Virtualized Congestion Control (Extended Version)

Bryce Cronkite-Ratcliff[1,2], Aran Bergman[3], Shay Vargaftik[3], Madhusudhan Ravi[1],
Nick McKeown[2], Ittai Abraham[1], Isaac Keslassy[1,2,3]
[1] *VMware* [2] *Stanford* [3] *Technion*

ABSTRACT

New congestion control algorithms are rapidly improving data centers by reducing latency, overcoming incast, increasing throughput and improving fairness. Ideally, the operating system in every server and virtual machine is updated to support the new congestion control algorithms. However, legacy applications often cannot be upgraded to a new operating system version, which means the advances are off-limits to them. Worse, as we show, legacy applications can be squeezed out, which in the worst case prevents the entire network from adopting new algorithms.

Our goal is to make it easy to deploy new and improved congestion control algorithms into multitenant data centers, without having to worry about TCP-friendliness with non-participating virtual machines. This paper presents a solution we call *virtualized congestion control*. The data center owner may introduce a new congestion control algorithm in the hypervisors. Internally, the hypervisors translate between the new congestion control algorithm and the old legacy congestion control, allowing legacy applications to enjoy the benefits of the new algorithm. We have implemented proof-of-concept systems for virtualized congestion control in the Linux kernel and in VMware's ESXi hypervisor, achieving improved fairness, performance, and control over guest bandwidth allocations.

1. INTRODUCTION

The rise of hyperscale datacenters has driven a huge growth in network communications. Because the large datacenter companies control both ends of the internal connections, they are now deploying new congestion control algorithms, either published (e.g. TCP with ECN, DCTCP, TIMELY, etc. [1–15]) or proprietary, to reduce latency and flow completion times for their traffic. This trend seems likely to continue, as datacenter companies seek ways to maximize utilization of their network, by customizing their network's behavior to best serve their large distributed applications.

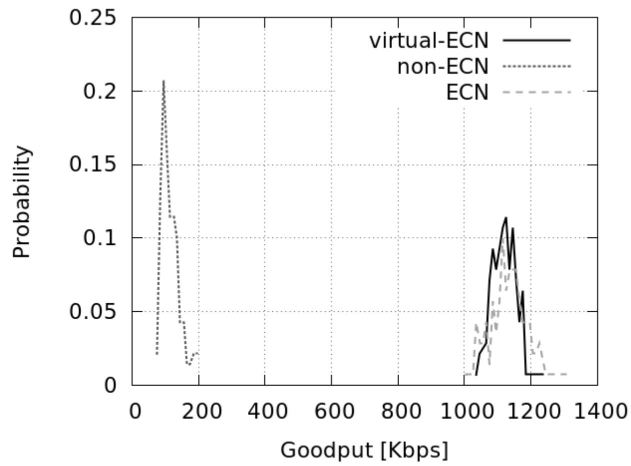


Figure 1: 8 TCP flows with ECN, one non-ECN flow and one virtual-ECN flow run on the same bottleneck link. The figure plots the frequency of their average goodput. The non-ECN flow is starved, reaching only 10% of ECN goodput on average. After translation to virtual-ECN, the average goodput is almost identical to that of ECN.

Multitenant datacenters, in which many tenants lease and share a common physical infrastructure to run their virtual machine (VM) workloads, have a harder problem to solve [5]: their tenants implement their *own* congestion control algorithms. Yet, the multitenant datacenter owner must ensure that all the algorithms still play well together, so as to meet agreed-upon SLAs. Given the complex interplay between different congestion control algorithms (just think how hard it is to make a single new algorithm "TCP-friendly"), what is a multitenant datacenter owner to do?

A few approaches come to mind. The multitenant datacenter owner can strictly divide the bandwidth among the tenants, giving each a fixed allocation [16]. However, this prevents statistical sharing of unused bandwidth. Another approach is to modify all the datacenter switches and tweak the fairness rules between tenants at each switch, for in-

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

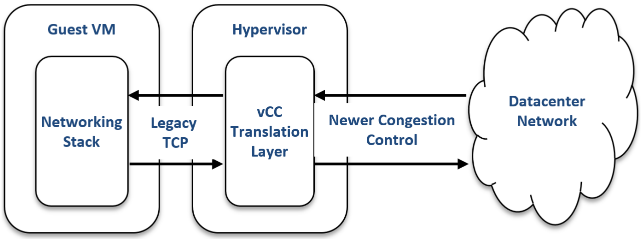


Figure 2: vCC (virtualized congestion control) translation layer implemented in the hypervisor.

stance by implementing separate queues or applying different marking thresholds within the same queue [5,17]. Unfortunately, as the number of tenant algorithms increases, this approach becomes harder to deploy while still guaranteeing fairness. Instead, we adopt a different approach, taking advantage of the fact that all traffic passes through hypervisors controlled by the multitenant datacenter owner. As illustrated in Figure 2, what if a translation layer in the hypervisors ensured that the whole datacenter uses a single best-of-breed congestion control algorithm, while giving the illusion to each of the VM guests that it keeps using its own congestion control algorithm? In other words, the guest congestion control algorithm is an *overlay* algorithm, while the hypervisor translates it (transparently) to the *underlay* algorithm used in the datacenter network. We call this approach *virtualized congestion control*, or vCC.

A common manifestation of the problem we are trying to solve is when a guest VM contains a legacy application running on a legacy operating system (OS) version with an old TCP version (e.g. TCP Reno). The application has been functioning correctly for years, with little or no maintenance, but has recently been moved to the cloud. If other VMs are using more modern ECN-aware congestion control algorithms, they can starve the older application's traffic, as seen in Figure 1.[1]

It is frequently impractical to port the old application to a newer OS, and out of the question to force new applications to use outdated congestion control algorithms. Our approach solves the problem by allowing both types of applications to enjoy the benefits of ECN. As shown in Figure 1, while a non-ECN TCP flow can get starved when running alongside many ECN flows, its *virtual-ECN* augmentation with our vCC translation layer provides significantly increased fairness. Specifically, the vCC translation layer in the hypervisor (a) modifies the TCP header fields of the sent packets to enable ECN support in the underlay; (b) upon receiving an ECN congestion notification, decreases the receiving window to force the overlay TCP guest sender to reduce its pace and behave *as if* it were ECN-aware; and (c) modifies the TCP header fields of the ACK packets to mask congestion notifications in the overlay.

The longer-term goal of our vCC datacenter solution is to be able to introduce a new best-of-breed underlay congestion control algorithm that is implemented in the hypervisor and is decoupled from the congestion control algorithm in the overlay guest OS. The new underlay algorithm would not need to limit itself to be TCP-friendly or legacy-friendly, and therefore may be even more efficient than the existing ones. This vCC architecture should seamlessly support arbitrary legacy guest OSes and congestion control algorithms. The software implementation of vCC at the hypervisor allows update of the datacenter-wide congestion control algorithm without changes in the hardware switches or guest VMs. Finally, since each hypervisor can determine the application and tenant that generated each of its flows, vCC can implement congestion control algorithms that generalize fairness among flows to fairness and resource allocation among both tenants and applications.

