Site-to-Site Internet Traffic Control

Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, Hari Balakrishnan
MIT CSAIL

ABSTRACT

Queues allow network operators to control traffic: where queues build, they can enforce scheduling and shaping policies. In the Internet today, however, there is a mismatch between where queues build and where control is most effectively enforced; queues build at bottleneck links that are often not under the control of the data sender. To resolve this mismatch, we propose a new kind of middlebox, called Bundler. Bundler uses a novel inner control loop between a sendbox (in the sender’s site) and a receivebox (in the receiver’s site) to determine the aggregate rate for the bundle, leaving the end-to-end connections and their control loops intact. Enforcing this sending rate ensures that bottleneck queues that would have built up from the bundle’s packets now shift from the bottleneck to the sendbox. The sendbox then exercises control over its traffic by scheduling packets to achieve higher-level objectives. We have implemented Bundler in Linux and evaluated it with real-world and emulation experiments. We find that it improves median flow completion time by between 28% and 97% across various scenarios.

1 INTRODUCTION

This paper introduces the idea of site-to-site Internet traffic control. By “site”, we mean a single physical location with tens to many thousands of endpoints sharing access links to rest of the Internet. Examples of sites include a company office, a coworking office building, a university campus, a single datacenter, and a point-of-presence (PoP) of a regional Internet Service Provider (ISP).

Consider a company site with employees running thousands of concurrent applications. The administrator may wish to enforce certain traffic control policies for the company; for example, ensuring rates and priorities for Zoom sessions, de-prioritizing bulk backup traffic, prioritizing interactive web sessions, and so on. There are two issues that stand in the way: first, the bottleneck for these traffic flows may not be in the company’s network, and second, the applications could all be transiting different bottlenecks. So what is the company to do?

Cloud computing has made the second issue manageable. Because the cloud has become the prevalent method to deploy applications today, applications from different vendors often run from a small number of cloud sites (e.g., Amazon, Azure, etc.). This means that the network path used by these multiple applications serving the company’s users are likely to share a common bottleneck; for example, all the applications running from Amazon’s US-West datacenter, all the video sessions from a given Zoom datacenter, and so on. In this setting, by treating the traffic between the datacenter site and the company site as a single aggregate, the company’s network administrator may be able to achieve their traffic control objectives.

But what about the first issue? The bottleneck for all the traffic between Amazon US-West and the company may not be the site’s access link or at Amazon, but elsewhere, e.g., within the company’s ISP; indeed, that may be the common case [13, 16, 30, 42]. Unfortunately, the company cannot control traffic when the queues are inside its ISP. And the ISP can’t help because it does not know what the company’s objectives are.¹

We propose a system, Bundler, that solves this problem. Bundler enables flexible control of a traffic bundle between a source site and a destination site by shifting the queues that would otherwise have accumulated elsewhere to the source’s site (Figure 2). It then schedules packets from this shifted queue using standard techniques [12, 15, 18, 19, 35, 38, 39, 44, 46, 48, 51] to reduce mean flow-completion times, ensure low packet delays, isolate classes of traffic from each other, etc.

The key idea in Bundler is a control loop between the source and destination sites to calculate the dynamic rate for the bundle. Rather than terminate end-to-end connections at the sites, we leave them intact and develop an “inner loop” control method between the two sites that computes this rate. The inner control loop uses a delay-based congestion control algorithm that ensures high throughput, but controls self-inflicted queueing delays at the actual bottleneck. By avoiding queues at the bottleneck, the source site can prioritize latency-sensitive applications and allocate rates according to its objectives.

By not terminating the end-to-end connections at the sites, Bundler achieves a key benefit: if the bottleneck congestion is due to other traffic not from the bundle, end-to-end algorithms naturally find their fair-share. It also simplifies the implementation because Bundler does not have to proxy TCP, QUIC, and other end-to-end protocols.

As shown in Figure 1, Bundler implements its source site and destination site functions in a sendbox and receivebox, respectively. The sendbox of one site pairs with the receivebox of another site when sending traffic to it.² These two middleboxes measure congestion signals such as the round-trip time

¹Interdomain QoS mechanisms [8, 50] have not succeeded in the Internet despite years of effort.
²One sendbox can pair with multiple receiveboxes and vice versa.
Figure 1: An example deployment scenario for Bundler in sites A and B. Traffic between the two boxes is aggregated into a single bundle, shown as shaded boxes. The sendbox schedules the traffic within the bundle according to the policy the administrator specifies (§4).

(RTT) and the rate at which packets are received, and passes these signals to a congestion control algorithm at the sendbox (§4) to dynamically compute the bundle’s sending rate. We introduce a lightweight method for the coordination between the sendbox and the receivebox that does not require any per-flow state and can be deployed in a mode that forwards packets without modification. Bundler requires no changes to the end hosts or to network routers.

Our focus thus far has been to control traffic only within a given bundle and not across different bundles. Furthermore, as we will discuss in §3, there may be instances where Bundler cannot improve performance for the bundled traffic, and falls back to the status quo; i.e., the performance achieved today when queues build in the network instead of the edge. For example, when traffic between the two sites traverses different paths with different levels of congestion, Bundler will detect this and performance will revert to the status quo, but if all of these paths have the same rates, then performance will improve.

In emulated scenarios (§7), Bundler reduces the median FCT of a representative flow size distribution between 28% to 97% across different scenarios and policies. Furthermore, performance benefits due to Bundler are within 15% of what would be achievable if in-network scheduling were a possibility. In experiments over the public Internet (§8), we find that Bundler reduces short-flow latencies by 57%.

