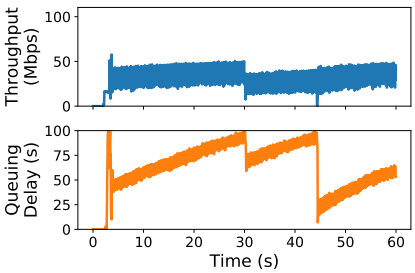


(d) Nimbus: 80 Mbit/s CBR

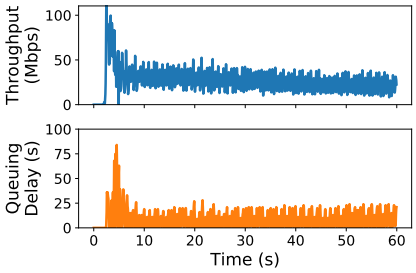
Figure 23: **Queuing delay and throughput dynamics for inelastic CBR cross traffic.** When the CBR traffic is low (a), Copa classifies the traffic as non buffer-filling and is able to achieve low queuing delays. But when the CBR traffic occupies a high fraction (c), Copa incorrectly classifies the traffic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



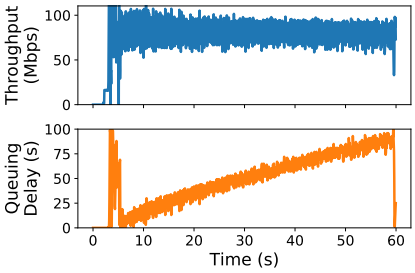
(a) Copa: Cross Traffic RTT = 1 × Flow RTT



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



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



(d) Nimbus: 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 throughput than its fair share. In both the situations (b and d), Nimbus correctly classifies the traffic as elastic and is able to achieve its fair share of throughput.