Fundamentally, we view our problem as an instance of *algorithmic virtualization*. While *resource virtualization* is about sharing a common resource and making each guest believe that it keeps using its resource privately, algorithmic virtualization is about implementing a common algorithm while making each guest believe that it keeps using its private algorithm. In our setting, we provide an algorithmic virtualization of congestion control. The hypervisor implements a common best-of-breed congestion control algorithm while allowing each guest to keep using its private legacy congestion control algorithm. Formally, a congestion and flow control algorithm is a function from a sequence of input events (e.g., ACKs or receiving window sizes from the network, or new data from the application layer) to a sequence of output events (releasing packets to the network). Given an input sequence x, we define our *target* output $f(x)$ as the output obtained by the target datacenter congestion control algorithm f. The goal of our hypervisor translation layer T is to map input x into $T(x)$ so that the private guest congestion control algorithm g applied to the modified input $T(x)$ yields the same target output, i.e.

$$g(T(x)) = f(x). \tag{1}$$

In this paper, we propose to add a translation layer at the hypervisor that will virtualize the congestion control. While legacy applications implemented in VMs will continue to run their legacy TCP implementations, the hypervisor will translate this legacy TCP into a newer congestion control algorithm *under-the-hood*. As a result, it can provide a large set of benefits (e.g. ECN awareness, Selective ACK, smaller timeouts, etc.) to all legacy applications. It will ensure that all datacenter applications are afforded the same benefits, resulting in a similar expected performance and therefore an increased fairness. At the same time, unlike a TCP proxy [18, 19], it can also support solutions that respect the

[1]The data presented here represents 140 runs of the experiment. Each run was 37-seconds long; the first 5 and last 2 seconds were not included in the average to avoid experiment start-up and tear-down effects. Each of the 10 senders is connected to a single switch, which is connected to a single receiver by a single (bottleneck) link. The links all have 10 Mbps bandwidth and 250 μs delay per direction. The virtual-ECN flow was provided by our Linux vCC translation layer, described in Section 3. The switch's port connected to the receiver was configured with the RED1 parameter presented in Table 1 of Section 3.

end-to-end semantics by avoiding any acknowledgments of packets unreceived by the destination—although we could equally implement our ideas using a TCP-proxy-like solution. In particular, our contributions are as follows:

Techniques. In Section 2, we consider a wide range of techniques that the hypervisor can implement, and discuss the tradeoffs between their implementation complexity and the potential benefits that they can provide. For instance, an algorithm that allows the hypervisor to directly modify the guest memory essentially enables it to replace the whole networking stack, but at the cost of complex implementations costs. Instead, we suggest more lightweight approaches that provide a limited set of benefits. For instance, if the hypervisor can update the receive window field in ACKs, then we prove that it can provide DCTCP-like or ECN-like properties to a simple TCP congestion control.

Fairness in mixed-ECN environments.[2] In Section 3, we show that a minority of non-ECN legacy flows can get starved by a majority of ECN flows. However, we demonstrate that fairness can be achieved by using our Linux-based vCC translation layer to make non-ECN flows ECN-capable. In addition to restoring fairness, we provide the benefits of ECN to the non-ECN flows, i.e. achieve high link utilization without dropping and retransmitting packets.

Dynamic hypervisor-based TCP bandwidth sharing. In Section 4, we present a proof-of-concept VMware ESXi implementation of the vCC translation layer. We show that this vCC layer is capable of dynamically throttling traffic using the TCP receive window, and therefore provide preferential treatment to certain applications without queueing or dropping packets in the network.

Discussion. In Section 5, we discuss the architectural roadblocks to the implementation of our vCC solution in datacenters.

2. HYPERVISOR TRANSLATION TECHNIQUES

In this section, we look at several available TCP modification techniques that may be used in our vCC architecture. In particular, we borrow several existing techniques that were proposed in the literature to regulate the rate of TCP flows. The techniques are roughly ordered from the most to least intrusive ones. The first two techniques are specific to hypervisors, while those following can also be used in a network edge middlebox. The simpler and less intrusive techniques are most appealing and practical, and we focus on those in the paper.

In addition, we detail the cons of each technique, and in particular the ways it needs to spoof and bend or break architectural principles or semantics. *Not all lies are created equal*: breaking the end-to-end principle can be considered as more severe than merely reducing the receiver window.

Write into guest memory. Modern virtualization techniques such as active memory introspection [21, 22] and industry products such as VMware's VMSafe [23] enable the hypervisor to securely monitor a guest VM by having complete visibility over its raw memory state, and write into this memory when needed. Therefore, the hypervisor could directly modify the congestion control algorithm in the guest by writing the desired TCP parameters in the corresponding guest memory and registers.

Example. Assume we want to add a full modern congestion control stack to an old guest VM. Then the hypervisor could inject code in the guest as if it were malware with unlimited memory access.

Cons. Tenants may expect stronger VM isolation guarantees and not accept that the hypervisor writes into the VM memory, even for the purpose of improving performance. In cases where both the hypervisor and the guest VM control the networking stack, writing into memory may also slow down the VM because of the need for keeping consistency and ensuring synchronization between the write operations.

Read from guest memory. As above, the hypervisor may access the guest memory using guest introspection. However, by avoiding memory writes, it only monitors the memory and does not need synchronizations.

Example. This white-box solution makes the guest parameters transparent to the hypervisor translation layer. This could provide access to the congestion window without the need to maintain state to track it in the hypervisor, thereby enabling the receive-window technique (see below) .

Cons. Again, tenants may not accept that the hypervisor get a sneak peek inside their VMs. Also, when the hypervisor accesses the guest memory instead of keeping an internal state machine, it adds processing and communication delays.

Split connection. The split-connection approach breaks a TCP connection into several sub-connections, e.g. using a TCP proxy [18, 19]. It can acknowledge packets to the guest VM at any desired rate, then send them on the datacenter network using the desired target congestion control algorithm.

Example. This black-box solution functions as a pipe, and *can implement nearly any congestion control algorithm*. For instance, to implement MPTCP, the hypervisor can quickly prefetch packets from the guest VM at a high rate, then send them to the destination hypervisor using several paths.

Cons. In addition to needing many resources for buffering packets, this solution goes against TCP end-to-end semantics. For instance, a barrier-based application may believe that all its packets were ACKed, and advance to the next phase, while they were not actually received, causing errors in the application.

Buffer packets. The hypervisor translation layer can buffer in-flight packets, e.g. to be able to resend them without informing the guest [19, 24].

Example. In case of TCP incast, it is useful to reduce the retransmission timeout value RTO_{min}. Yet the guest OS may not make it available, or may not have a sufficiently low granularity. A simple alternative is for the hypervisor to retransmit from the desired sequence number without waiting for the corresponding timeout at the guest.

Cons. The hypervisor needs to manage packet buffers. Generally, it may also increase latency when packets are not sent immediately.

[2] All of our Linux code and experimental settings are publicly available on Github [20].

Buffer ACKs. The hypervisor can similarly buffer received ACKs [25–27] (while splitting and not buffering any piggy-backed data).
Example. The hypervisor can pace ACKs to make TCP less bursty.
Cons. The hypervisor needs to manage ACK buffers. It may also increase latency when ACKs are delayed.
Duplicate ACKs. The hypervisor can duplicate and resend the last sent ACK to force the guest to halve its congestion window.
Example. In case of TCP incast, the hypervisor can force a fast retransmit by sending three duplicate ACKs.
Cons. Beyond the need to keep the last ACK, this technique may also violate TCP semantics[3]. For instance, sending three ACKs on the last outstanding packet means that three additional packets have been received, which cannot happen.
Throttle the receive window. The hypervisor can decrease the receive window [25, 27, 29] to force the guest to have fewer outstanding packets, since the number of packets in flight is upper-bounded by the minimum of the congestion and the receive windows. Therefore, the advertised receive window could follow the target congestion window to make the guest adapt to this target.
Example. The hypervisor can implement ECN or DCTCP. Specifically, upon explicit congestion notification, the hypervisor translation layer decreases the receive window that it sends to the guest, without forwarding the explicit congestion notification itself (see experiments in Section 3).
Cons. This technique can make the congestion window meaningless, since it relies on the receive window to bound the number of in-flight packets. Also, the receive window cannot be decreased to less than the current number of packets on flight, because it would conflict with the TCP buffer management policy, and therefore the hypervisor needs to manage a gradual decrease while closely monitoring the connection state. Finally, while the technique helps make TCP less aggressive, it cannot make it more aggressive. For that, we would need to rely on a heavier technique, such as a split connection.
Modify the three-way handshake. The hypervisor can change the options that are negotiated when setting up the connection.
Example. The hypervisor can modify the negotiated MSS, or enable timestamps. This technique is also needed for several of the above techniques, e.g. to enable ECN support (see experiments in Section 3).
Cons. The technique can barely help for most practical benefits.