2 RELATED WORK

Aggregating congestion information. There have been multiple proposals to aggregate congestion control information in different contexts – across flows sharing the same endpoint [5], across flows between two racks within a datacenter [52], and across flows originating from a large cloud/content provider [43]. The goal of these approaches is to share information among end-to-end congestion controllers, which allows them to better adapt to network conditions. In contrast, Bundler uses aggregate congestion control in a different way (with a different goal): as a light-weight mechanism to shift the queues from the middle of the network to the edge, without interfering with the end-to-end congestion controllers of individual flows. It is orthogonal and complementary to prior proposals on aggregate congestion control.

Using a middlebox for queue management. Remote Active Queue Management (AQM) [3] aims reduce VoIP traffic latency by deploying a middlebox at a site’s access link that drops packets or injects ECN marks for the remaining flows in order to manipulate their end-to-end congestion control loops. It makes a core assumption that the bottleneck is the site’s access link. In contrast, Bundler tackles arbitrary bottleneck locations in the middle of the network. Moreover, unlike Remote AQM, Bundler is not restricted to a specific queue management policy for a specific traffic class.

Overlay Networks. Bundler’s motivation is closer to a proposal in overlay networks, OverQoS [49], which aimed to provide QoS benefits in the Internet by enforcing traffic management policies at the nodes of an already-deployed overlay network [2]. Bundler’s approach is more lightweight; instead of relying on an overlay network, Bundler only requires each site to deploy a middlebox, and uses a novel control loop between the middleboxes to facilitate traffic management at the sites.

3 GOALS AND ASSUMPTIONS

Figure 1 describes Bundler’s deployment model. Bundler aggregates traffic from Site A to Site B, and vice-versa, into two unidirectional bundles. In the egress path, the sendbox moves the in-network queues built by the bundled traffic to itself (illustrated in Figure 2) (we describe the specific mechanism in §4). It can thus enforce desired scheduling policies across the traffic in the bundle.

Our primary goal with Bundler is to provide control over self-inflicted queueing, i.e., when traffic from a single bundle causes a queue to build up at the bottleneck links in the network, even without any other cross-traffic. In the remainder of this section, we detail the conditions in which Bundler can
Bundler detects conditions in which it cannot operate and disables itself, reverting to status-quo performance.

**Bottleneck in the middle of the network.** Network administrators already have control over packets which queue within their site. Bundler is thus useful for moving queues that build up due to congestion in the middle of the network (i.e., between a sendbox-receivebox pair). A recent measurement study [16] indicates that inter-domain bottlenecks (such as the red bottleneck link in Figure 1) do exist.

**External congestion.** Other than self-inflicted congestion, Bundler must coexist with traffic from external sources.

**Congestion due to bundled cross-traffic.** Bundler continues to provide benefits when the competing flows are part of other bundles from/to other sites because the rate control algorithm at each of the other sendboxes would ensure that the in-network queues remain small, and different bundles compete fairly with one another. Since each sendbox manages the self-inflicted queues for its own bundles, it can apply the appropriate scheduling policy in its per-bundle queues.

**Congestion due to un-bundled cross-traffic.** We now consider the scenario where the cross-traffic includes un-bundled flows. If all such un-bundled competing flows are short-lived (up to a few MBs) or application-limited (e.g., a paced video stream), the bundled traffic still sees significant performance benefits, because there are not enough packets in such short-lived flows to fill up the queues or claim a greater share of network bandwidth. However, if the cross traffic is long lasting, and employs a loss-based congestion controller to send back-logged bulk data, it aggressively fills up all available buffer space at the bottleneck link. Naively using a delay-based congestion controller at Bundler against such aggressive buffer-filling cross-traffic would severely degrade the throughput of the bundled traffic. Therefore, Bundler’s congestion controller detects the presence of buffer-filling cross-traffic and, to compete fairly, pushes more packets into the network. It thus relinquishes most of its control (and scheduling opportunities) over the bundled traffic, while still maintaining a small queue for continued detection of cross-traffic (as detailed in §5). However, such pathological buffer-filling cross traffic is rare. A recent study in CDNs [6] and our analysis of a packet trace from an Internet backbone router [10] reveal that the vast majority of connections are smaller than 1MB: too small to build persistent queues. Our experiments on Internet paths (§8), also did not encounter pathological buffer-filling cross traffic.

**Shared congestion across flows in the bundle.** Bundler’s design for moving queues via aggregate congestion control assumes that the component flows within a bundle share in-network paths, and thus congestion. To test this assumption, we used Scamper [31] to probe all paths to 5000 random IPv4 addresses from each of 30 cloud instances across the regions of public cloud providers AWS and Azure. In no cases did we find that probe packets took different AS-level paths through the network. However, we observed instances of IP-level load balancing in 26% of IP hops. In pathological scenarios with persistent imbalance in queueing between the load-balanced paths, Bundler cannot gather accurate measurements and perform aggregated delay-based rate control for the bundled traffic. Designing a new congestion control algorithm for such scenarios remains an avenue for future work. Nonetheless, Bundler can detect these scenarios (§5.2) and disable its rate control in such cases, falling back to status-quo performance.

We expect a well-implemented load balancer will work to prevent persistent imbalance from occurring; indeed, our success with using Bundler on real Internet paths (§8) suggests that such pathological cases do not occur in practice.

### 4 DESIGNING BUNDLER

Recall that in order to do scheduling, we need to move the queues from the network to the Bundler. In this section, we first describe our key insight for moving the in-network queues, and then explain our specific design choices. Recall

---

3This implies that flows within a bundle may also be short-lived requests or paced audio/video traffic which, when aggregated by Bundler, can form a heavy, long-lasting bundle.
that each site deploys one Bundler middlebox which we logically partition into sender-side (sendbox) and receiver-side (receivebox) functionality.