In all these techniques, we assumed for simplicity that the hypervisor knows the specific OS version and congestion control algorithm. In some cases, it may be straightforward to detect these automatically either by packet inspection, VM metadata, or other communication with the guest.

[3]Although it does not seem to directly go against RFC 5681 [28], which mentions the possibility of the replication of ACK segments by the network.

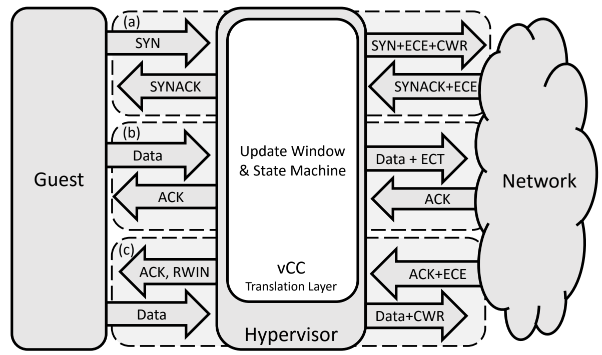


Figure 3: Interactions of the vCC translation layer in the hypervisor with TCP traffic. From top to bottom: (a) Connection negotiation, where the translation layer enables the ECE and CWR flag in the SYN packet to indicate ECN support, but hides the ECE field in the returning SYNACK; (b) normal data packets get their ECT bit set in the IP header and ACKs pass through the translation layer unchanged. The translation layer updates its internal state as data packets and ACKs pass through; (c) when an ACK with ECE bits is received, the translation layer erases the ECE bit and modifies the RWIN in the TCP header according to its window state, and the translation layer sets CWR on the next outgoing packet.

However, if the hypervisor either does not know or does not want to trust the information [30], it could equally use the heavier techniques (e.g. split connection) with a fixed-size receive buffer that is reflected in the receive window, and simply drop anything beyond the receive window.

In addition, note that when the guest already implements the target modern congestion control algorithm, vCC could either tunnel its traffic transparently, or still translate the traffic to make sure it obeys the exact same protocol implementation as any other translated vCC traffic.

Figure 3 illustrates how a combination of the three-way handshake modification and the receive-window throttling techniques can help provide virtual ECN benefits to non-ECN TCP traffic. The vCC translation layer in the hypervisor first uses the three-way handshake modification technique: in Figure 3(a), it modifies the TCP header fields of the sent packets to enable ECN support in the underlay. Next, while it only sets the ECT bit in the IP header of outgoing data packets and forwards incoming ACKs transparently (Figure 3(b)), it uses the receive-window throttling technique upon congestion. As shown in Figure 3(c), upon receiving an ECN congestion notification, it decreases the advertised receive window to force the overlay TCP guest sender to reduce its pace and behave *as if* it were ECN-aware. It also modifies the TCP header fields of the ACK packets to mask congestion notifications in the overlay. Note that we assume that the receiver is either ECN-enabled, or also has a vCC translation layer. In addition, in all these cases, we recompute the checksum by looking at the changed bytes only.

More formally, in that case, we are able to prove that we can exactly emulate ECN (where emulation is defined as in Equation (1) in the Introduction). We need two major assumptions. First, we assume that all the processing and communication times within the guest and hypervisor are negligible. Second, we build a TCP Reno state machine that follows RFC 5681 [28] and assume that the guest follows this state machine. We do so because our proof depends on this state machine, and we found that different OSes and even different OS versions can follow different state machines even for TCP Reno. We can then prove the following emulation theorem using the three-way handshake modification and the receive-window throttling techniques.

THEOREM 1. *The translation layer can exactly emulate an ECN-aware TCP Reno protocol given a TCP Reno guest.*

Likewise, assume that we want to emulate DCTCP based on the same TCP Reno protocol state machine. Then, using the same translation techniques, we can prove exact emulation again:

THEOREM 2. *The translation layer can exactly emulate DCTCP given a TCP Reno guest.*

The full formal proofs of these two theorems are in the Appendix. The full formal proofs of these two theorems are available online [20]. We gained two insights on full emulation when writing the proofs. First, the proofs strongly rely on the fact that given the same sequence of inputs (e.g., ACKs), ECN and DCTCP are actually surprisingly *less aggressive* than non-ECN TCP, in the sense that their resulting congestion windows will *never* be larger. For instance, if an explicit congestion notification arrives to a non-ECN TCP sender, it will ignore it, while an ECN-aware sender or DCTCP may reduce their congestion windows. The second insight that we gained is that it is much easier to prove full emulation when the timeouts are simultaneous in the state machines of the guest and of the hypervisor translation layer. This is why we assume negligible processing and communication times.

We believe that we could generalize these theorems to more complex translations by concatenating simpler translations: e.g., we could translate TCP Reno with ECN to DCTCP by concatenating (a) TCP Reno with ECN to TCP Reno without ECN (simply modify the three-way handshake); and (b) TCP Reno without ECN to DCTCP (as shown above).

3. EVALUATION: SOLVING ECN UNFAIRNESS

In the next sections, we show how a practical implementation of vCC can improve the performance and fairness of the network. We implement vCC in two distinct environments. The first implementation is realized at the edge of the Linux kernel TCP implementation. We demonstrate that vCC can help address unfairness between ECN and non-ECN traffic in this Linux environment. All experiments on this Linux vCC system are carried out with Mininet [31] on a virtual machine running Ubuntu 14.04 with Linux kernel version 3.19, except for the experiments with 1 Gbps links, which where performed using Mininet on a native Ubuntu 14.04 with Linux kernel version 3.13. We set TSO (TCP Segmentation Offloading) off in all Mininet experiments, because there is no real NIC within Mininet to implement TSO. The CPU and memory were never a bottleneck in all experiments.

The second environment is a proof-of-concept system in the VMWare ESXi hypervisor's vSwitch. We illustrate in Section 4 how our vCC solution can help provide bandwidth sharing in this hypervisor environment.

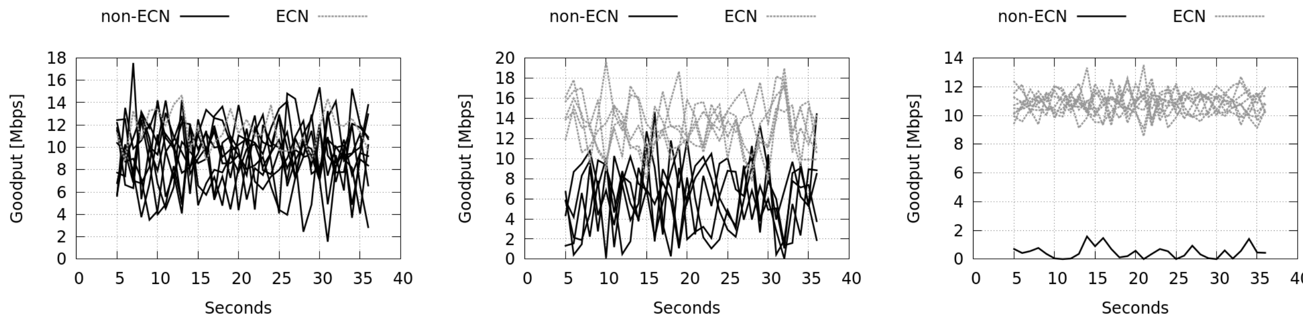
3.1 ECN Unfairness

ECN allows flows to react to impending congestion before any data has been lost due to congestion [1]. ECN can be a valuable tool to increase network performance, but it has not been widely supported in operating systems until recently [32]. Thus, legacy guests in a datacenter may not support ECN. Unfortunately, a lack of ECN support can cause such legacy systems to suffer. Figure 1 shows that, even across many dozens of runs (140 in this case), there is consistent starvation of non-ECN flows.

We first run an experiment to analyze the unfairness between ECN and non-ECN flows, for various numbers of ECN and non-ECN flows. 10 senders are connected through a switch to a single receiver. All links have a bandwidth of 100 Mbps and a delay of 0.25 ms, so the Round-Trip Time (RTT) is 1 ms. The RED parameters in these experiments are detailed in the RED1 column of Table 1. We use TCP Reno as the congestion control algorithm in all our experiments. We measure the goodput of long-lived TCP flows, using iPerf as the traffic source and TShark for capturing packets and measuring statistics. Each datapoint represents the average goodput over a second for a single flow.

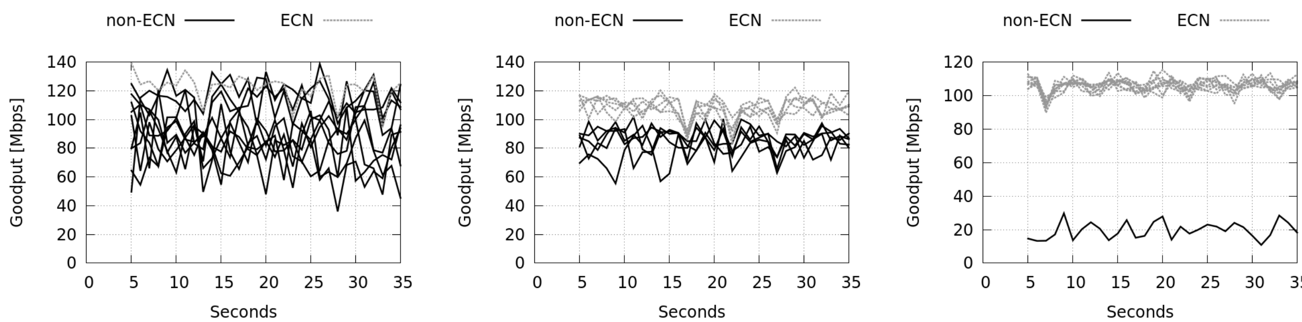
Figure 4 demonstrates the unfairness between ECN and non-ECN flows by plotting the time-series of their goodput. It shows that while the ECN flows fairly share the bottleneck link among themselves, the non-ECN flows can become significantly starved. The unfairness grows as ECN becomes more widespread and the ratio of ECN flows to non-ECN flows increases. This unfairness points out a *curse of legacy*: as applications increasingly adopt ECN, the holdout legacy applications become increasingly starved. Limited unfairness between ECN and non-ECN TCP flows was known given equal numbers of flows in each group [33]. However, the large impact of a plurality of newer ECN guests on a few non-ECN legacy guests appears to be new. To address this issue, it is possible to design alternative switch marking schemes that would favor legacy applications. However, ensuring fairness with legacy applications appears quite challenging.

We have also repeated this experiment with higher-rate links to emulate a datacenter environment more closely. Specifically, in this setting we use 1 Gbps links, a delay of 0.025 ms (i.e., RTT is 100 μs), an RTO_{min} of 20 ms (instead of the default 200 ms) and RED parameter set RED1 from



(a) 9 non-ECN vs. 1 ECN flows (b) 5 non-ECN vs. 5 ECN flows (c) 1 non-ECN vs. 9 ECN flows

Figure 4: Unfairness between ECN and non-ECN flows for several mixes of numbers of flow types, given a constant total number of 10 flows going through a shared 100 Mbps bottleneck link. As the ratio of ECN to non-ECN flows increases, the non-ECN flows suffer from increasing starvation and can send fewer and fewer packets.



(a) 9 non-ECN vs. 1 ECN flows (b) 5 non-ECN vs. 5 ECN flows (c) 1 non-ECN vs. 9 ECN flows

Figure 5: Repeated unfairness test between ECN and non-ECN flows with a 1 Gbps bottleneck link.

Table 1: RED Parameters used in the experiments.

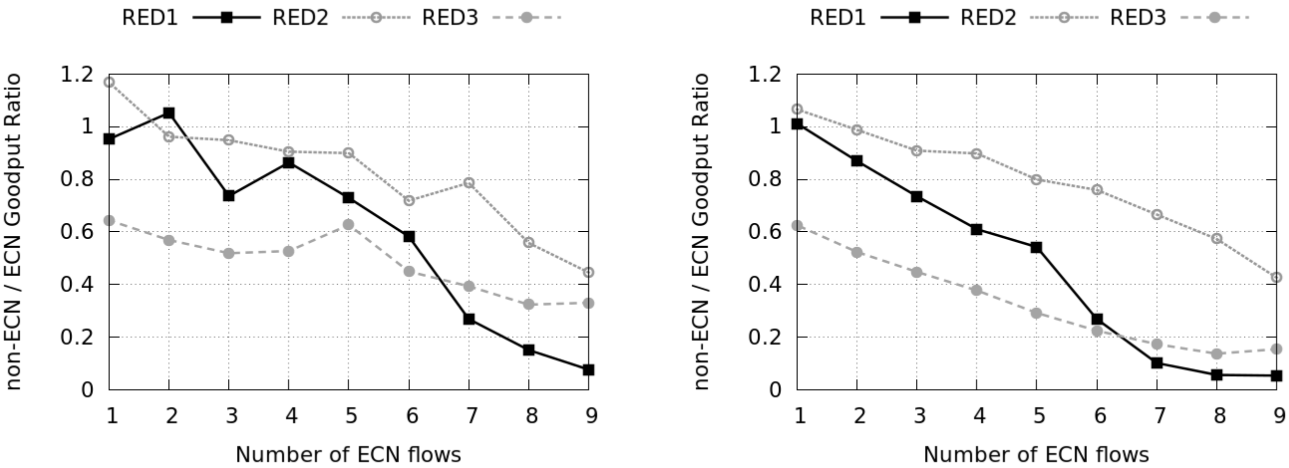
Parameter	Value		
	RED1	RED2	RED3
RED_{min}	90000	30000	30000
RED_{max}	90001	90000	90000
RED_{limit}	1M	400K	400K
RED_{burst}	61	55	55
RED_{prob}	1.0	0.02	1.0

Table 1. The results are presented in Figure 5. Note that the same trend is evident, although the unfairness appears even more severe than in Figure 4.

We next analyze this ECN unfairness for different ratios of ECN to non-ECN flow numbers using 10 Mbps and 100 Mbps links, and for various RED parameter sets (Table 1). The RED1 parameter set emulates a hard threshold AQM, where packets are dropped for non-ECN flows once the queue occupancy exceeds a certain threshold (RED_{min}), in a similar way to the AQM described for DCTCP [2]. The RED_{burst} parameter is set to the minimum allowed value in tc-red for RED1 parameters. RED2 was chosen as a default setting for RED, taken from tc-red's manpage example. RED3 is a slight modification of RED2 (modified RED_{prob}), to test a more aggressive marking/dropping policy.