### 4.1 Key Insight

We induce queuing at the sendbox by rate limiting the outgoing traffic. If this rate limit is made smaller than the bundle’s fair share of bandwidth at the bottleneck link in the network, it will decrease throughput. Conversely, if the rate is too high, packets will pass through the sendbox without queuing. Instead, the rate needs to be set such that the bottleneck link sees a small queue while remaining fully utilized (and the bundled traffic competes fairly in the presence of cross traffic). We make a simple, but powerful, observation: existing congestion control algorithms calculate exactly this rate [25]. Therefore, running such an algorithm to set a bundle’s rate would reduce its self-inflicted queue at the bottleneck, causing packets to queue at the sendbox instead, without reducing the bundle’s throughput. Note that end hosts would continue running a traditional congestion control algorithm as before (e.g., Cubic [23], BBR [11]) which is unaware of Bundler. Rather, the sendbox’s congestion control algorithm acts on the traffic bundle as a single unit.

Figure 2 illustrates this concept for a single flow traversing a bottleneck link in the network. Without Bundler, packets from the end hosts are queued in the network, while the queue at the edge is unoccupied. In contrast, a Bundler deployed at the edge is able to shift the queue to its sendbox.

### 4.2 System Overview

Figure 3 shows Bundler’s sub-systems: (1) A congestion control module at the sendbox which implements the rate control logic and cross-traffic detection, as discussed in §4.3. (2) A mechanism for sending congestion feedback (ACKs) in the receivebox, and (3) a measurement module in the sendbox that computes congestion signals (RTT and receive rate) from the received feedback. We discuss options for implementing congestion feedback mechanism in §4.4 and how to use that feedback in the measurement module in §4.5. (4) A datapath for packet processing (which includes rate enforcement and packet scheduling). Any modern middlebox datapath, e.g., BESS [24], P4 [7], or Linux qdiscs (as used in our prototype implementation—see §6), is suitable. We detail the interaction between these subsystems when discussing our prototype implementation in §6. In the rest of this section, we discuss our key design choices.

### 4.3 Choice of congestion control algorithm

Bundler’s congestion control algorithm must satisfy the following requirements:

1. **Ability to limit network queueing.** Bundler must limit queueing in the network to move the queues to the sendbox. Therefore, congestion control algorithms which are designed to control delays, and thus queueing, are the appropriate choice. A loss-based congestion control algorithm which fills buffers (e.g., Cubic, NewReno), for example, is not a good choice for Bundler, since it would build up a queue at the network bottleneck and drain queues at the sendbox.

2. **Detection of buffer-filling cross-traffic.** It is well known that delay-controlling schemes (e.g., Vegas [9]) compete poorly with buffer-filling loss-based schemes [4]. Therefore, Bundler must have a mechanism to detect the presence of such competing buffer-filling flows and fall back to status quo performance, and then detect when they have left to take back its control over the network queues.

   The emergence of such detection mechanisms is recent: Copa [4] detects whether it is able to empty the queues, and Nimbus [22] provides a more general mechanism which overlays a pattern on the sending rate and measures the cross traffic’s response. Copa is not designed for aggregate congestion control (see §5); thus, we use the more general Nimbus mechanism.

### 4.4 Congestion Feedback Mechanism

A congestion control algorithm at the sendbox would require network feedback from the receivers to measure congestion and adjust the sending rates accordingly. We discuss multiple options for obtaining this.

- **Passively observe in-band TCP acknowledgements.** Conventional endhost-based implementations have used TCP acknowledgements to gather congestion control measurements. A simple strategy for Bundler is to passively observe the receiver generated TCP acknowledgements at the sendbox. However, we discard this option as it is specific to TCP and thus incompatible with alternate protocols, i.e., UDP for video streaming or QUIC’s encrypted transport header [29].

- **Terminate connections and proxy through TCP.** With this approach, one would terminate end-host TCP connections at the sendbox and open new connections to the receivebox, allowing the sendbox to control the rate of traffic in these connections. This approach can improve...
performace by allowing end-to-end connections to ramp up their sending rates quickly. The primary disadvantage of this approach is that Bundler must take responsibility for reliable delivery of component traffic, which requires large amounts of queuing and, in the case of UDP applications, can harm application performance. Furthermore, proxying TCP connections introduces a new potential point of failure at Bundler that violates fate-sharing; if Bundler crashes, connections will be lost. Finally, from a practical standpoint, to avoid head-of-line blocking this approach requires that Bundler open a new proxy connection for each component end-host connection, but still determine the bottleneck rate of the traffic aggregate. While this approach may be technically feasible [5], it would result in high overhead. Thus, we set aside TCP proxies for the remainder of this discussion, but explore their compatibility with Bundler in §7.5.

**Out-of-band feedback.** Having eliminated the options for using in-band feedback, we adopt an out-of-band feedback mechanism: the receivebox sends out-of-band congestion ACKs to the sendbox. This decouples congestion signalling from traditional ACKs used for reliability and is thus indifferent to the underlying protocol (be it TCP, UDP, or QUIC).

### 4.5 Measuring Congestion

Sending an out-of-band feedback message for every packet arriving at the receivebox would result in high communication overhead. Furthermore, conducting measurements on every outgoing packet at the sendbox would require maintaining state for each of them, which can be expensive, especially at high bandwidth-delay products. This overhead is unnecessary; reacting once per RTT is sufficient for congestion control algorithms [36]. The sendbox therefore samples a subset of the packets for which the receivebox sends congestion ACKs. We refer to the period between two successively sampled packets as an *epoch*, and each sampled packet as an *epoch boundary packet*.