Figure 6 plots the ratio of the average goodput of a non-ECN flow by the average goodput of an ECN flow, i.e. a measure of this unfairness, as a function of the number of ECN flows, given a total of 10 flows. It illustrates how for all tested parameter sets, introducing even a small number of ECN flows into the mix violates fairness between ECN and non-ECN flows. Even more, as there is only one non-ECN flow left out of the 10 flows, its goodput is always below 45% of the goodput of the ECN flows.

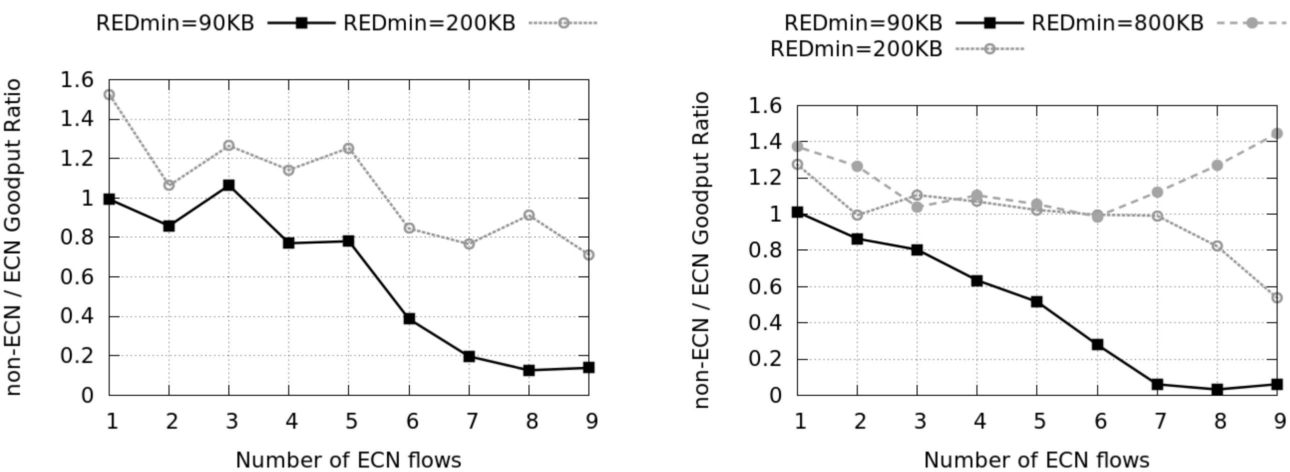
Figure 7 explores the effect on fairness when modifying the RED_{min} parameter in the RED1 parameter set. We set $RED_{max} = RED_{min} + 1$, and set RED_{burst} to the minimum allowed by tc-red. The figure depicts the goodput ratio for different values of RED_{min}. In general we see that the unfairness worsens as the proportion of ECN flows in the mix increases and as the RED_{min} decreases. The non-ECN flows suffer great goodput loss unless RED_{min} is set to a particularly large value ($64 \times \text{BDP} = 800$ KB). This is particularly worrying in datacenters, where RED_{min} tends to be smaller in order to keep low queue sizes and low delays.



(a) 10 Mbps links

(b) 100 Mbps links

Figure 6: Unfairness between ECN and non-ECN flows for several mixes of number of flow types and RED parameter sets, given a constant number of 10 flows. In all parameter sets, the unfairness becomes larger when there are fewer remaining non-ECN legacy flows.



(a) 10 Mbps links

(b) 100 Mbps links

Figure 7: Average goodput ratio with varying values of RED_{min}, given 10 senders. Realistic values of RED_{min} for datacenters lead to high ECN unfairness.

What is causing this unfairness? Figure 8 presents a 100 ms binned histogram of the time between consecutive acknowledgments sent by the receiver to a non-ECN and to an ECN flow, where the non-ECN flow is competing with 9 ECN flows on a 100 Mbps link using RED 1 parameters. The non-ECN flow suffers from repeated retransmission timeouts, as seen by the 200ms and the 600ms latencies. We found two dominant factors for these repeated timeouts:

Queue length averaging: Consider a state in which the *average* queue length measured by the switch grows beyond RED_{max}. In this scenario, the average queue length may remain above RED_{max} for a few RTTs due to the moving exponential averaging of the queue length. Meanwhile, every incoming packet of the non-ECN flow is discarded, causing the sender to time out waiting for ACKs on the dropped packets. Note that in this scenario fast retransmit is often not sufficient to save the sender's window, because the fast-retransmitted packets are dropped as well. After such a timeout, the non-ECN sender returns to slow-start, which further decreases its ability to recover due to the small number of duplicate ACKs at its disposal in a case of additional drops. In contrast, the packets of an ECN-capable sender are marked and not dropped. Upon receipt of an ACK-marked ECE, the sender halves its window and continues in congestion avoidance, without ever losing a packet or experiencing a timeout.

ECN to non-ECN flows ratio: Why does the unfairness

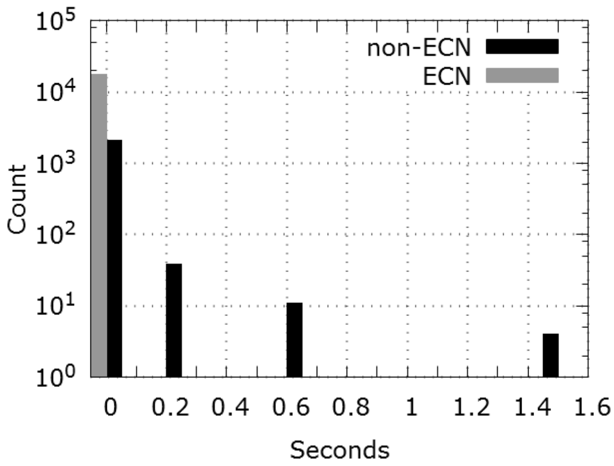


Figure 8: Histogram of time between consecutive acknowledgments sent by the receiver, divided into 100 ms bins, given a single non-ECN flow competing with 9 ECN flows on a 100 Mbps link.

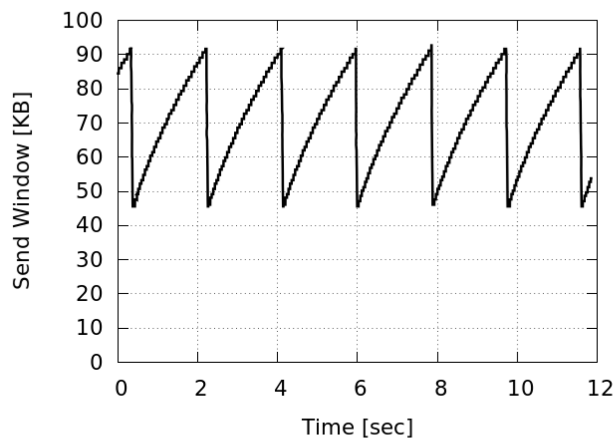


Figure 9: Send window time series for a virtual-ECN flow.

to non-ECN flows become more severe as the proportion of ECN-capable flows increases? Consider an equal number of ECN and non-ECN flows that share a bottleneck link, and a state in which the average queue length just increased beyond the marking threshold. Then, roughly half of the packets arriving at the queue during the following RTT will be discarded (i.e., those from non-ECN flows) and half will be enqueued (i.e., those from ECN flows). However, if we consider the same scenario with, say, 90% ECN-capable flows, then roughly 90% of the packets arriving at the queue will be enqueued. That is, more ECN flows in the mix cause the average queue length to be higher, providing even less room for the non-ECN flows to recover from their losses.