The simplest way to sample an epoch boundary packet would be for the sendbox to probabilistically modify a packet (i.e., set a flag bit in the packet header) and the receivebox to match on this flag bit. However, where in the header should this flag bit be? Evolving packet headers has proved impractical [34], so perhaps we could use an encapsulation mechanism. Protocols at both L3 (e.g., NVGRE [21], IP-in-IP [41]) and L4 (e.g., VXLAN [32]) are broadly available and deployed in commodity routers today.

Happily, we observe that such packet modification is unnecessary; since the same packets pass through sendbox and receivebox, uniquely identifying a given pattern of packets is sufficient to meet our requirements. In this scheme, the sendbox and receivebox both hash a subset of header fields with a fixed sampling period to produce about 4 samples per RTT and communicates it to the receivebox. Upon identifying a packet as an epoch boundary packet the receivebox records: (i) its hash, \(h(p_i)\), (ii) the time when it is sent out, \(t_{sent}(p_i)\), and (iii) the total number of bytes sent thus far including this packet, \(b_{sent}(p_i)\). When the receivebox sees \(p_i\), it also identifies it as an epoch boundary and sends a congestion ACK back to the sendbox. The congestion ACK contains \(h(p_i)\) and the running count of the total number of bytes received for that bundle. Upon receiving the congestion ACK for \(p_i\), the sendbox records the received information, and using its previously recorded state, computes the RTT and the rates at which packets are sent and received, as in Figure 4. In a micro-benchmark evaluating the fidelity of our measurements, 80% of our RTT estimates were within 1.2ms of the actual value and 80% of our receive rate estimates were within 4Mbps of the actual value.

The packet header subset that is used for identifying epoch boundaries must have the following properties: (i) It must be the same at both the sendbox and the receivebox. (ii) Its values must remain unchanged as a packet traverses the network from the sendbox to the receivebox (so, for example, the TTL field must be excluded). (iii) It differentiates individual *packets* (and not just flows), to allow sufficient entropy in the computed hash values. (iv) It also differentiates a retransmitted packet from the original one, to prevent spurious samples from disrupting the measurements (this precludes, for example, the use of TCP sequence number). We expect that the precise set of fields used will depend on specific

---

\(^{5}\)The idea of an out-of-band congestion notification is similar to that in QCN [1], although our goals differ.
deployment considerations. For example, in our prototype implementation (§6) we use a header subset of the IPv4 IP ID field and destination IP and port. We make this choice for simplicity; it does not require tunnelling mechanisms and is thus easily deployable, and if Bundler fails, connections are unaffected. We note that previous proposals [45] have used IP ID for unique packet identification. The drawback of this approach is that it cannot be extended to IPv6. To support a wider set of scenarios, Bundler could use dedicated fields in an encapsulating header (as in [33]).

Note that our congestion measurement technique is robust to a boundary packet being lost between the sendbox and the receivebox. In this case, the sendbox would not get feedback for the lost boundary packet, and it would simply compute rates for the next boundary packet over a longer epoch once the next congestion ACK arrives.

4.6 Implications of Bundler’s Design

Our design choices result in an architecture where Bundler’s inner rate control loop can be implemented entirely the “control plane” of the sendbox and the receivebox, which passively observes the packets traversing the datapath of the middleboxes, without modifying them. This results in a truly transparent system, that is light-weight, has low overhead, preserves fate-sharing, and in no way interferes with the end-to-end controllers of individual flows. The only datapath action that Bundler performs is the enforcement of the desired scheduling and queue management policies at the sendbox.

5 UNFAVORABLE CONDITIONS

Recall from §3 that Bundler can reliably shift queue build up from the bottleneck to itself when, (a) the cross-traffic is not buffer-filling, and (b) all of its component traffic shares the same bottleneck in the network. In practice, either of these conditions may break. In this section, we describe how Bundler can re-use the same measurements from §4.5 to detect when these conditions do not hold. In such cases, Bundler (temporarily) disables its rate limiting (falling back to status quo performance) until favorable conditions arise again.

5.1 Buffer-Filling Cross Traffic

It is well known that delay-based congestion control algorithms (as Bundler uses) lose throughput when competing with traditional loss-based controllers [4]. Therefore, in order to compete fairly with buffer-filling cross-traffic, Bundler must first detect the presence of such traffic and disable its use of a delay-based controller.

Prior work (Nimbus [22]) presents a method for detecting buffer-filling cross-traffic, that Bundler employs. But what exactly should the sendbox do when it detects this condition? A naive approach might replace the delay-based control at Bundler with a loss-based buffer-filling congestion control such as Cubic. However, with this approach, Bundler must measure the number of flows in a bundle, and should compete as aggressively in proportion to achieve an aggregate throughput equivalent to status quo [14]. On high-performance datapaths, it may be difficult to measure this number [47].

We propose a simpler solution. Since bundles comprise of traditional end-host connections, with their own congestion controllers, Bundler can simply let the traffic pass, i.e., increase the pacing rate at the sendbox to stop controlling queues. Then, the end-host congestion control loop will naturally compete fairly with the aggressive cross traffic, just as traffic in the Internet does today.

This brings us to the next natural question: How can the sendbox know it is safe to resume delay-control (and scheduling) after disabling it? One approach is to send passive probes along the network to detect the presence/absence of queuing. However, such passive measurements cannot distinguish between the self-inflicted queuing due to Bundler’s traffic and the queuing due to cross-traffic. If the bottleneck queue is entirely self-inflicted, it is safe (and desirable) to resume delay-control and scheduling. Therefore, it is important to actively probe, that is, change the rate of bundled traffic and observe the response of the cross traffic. This is what the Nimbus mechanism does. At a high level, Nimbus sinusoidally varies the sending rate $r(t) = A \sin\left(\frac{2\pi t}{T}\right)$ during the up-pulse, where $A$ is the pulse amplitude (set to one-fourth of the estimated bottleneck bandwidth) and $T$ is the pulse duration, and measures the cross traffic’s response in the frequency domain. The sendbox can use the Nimbus algorithm to detect when to relinquish control over the queue by interposing this sending pattern over the delay-controller’s rate decisions. However, if the sendbox entirely drains its queues into the network, it will no longer be possible for Nimbus to overlay pulses onto the traffic pattern, and it will be unable to determine the nature of the cross traffic. Practically, this would mean that once sendbox switches to compete with cross traffic, it would never gain the information necessary to switch back.

Instead, to support active probing while also letting the traffic pass, the sendbox must maintain some queueing. How many packets should this be? The sendbox should be able to generate enough packets for a Nimbus up-pulse, i.e., the area under the up-pulse curve: $A \int_{-\frac{T}{2}}^{\frac{T}{2}} \sin \left(\frac{2\pi t}{T}\right) dt = \frac{A T}{2\pi}$. From Nimbus, we use $T = 0.2$ seconds and $A$ is as above, one-fourth the bottleneck bandwidth. By Little’s law, we can calculate the amount of extra queueing: $\frac{T}{8\pi}$, or 8ms. We thus configure the sendbox to hold back 10ms of queueing for active probing; the additional queueing is a cushion against input variance. Note that this extra queueing is in addition to queueing in the network. As a result, the end-to-end Cubic connections will see mild RTT inflation. In §7.3 we show that this effect is not large; Bundler still achieves performance comparable to the status quo.
How should we achieve this target queueing delay? This problem is similar to the role of the PIE AQM mechanism [39], which also seeks to maintain a queuing delay target. Correspondingly, we design a PI controller at the sendbox as part of the fairness control module. It overloads a rate $r$ corresponding to $\dot{r}(t) = \alpha(q - qr) + \beta(q)$, where $q$ is the queue size and $qr$ is the target queue size computed above. We pick $\alpha = 10$ and $\beta = 10$ by solving for a convergence time of one RTT.

5.2 Imbalanced Multipathing

Since a bundle contains many component connections, a load balancer may send them along different paths. If the load along different paths is well-balanced, Bundler will accurately treat a load-balanced bottleneck link as a single link with the aggregate rate of each sub-link. However, when the load along different paths is imbalanced, the series of measurements at the Bundler will be a random sampling of the different paths, which would confuse the delay-control algorithm and cause it to perform poorly. Fortunately, such cases are straightforward to detect with our measurement technique. More specifically, load imbalance will result in many epoch measure packets arriving out-of-order at the receivebox (whenever epoch packet $i$ happens to traverse a path with a larger delay than epoch packet $i+1$), and consequently, out-of-order "congestion ACKs" at the sendbox. Figure 6 shows this effect for an emulated imbalance scenario. Therefore, we use the fraction of epoch measurement packets that arrive out-of-order as an indicator of load imbalance due to multipathing. If this number is small, the links are roughly balanced and Bundler will operate correctly. If it is large, it indicates load imbalance, in which case Bundler’s rate-control would work incorrectly. Therefore, if the reordering level is above 5%, we disable rate control and revert to status quo performance. We evaluate this approach (and justify our chosen threshold) in §7.6.

6 IMPLEMENTATION

Bundler boxes can be implemented as described below (although the specific implementations could vary across deployments).
We now describe our prototype implementation of Bundler.

**Sendbox data plane.** We implement it using Linux tc [28]. We patch the TBF queueing discipline (qdisc) [27] to detect epoch boundary packets and to report them to the control plane using a netlink socket. We use the FNV hash function [20], a non-cryptographic fast hash function with a low collision rate, to compute the packet header hash for identifying epoch boundaries. This hash function, comprising 4 integer multiplications, is the only additional per-packet work the data plane must perform to support Bundler; in our experiments, it had negligible CPU overhead.

We patch TBF’s inner_qdisc to support any qdisc-based traffic controller. By default, TBF instantaneously re-fills the token bucket when the rate is updated; we disable this feature to avoid rate fluctuations caused by our frequent rate updates. Our patch to the TBF qdisc comprises 112 lines of C.

**Sendbox control plane.** We implement it to run in user-space in 1365 lines of Rust. We use CCP [36] to run different congestion control algorithms (described next). CCP is a platform for expressing congestion control algorithms in an asynchronous format, which makes it a natural choice for our epoch-based measurement architecture. The control plane uses libccp [36] to interface with the congestion control algorithm, and libnl to communicate with the qdisc.

**Congestion control algorithms.** We use existing implementations of congestion control algorithms (namely, Nimbus [22], Copa [4] and BBR [11]) on CCP to compute sending rates at the sendbox. If the algorithm uses a congestion window, the sendbox computes an effective rate of \( \frac{CWND}{RTT} \) and sets it at the qdisc. We validated that our implementation of these congestion control schemes at the sendbox closely follows their implementation at an endhost.

### 6.2 Bundler Event Loop

Figure 5 provides an overview of how our Bundler implementation operates on an already-established bundle.