3.2 Receive Window Throttling

In order to address this fairness problem, we propose using the vCC translation layer to provide ECN capabilities to the non-ECN flows. We transform non-ECN flows from a guest to virtual-ECN flows that take advantage of ECN, using receive window throttling in the vCC translation layer. To demonstrate this, we configure one sender to send traffic through a switch to a receiver. The switch is configured with the RED1 parameter set from Table 1. The switch-sender link has a bandwidth of 12 Mbps, while the switch-receiver link has a bandwidth of 10 Mbps. Each link has a simulated link delay of 250 microseconds (i.e., RTT = 1 ms). The system is given 25 seconds to stabilize before data is collected for 12 seconds. The sender uses either native TCP-ECN, native TCP without ECN or virtual-ECN provided by the vCC translation layer (wherein the ECE bits are hidden from the guest to simulate ECN-ignorance). As previously detailed, Figure 3 diagrams the behavior of the translation layer for ECN support.

Figure 9 depicts the send window for the vCC experiment as reported by the tcp_probe kernel module. We can observe the familiar sawtooth pattern that would otherwise be seen in the congestion window. In our Linux implementation, when the receive window was the limiting window, the congestion window stayed larger than the receive window for the entire time, rendering the congestion window meaningless. Thus, modulating the receive window modulates the send window of the guest directly, and the resulting traffic flows are very similar. We have therefore created a virtual-ECN flow.

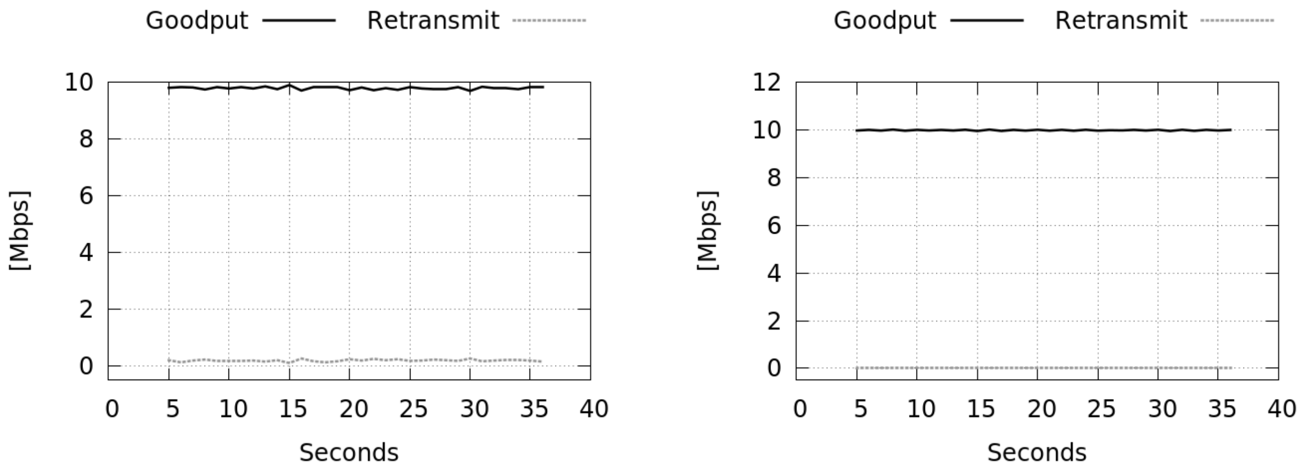
To demonstrate that indeed we get the ECN benefit of reduced retransmissions when using virtual-ECN, we repeat the experiment with 10 senders connected to a single receiver through a single switch, and compare the goodput and retransmission throughput obtained when using non-ECN flows to the same setup when virtual-ECN is used on all the flows.

Figure 10(a) illustrates that when using only non-ECN flows, some 2.3% of the link capacity is wasted on retransmissions due to packets dropped in the congested queue at the port connecting the switch to the receiver. However, as shown in Figure 10(b), once virtual-ECN is activated, the lost capacity is regained as virtual-ECN can react to congestion without dropping packets and retransmitting them.

3.3 Restoring Fairness with virtual-ECN

vCC offers the ability to transform a non-ECN flow into a virtual-ECN flow. We now evaluate whether this is sufficient to improve fairness beyond what we saw in Section 3.1.

Figure 11 shows the goodput achieved for 10 flows in the same setup as above, except that the non-ECN flow has been replaced with virtual-ECN. The resulting goodput of the flow from the ECN-incapable guest, seen in Figure 11 is now similar to that of its ECN-capable peers, with goodput 103.8% of the average goodput of the ECN-capable flows. This is a dramatic improvement over Figure 12, which shows the exact same experimental setup with a non-ECN flow instead of the virtual-ECN flow.



(a) 10 non-ECN flows

(b) 10 virtual-ECN flows

Figure 10: Total goodput and retransmission throughput for 10 concurrent non-ECN flows sharing a 10 Mbps link compared to the same experiment with virtual-ECN.

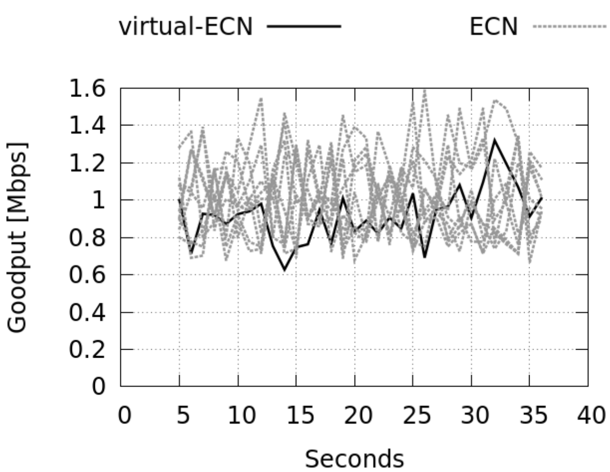


Figure 11: 9 ECN flows coexisting with one virtual-ECN flow provided by the vCC translation layer. The virtual-ECN flow has a goodput of 103.8% of the average goodput of the ECN flows. The fairness index is .994.

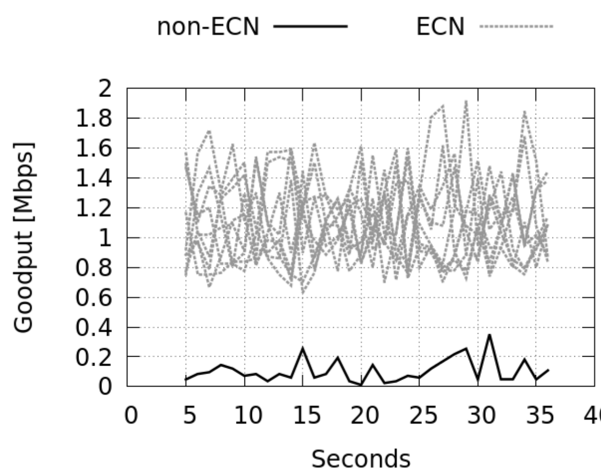


Figure 12: 9 ECN flows and one non-ECN flow. The non-ECN flow goodput is only 14.23% of the average goodput of the ECN flows. The fairness index is 0.921.

To summarize, the translation layer uses receive window throttling to cause the guest that does not support ECN to mimic its ECN-using peers, significantly improving its own goodput and the fairness of the network.

4. EVALUATION: HYPERVISOR BANDWIDTH SHARING

In this section, we describe a proof-of-concept vCC translation layer, which we implement on the VMware vSphere ESXi 6.0 hypervisor. We later illustrate how it can be used to provide bandwidth sharing.