1. In the datapath, packets arrive at the sendbox qdisc. (2) The qdisc determines whether a packet matches the epoch boundary condition (§4.5). If so, it sends a netlink message to the control plane process running in user-space, and then forwards the packet normally (note the datapath does not send packets to user-space). (3) The receivebox observes the same epoch boundary packet via libbcp. (4) It sends an out-of-band UDP message to the sendbox that contains the hash of the packet and its current state. (5) The sendbox receives the UDP message, and uses it to calculate the epochs and measurements as described in §4.5. (6) Asynchronously, the sendbox control plane invokes the congestion control algorithm every 10ms [36] via libccp. (7) The sendbox control plane communicates the rate, if updated, to the qdisc using libnl. Finally (8), if the sendbox changes the desired epoch length based on new measurements, it communicates this to the receivebox, also out-of-band.

### 7 EVALUATION

Given Bundler’s ability to move the in-network queues to the sendbox (as shown earlier in Figure 2), we now explore:

1. Where do Bundler’s performance benefits come from? We discuss this in the context of improving the flow completion times of Bundler’s component flows. (§7.2)
2. Do Bundler’s performance benefits hold across different scenarios? (§7.3)
3. Can Bundler work with different congestion control algorithms (§7.4)?
4. Are Bundler’s core ideas still applicable with other design decisions? (§7.5)
5. Is Bundler’s heuristic (§5.2) for detecting imbalanced multipath scenarios robust? (§7.6)
6. Can Bundler effectively control the queues on real Internet paths? (§8)

### 7.1 Experimental Setup

We use network emulation via mahimahi [37] to evaluate our implementation of Bundler in a controlled setting; we also present results on real Internet paths in §8. There are three 8-core Ubuntu 18.04 machines in our emulated setup: one machine is a sender, an sendbox, and a receiver. The receivebox runs on the same machine as the receiver. Since our receivebox implementation uses libbcp, GRO would change the packets before they are delivered to the receivebox, which would cause inconsistent epoch boundary identification between the two boxes. We, therefore, disable TSO and GRO. Nevertheless, throughout our experiments CPU utilization on the machines remained below 10%.

Unless otherwise specified, we emulate the following scenario. A many-threaded client generates requests from a request size CDF drawn from an Internet core router [10] and assigns them to one of 200 server processes. The workload is heavy-tailed: 97.6% of requests are 10KB or shorter, and the largest 0.002% of requests are between 5MB and 100MB. Each server then sends the requested amount of data to the client and we measure the flow completion time of each such request. The link bandwidth at the mahimahi link is set to 96Mbps, and the RTT is set to 50ms. The requests result in an offered load of 84Mbps.

The endhost runs Cubic [23], and the sendbox runs Copa [4] (we test other schemes in §7.4) with Nimbus [22] for cross traffic detection. Each experiment is comprised of 1,000,000 requests sampled from this distribution, across 10 runs each with a different random seed. The sendbox schedules traffic using stochastic fair queueing [35] in our experiments.
7.2 Understanding Performance Benefits

We first present results for a simplified scenario without any cross-traffic, i.e., all traffic traversing through the network is generated by the same customer and is, therefore, part of the same bundle. This scenario highlights the benefits of using Bundler when the congestion on the bottleneck link in the network is self-inflicted. We explore the effects of congestion due to other cross-traffic in §7.3.

**Using Bundler for fair queueing.** In this section, we evaluate the benefits provided by doing fair queuing at the Bundler, and use median slowdown as our metric, where the “slowdown” of a request is its completion time divided by its completion time would have been in an unloaded network. A slowdown of 1 is optimal, and lower numbers represent better performance.

We evaluate three configurations: (i) The “Status Quo” configuration represents the status quo: the sendbox simply forwards packets as it receives them, and the mahimahi bottleneck uses FIFO scheduling. (ii) The “In-Network” configuration deploys fair queueing at the mahimahi bottleneck. Recall from §1 that this configuration is not deployable. (iii) The default Bundler configuration, that uses stochastic fair queueing [35] scheduling policy at the sendbox, and (iv) Using Bundler with FIFO (without exploiting scheduling opportunity).

Figure 7 presents our results. The median slowdown (across all flow sizes) decreases from 1.76 for Baseline to 1.26 with Bundler, 28% lower. In-Network’s median slowdown is a further 15% lower then Bundler: 1.07. Meanwhile, in the tail, Bundler’s 99%ile slowdown is 41.38, which is 48% lower than the Status Quo’s 79.37. In-Network’s 99%ile slowdown is 27.49.

**Using Bundler for other policies.** We additionally evaluated other scheduling and queue management policies with Bundler. We omit detailed results for brevity, and present a few highlights. With FQ-CoDel [17], Bundler can achieve 97% lower median end-to-end RTTs and 89% lower 99%ile RTTs. By strictly prioritizing one traffic class over another, Bundler results in 65% lower median FCTs for the higher-priority class.

**Aggregate congestion control is not enough.** It is important to note that Bundler’s congestion control by itself (i.e., running FIFO scheduling) is not a means of achieving improved performance. To see why this is the case, recall that Bundler does not modify the endhosts: they continue to run the default Cubic congestion controller, which will probe for bandwidth until it observes loss. Indeed, the packets endhost Cubic sends beyond those that the link can transmit must queue somewhere in the network or get dropped. Without Bundler, they queue at the bottleneck link; with Bundler, they instead queue at the sendbox. In addition, the delay-based congestion controller at sendbox also maintains a small standing queue at the bottleneck link (which can be seen in Figure 2) to avoid under-utilization, which increases the end-to-end-delays slightly. Therefore, doing the FIFO scheduling at the Bundler, as is done by the Status Quo, results in slightly worse performance.