The vCC translation layer is implemented as a filter called DVFilter [34] in the hypervisor's vSwitch. All per-flow states necessary for translation are stored in the hypervisor's own memory. The translation layer monitors flows passing through the switch, and inspects the headers in order to maintain correct state information about the flow (e.g., the current srtt, or the number of packets in flight). When the vCC translation layer determines it should modify headers, it changes the packet headers, recomputes the checksum, and allows the packet to pass through the filter. In particular, in this section, We demonstrate how we implemented receive-window throttling in this vCC layer.

Consider a multi-tenant datacenter. Each virtual machine may be the source of many TCP flows. However, not all of these flows should necessarily be treated the same for optimal performance – for example, some may be short but time-sensitive, while others are long but elastic. Thus, it may be useful to limit the rate at which certain applications are able to send. More generally, the ability to enforce tenant-based dynamic bandwidth allocations down to the granularity of applications is important to meet performance and SLA targets. WAN traffic shaping using a local Linux bandwidth enforcer is a promising approach [35]. This requires a uniform OS installation that does not generally allow multi-tenant hosting. Bandwidth limiting is available at guest granularity in some modern hypervisors (such as Microsoft's Hyper-V and VMware's ESXi), but per-application throttling is generally not. Moreover, to throttle bandwidth, these techniques can rely on either dropping packets or building large queues, which can have a detrimental effect on flow performance and latency.

Here we show another application of the receive-window throttling abilities of vCC. By controlling the end-to-end number of in-flight packets, vCC provides a fine-grained, datacenter-wide coordination of bandwidth allocation. A tenant or port or packet signature is detected in the hypervisor, and the flow is modified to restrict the bandwidth used by this particular set of traffic. In addition, the bandwidth limit can be changed dynamically, depending on signals from the network or from the guest.

Our hypervisor implementation provides a proof-of-concept for dynamic application-graunlarity bandwidth throttling. In this experiment, the vCC-enabled hypervisor is nested on another ESXi running on a Dell Poweredge T610 server, with 12 GB of RAM and 2 Intel Xeon processors at 2.4 GHz. Two guest VMs (Linux Centos 6.4) are hosted on top of the hypervisor, with the vCC translation layer installed in its vSwitch. They communicate through that hypervisor's vSwitch. One guest runs an iPerf server on 5 TCP ports. We divide flows into "preferred" and "unpreferred" flows. The preference can be seen as reflecting either time-sensitive or higher-paying tenants for instance. Three ports are given to unpreferred flows, and two to preferred flows. The total amount of window space – that is, the sum of the RWINs of all active flows – remains constant at all times. The translation layer is configured to evenly divide the available window space among unpreferred flows in the absence of preferred ones. When it detects active in-flight preferred flows, the translation layer dynamically changes the window space allocation to proportionally assign more window space to preferred flows (3 times as much per preferred flow as per unpreferred flow), and divides the remainder among the unpreferred flows evenly.

Figure 13 illustrates a time series of this experiment. It shows that after the introduction of the preferred flows, the throughput of unpreferred flows drops due to receive-window throttling, thus providing the preferred flows a larger share of the bandwidth. The total throughput before and after the introduction of preferred flows remains relatively constant.

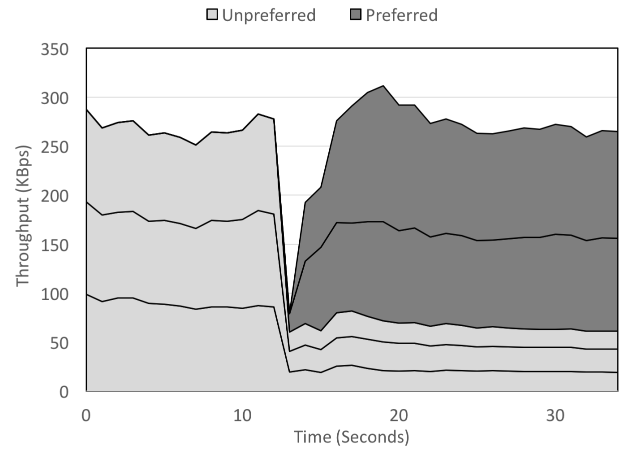


Figure 13: Stacked throughputs for three unpreferred and two preferred flows. The flows are receive window throttled by the ESXi vCC layer. The sum of the windows of all the live flows is kept constant throughout the experiment, but the vCC throttles unpreferred flows once preferred flows start in order to give the preferred flows greater bandwidth. The vCC layer uses the port number to differentiate between flows and preferences.

5. IMPLEMENTATION DISCUSSION

In this section, we discuss the architectural issues that a vCC implementation needs to address in the hypervisor.

Architectural complexity. Many hypervisor switches support an architecture where an independent module can inspect, modify, and re-inject the packet (e.g., Microsoft's Hyper-V Extensible Switch extension [36], VMware's ESXi DVFilter [34], and so on). This architecture is typically used by firewalls and security modules. For instance, a firewall implementation may allow over 1 million active connections with a per-connection state of under 1KB, given a total memory size of about 1GB.

A vCC implementation can leverage this architecture in order to modify packets, as we illustrate in our ESXi implementation. We would expect similar numbers given our simplest techniques without buffering. For instance, our Linux vCC implementation stores only 37 bytes per flow. This leaves room for a more complex implementation, given a per-connection footprint budget of well under 1KB. In addition, in most of the techniques mentioned in Section 2, the main CPU load consists of keeping track of the per-connection states of the guest congestion control algorithm.

Hypervisor delay. Processing delays in the hypervisor can increase the latency, and therefore the flow completion time, as well as affect the RTT estimation in the guest TCP algorithm. This effect would be more pronounced when the load of the hypervisor CPU is sufficiently high to cause context switches. In that case, the delay would be in the order of context switching delays, i.e. several μs.

Hypervisor bypass. High-performance virtualized workloads can benefit from bypassing the hypervisor and directly accessing the network interface card (NIC), using technologies such as SR-IOV [37–40]. vCC would not work in such architectures. However, hypervisor bypass is typically used in high-end devices with the newest OSes. Such OSes often already implement the latest congestion control, if only to obtain the best available performance In addition, future NICs could also implement vCC, although (a) software updates would not be as easy as for hypervisors, and (b) NICs may not have access to the more intrusive techniques such as guest introspection. The same would be true if servers had FPGAs or middleboxes.

TSO and LRO. TCP Segmentation Offload (TSO) and Large Receive Offload (LRO) are techniques for increasing the throughput of high-bandwidth network connections by reducing CPU overhead. TSO transfers large packet buffers to the NIC and lets it split them, while LRO does the reverse operation. The hypervisor needs to modify the vCC translation layer accordingly. Most techniques remain nearly unchanged. However, techniques that rely on packet buffering will need much larger buffers, and, if vCC wishes to retransmit TCP segments, it will also need to recreate individual segments.

Configuration. In the vCC architecture, the network administrator can use the hypervisor knowledge on the flows and decide to assign a different congestion control to different ports, IP addresses, applications, OSes, or tenants. For instance, long-term background flows may have a less aggressive congestion control than short urgent flows. Likewise, a proprietary congestion control can be restricted to intra-datacenter connections. Of course, a major disadvantage of modulating the congestion control is that several congestion control algorithms will coexist again in the datacenter.

UDP. This paper focuses on TCP, and therefore we would expect the hypervisor to let UDP traffic go through the translation layer in a transparent manner. Of course, we could generalize the same idea to UDP, and for instance make the translation layer translate UDP to a proprietary reliable UDP algorithm, at the cost of additional buffering and complexity.

Debugging. Adding packet-processing modules at the connection end-point necessarily makes debugging more complex when there are connection issues on a host. On the other hand, *normalizing* all the congestion control algorithms to the same reference algorithm, as enabled by vCC, can greatly simplify in-network debugging: where there were once many versions of different congestion control algorithms, there is now a single version of a single algorithm.