7.3 Impact of Cross Traffic

Can Bundler successfully revert to status-quo performance in the presence of buffer-filling cross traffic, then resume providing benefits once that cross traffic leaves? In Figure 8, we show this scenario. At first, the link is occupied only by Bundler’s traffic, similar to the setup described in §7.1. At time \(t = 60\) sec, a buffer-filling cross traffic flow arrives. Bundler detects its presence (indicated by the gray shading) and starts pushing more packets into the network to compete fairly, reverting back to performance that is slightly worse than Status Quo (median FCT for short flows is 12% higher). Performance is slightly worse because of the 10ms queue that Bundler continues to maintain at its sendbox for active probing to detect the cross-traffic’s departure, as described in §5. At time \(t = 120\) sec, the buffer-filling flow stops and non-buffer-filling traffic starts, generated from the same distribution as Bundler as described in §7.1. Bundler correctly detects that it is safe to resume delay-control, and continues providing scheduling benefits. For the remainder of this subsection, we present three micro-benchmarks which dig deeper into the latter

---

7We implement this scheme by modifying mahimahi (our patch comprises 171 lines of C++) to add a packet-level fair-queueing scheduler to the bottleneck link.

8We believe the benefits provided by Bundler in the more common regime with no competing buffer-filling cross traffic are substantial enough to make up for slight degradation in these specific scenarios.
two scenarios, where cross traffic can affect Bundler’s performance. We present results with both Nimbus and Copa being used as the congestion control algorithm at the sendbox.

**Short-lived flows.** We first consider the case where the cross traffic comprises of short-lived flows up to a few MBs. We keep Bundler’s offered load (comprising of web requests described in §7.1) constant, but vary the cross traffic’s offered load. Figure 9 shows that Bundler continues to provide benefits for this case, since maintaining low in-network queues is still possible in the presence of such flows.

**Buffer-filling flows.** We now evaluate how Bundler’s throughput is impacted due to competition from varying amount of buffer-filling cross-traffic. As discussed in §5, in this (rare) scenario, we expect component flows in a bundle to experience slightly higher RTTs compared to Status Quo. Indeed, Figure 10 shows that the component flows in the bundle experience 18% less throughput on average. The impact (which stems from the extra queuing at the sendbox) varies from 12% lower throughput with 10 competing flows to 22% lower with 50.

**Competing Bundles.** Last, we evaluate the case where flows from multiple bundles compete with one another. In Figure 11, we show the performance with two bundles of traffic competing with one another at the same bottleneck link. Both bundles comprise of web requests along with a backlogged Cubic flow. Both bundles maintain low queueing in the network and successfully control the queues at the

Figure 8: Bundler’s scheduling ability depends on the characteristics of the cross traffic over time. In this experiment, there are 3 periods: from 0 to 60 sec., there is no competing traffic, from 60 to 120 sec. there is buffer-filling cross traffic, and from 120 to 180 sec. there is non-buffer-filling cross traffic. The box-plots below each period show the distribution of short flow FCTs during that time. During the period with buffer-filling cross traffic, Bundler detects its presence and competes fairly. The shaded region in the Bundler graph shows the output of the cross-traffic detector.

Figure 9: Against cross traffic comprising of short lived flows. Bundler offers 48Mbps of load to the bottleneck queue. The cross traffic’s offered load increases along the x-axis, while Bundler’s offered load remains fixed.

Figure 10: Varying number of competing buffer-filling cross traffic flows. Bundler controls a fixed 20 buffer-filling flows in each case.

Figure 11: Competing traffic bundles. In both cases, the aggregate offered load is 84Mbps, as in Figure 7. For "1:1", we evenly split the offered load between the two Bundles; for "2:1", one bundle has twice the offered load of the other. In both cases, each bundle observes improved median FCT compared to its performance in the baseline scenario.
sendbox. Thus, Bundler provides benefits for both bundles, even when the amount of traffic in each bundle is different.

### 7.4 Impact of Congestion Control

We now evaluate the impact of a different congestion control algorithm running at the sendbox and at the endhosts.

**Sendbox congestion control.** So far we have evaluated Bundler by running Copa [4] at the sendbox. Figure 12 shows Bundler’s performance with other congestion control algorithms (namely, Nimbus’s BasicDelay [22] and BBR [11]), and using SFQ scheduling. We find that using BasicDelay provides similar benefits over Status Quo as Copa. BBR, on the other hand, performs slightly worse than Status Quo. This is because it pushes packets into the network more aggressively than the other schemes, resulting in a bigger in-network queue. This, combined with the queue built at the Bundler, results in higher queueing delays than Status Quo. This shows that the choice of congestion control algorithm, and its ability to maintain small queues in the network, plays an important role.

**Endhost congestion control.** We used Cubic congestion control at the endhosts for our experiments so far. When we configure endhosts to use Reno or BBR, Bundler’s benefits remain: Bundler achieves 58% lower FCTs in the median compared to the updated Status Quo where the endhosts use BBR. This shows that Bundler is compatible with multiple endhost congestion control algorithms.

### 7.5 Terminating TCP Connections

Although our Bundler prototype does not terminate connections (as discussed in §4.3), we note that terminating connections does provide one key advantage: the end-to-end congestion controller will observe a smaller RTT, since the proxy can acknowledge its segments much faster than the original receiver. This enables rapid window growth at the endhosts. While there are, of course, operational concerns with managing the resulting queue, it does provide additional scheduling opportunities as well as faster ramp-up for midsized connections.

How much benefit, then, could a proxy-based Bundler provide? To evaluate this, we emulate an idealized TCP proxy by modifying the endhosts to maintain a constant congestion window of 450 packets—slightly larger than the bandwidth-delay product in our setup—and increasing the buffering at the sendbox to hold these packets. The other aspects of Bundler remain unchanged. The result is in Figure 13.

For the short requests which never leave TCP slow start, terminating TCP connections does not yield additional benefits: with or without termination, they finish in a few RTTs. For medium-to-long requests, terminating TCP connections yields additional benefits since they no longer incur the penalty of window growth. Therefore, a site may benefit from proxying TCP connections at Bundler if its traffic pattern contains many medium-sized flows which benefit from fast ramp-up.

### 7.6 Multipath Detection

As described in §5.2, when the ratio of out-of-order to in-order measurements is above a certain threshold, it indicates that Bundler’s component flows are likely traversing multiple imbalanced paths. To evaluate the extent to which this heuristic corresponds with imbalance, we re-run the emulation experiment from Figure 8 for a variety of network conditions (bottleneck bandwidth ranging from 12 to 96 Mbps, end-to-end RTTs ranging from 10 to 300 ms, and bottleneck load-balancing from 1 to 32 paths) and consider the average value reported by the heuristic over each experiment. The maximum value reported across all experiments with a single path was 0.4%, while the minimum value reported across all experiments with 2-32 paths was 20%, two orders of magnitude greater. Thus, this heuristic provides a very clear separation between single and multiple path scenarios and a simple threshold is sufficient.

### 8 REAL INTERNET PATHS

We next evaluate our prototype implementation on real Internet paths to answer three key questions that our emulated setup did not answer: (1) Does congestion (and queuing) actually occur in the middle of the network? (2) Is buffer-filling cross traffic rare enough in practice to ensure that Bundler provides a net improvement in performance? and (3) Is Bundler’s
Figure 14: On 5 real-Internet paths, Bundler achieves lower latencies than Status Quo for latency-sensitive traffic. Each bar depicts an individual 5-tuple; load-balancing in the Internet prevents queueing for some 5-tuples. Bundler still offers scheduling for paths with queueing (achieving 57% lower latencies overall) while achieving overall throughput within 1% of that in the Status Quo scenario.

**Experiment Setup.** We deploy Bundler (sendbox) in a GCP datacenter in Iowa and generate traffic from multiple different machines in this datacenter (as detailed below). The generated traffic is sent to multiple machines in five different GCP datacenters (in Belgium, Frankfurt, Oregon, South Carolina, and Tokyo). We configured GCP to route traffic over the public Internet rather than a private network. We deploy a Bundler (receivebox) in each of these receiving datacenters, thus resulting in a total of five bundles spanning different regions of the globe. We evaluate two different workloads in this setup: (i) Each bundle comprising of 10 parallel closed-loop 40 bytes UDP requests, where the sender issues a new request every time it receives a response. We measure the request-response RTTs in this workload to use as a baseline (and call them Base RTTs). (ii) We add 20 backlogged (iperf) flows to the above workload in each bundle. We run this workload both with and without Bundler and measure the UDP request-response RTTs (represented as Bundler and Status Quo respectively). Effective SFQ across all flows with Bundler should not inflate the base request-response RTT. We verified that the backlogged senders achieve similar throughput in all cases (2-4Gbit/s on these paths) both with and without Bundler, and that the Bundler machine in Iowa is not a bottleneck itself.

**Result.** Figure 14 shows, for each of the five bundles, the resulting RTT box-plots for each of the ten request-response loops (with the 5 tuples in UDP/IP headers differing across all ten). We make the following key observations: (i) The Status Quo RTTs are significantly higher than the Base RTT, which indicates significant in-network queueing. (ii) Bundler is able to move these queues and enforce SFQ scheduling effectively, resulting in request-response RTTs comparable to Base RTTs, and 57% smaller than Status Quo at the median.

**9 DISCUSSION**

**Composability.** Bundles are naturally *composable*: a sub-site within site A can deploy its own Bundler to take control of its fraction of the in-network queues, with the site A’s Bundler enforcing a scheduling policy across the bundled traffic from each sub-site. For example, a department within an institute may bundle its traffic to a collaborating department in another institute, with the parent institutes bundling the aggregate traffic across multiple departments.

**Scheduling across different bundles at a sendbox.** We evaluate benefits of scheduling within a bundle. In practice, a given sendbox will see traffic from multiple bundles. Extending different scheduling policies to multiple such bundles can be done trivially.

**Rate allocation across different competing bundles.** When multiple bundles (belonging to different sites) compete at the same bottleneck, Bundler’s congestion control would ensure a fair rate allocation across each of these bundles, irrespective of the amount of traffic in them. It, therefore, provides fairness on per-site basis, as opposed to a per-flow basis, making it more robust to popular end-host strategies such as opening multiple connections to increase bandwidth share.

**10 CONCLUSION**

We have described Bundler, a new type of middlebox which uses a novel “inner” congestion control loop for traffic bundles between two sites to shift the queues from the middle of the network, where it is difficult to unilaterally express traffic control policy, to the site itself, where doing so is tractable. Bundler neither maintains any per-flow state, nor makes any modifications to the packets. We demonstrate, using both emulated network experiments and real Internet paths, that it is possible to shift queues and schedule packets to an extent sufficient to enforce well-known scheduling disciplines.
ACKNOWLEDGMENTS
We thank Srinivas Narayana, Ahmed Saeed, and Rachee Singh for their helpful discussions and feedback. This work is supported in part by DARPA contract HR001117C0048 and NSF grants 1526791, 1563826, and 1407470.
[31] L. Zhang. Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks. In SIGCOMM, 1990.

[32] D. Zhuo, Q. Zhang, V. Liu, A. Krishnamurthy, and T. Anderson. Rack-Level Congestion Control. In HotNets, 2016.