Inside connections. When two guest VMs on the same hypervisor communicate, they still go through the hypervisor, and therefore through the same translation layer. As a result, vCC is expected to work without changes.

Delay-based congestion control. We believe that vCC can translate to/from delay-based TCP algorithms like TCP Vegas and TIMELY [7]. To do so, it would need to use the more heavyweight techniques at its disposal, such as split connections and buffers.

Encryption: We believe that the vCC architecture can similarly be used to offer encryption services, such as TCPCrypt and IPSec [41, 42], to legacy unencrypted TCP flows. If the guest is already encrypting communications, vCC would need for instance to either read guest memory or intermediate during the initial private key negotiation to be able to operate.

Universal language. In order to directly translate between n congestion control algorithms, we would theoretically need to implement $O(n^2)$ translations. Instead, we could envision a universal atomic congestion control protocol enabling us to implement only $2n$ translation to/from this protocol.

6. RELATED WORK

Congestion control algorithms. Many congestion control algorithms and extensions have been suggested for datacenters, including ECN, DCTCP, D2TCP, MPTCP, TIMELY, DX, TCP-Bolt, Halfback and DCQCN [1–12]. The goal of this paper is to enable the hypervisor to implement such novel algorithms in the underlay physical network given legacy algorithms in the overlay guests.

Several research works have also suggested that the congestion control algorithm could adapt to the datacenter network conditions, with, for example, the use of Remy, Tao or PCC [13–15]. Our vCC architecture is ideally situated to implement such an adaptable congestion control algorithm in the underlay network.

TCP rate control. The ACK pacing and TCP rate control approaches [25–27, 29, 43] attempt to regulate the sending rate of the TCP flow. These papers presented the techniques of buffering TCP packets and ACKs, and reducing the receive window. Our vCC approach uses similar approaches. While these papers typically attempt to reach a fixed sending rate, the goal of our vCC solution is to translate between legacy congestion control algorithms and *any* modern congestion control algorithm.

Link-level retransmissions. In wireless communications, the Snoop protocol [19, 24] can buffer data packets at the base station, and then snoop on ACKs and retransmit lost packets on behalf of the sender, making sure to block duplicate ACKs from reaching the sender. This is similar to a link-level retransmission protocol, and can help address large loss rates at the last-mile link. Our vCC hypervisor can similarly snoop on ACKs and prevent duplicate ACKs from reaching the sender. However, it operates at the end-to-end level, and not at the link level.

Split connection. The split-connection approach breaks a TCP connection into several sub-connections, e.g. using a TCP proxy [18, 19, 44]. In contrast, the vCC architecture can keep the original TCP connection, and avoid breaking the end-to-end principle by not acknowledging packets unreceived by the receiver.

Shapers. Rate limiters and shapers in hypervisors and in NIC hardware [37, 45] enable better fairness between VMs, and can help provide per-VM max-min weighted fair shares [46–49]. They are complimentary to our vCC solution.

7. CONCLUSION

Our goal was to make it easy to deploy new and improved congestion control algorithms into multitenant data centers, without having to worry about TCP-friendliness with non-participating virtual machines. This paper presented our vCC solution, which enables the data center owner to introduce a new congestion control algorithm in the hypervisors. Internally, the hypervisors translate between the new congestion control algorithm and the old legacy congestion control, allowing legacy applications to enjoy the benefits of the new algorithm. Using the example of ECN traffic, we showed how this vCC solution can have an impact on fairness among tenants.

In the longer term, our goal is for the hypervisor translation layer to provide hooks that would simplify the coding of new congestion control algorithms, similarly to the existing hooks in the current Linux TCP stack implementations. These hooks would significantly reduce the deployment time of novel congestion control algorithms in large-scale datacenters.

8. ACKNOWLEDGMENTS

We would like to thank the many people whom we consulted for this paper, including Guido Appenzeller, David Tennenhouse, Dahlia Malkhi, Mukesh Hira, Ben Basler, Jim Stabile, Azeem Feroz, Boon Ang, Pushkar Putil, Naga Katta, Subrahmanyam Manuguri, Steven Hand, Marco Canini, as well as our shepherd, Vishal Mishra.

9. REFERENCES

[1] Sally Floyd. TCP and explicit congestion notification. *ACM SIGCOMM*, 1994.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). *ACM SIGCOMM*, 2011.

[3] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. *ACM SIGMETRICS*, 2011.

[4] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). *ACM SIGCOMM*, 2012.

[5] Glenn Judd. Attaining the promise and avoiding the pitfalls of TCP in the datacenter. *USENIX NSDI*, 2015.

[6] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM*, 2011.

[7] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, et al. TIMELY: RTT-based congestion control for the datacenter. *ACM SIGCOMM*, 2015.

[8] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. *USENIX ATC*, 2015.

[9] Brent Stephens, Alan L Cox, Anubhav Singla, Jenny Carter, Colin Dixon, and Wes Felter. Practical DCB for improved data center networks. *IEEE Infocom*, 2014.

[10] Qingxi Li, Mo Dong, and Brighten Godfrey. Halfback: Running short flows quickly and safely. *ACM CoNEXT*, 2015.

[11] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. *ACM SIGCOMM*, 2015.

[12] Prasanthi Sreekumari and Jae-il Jung. Transport protocols for data center networks: a survey of issues, solutions and challenges. *Photonic Network Communications*, pages 1–17, 2015.

[13] Keith Winstein and Hari Balakrishnan. TCP ex machina: Computer-generated congestion control. *ACM SIGCOMM*, 2013.

[14] Anirudh Sivaraman, Keith Winstein, Pratiksha Thaker, and Hari Balakrishnan. An experimental study of the learnability of congestion control. *ACM SIGCOMM*, 2014.

[15] Mo Dong, Qingxi Li, Doron Zarchy, Brighten Godfrey, and Michael Schapira. Rethinking congestion control architecture: Performance-oriented congestion control. *ACM SIGCOMM*, 2014.

[16] Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, Avinoam Kolodny, and Isaac Keslassy. Links as a Service (LaaS): Guaranteed tenant isolation in the shared cloud. *ACM/IEEE ANCS*, 2016.

[17] Mirja Kuhlewind, David P Wagner, Juan Manuel Reyes Espinosa, and Bob Briscoe. Using Data Center TCP (DCTCP) in the Internet. *IEEE Globecom Workshops*, 2014.

[18] Michele Luglio, M Yahya Sanadidi, Mario Gerla, and James Stepanek. On-board satellite split TCP proxy. *IEEE J. Select. Areas Commun.*, 22(2):362–370, 2004.

[19] Xiang Chen, Hongqiang Zhai, Jianfeng Wang, and Yuguang Fang. A survey on improving TCP performance over wireless networks. *Resource management in wireless networking*, 2005.

[20] vCC project. http://webee.technion.ac.il/~isaac/vcc/.

[21] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. *NDSS*, 2003.

[22] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. *IEEE Symposium on Security and Privacy*, 2008.

[23] VMsafe. https://www.vmware.com/company/news/releases/vmsafe_vmworld.

[24] Hari Balakrishnan, Srinivasan Seshan, and Randy H Katz. Improving reliable transport and handoff performance in cellular wireless networks. *Wireless Networks*, 1(4):469–481, 1995.

[25] Shrikrishna Karandikar, Shivkumar Kalyanaraman, Prasad Bagal, and Bob Packer. TCP rate control. *ACM SIGCOMM*, 2000.

[26] James Aweya, Michel Ouellette, and Delfin Y Montuno. A self-regulating TCP acknowledgment (ACK) pacing scheme. *International Journal of Network Management*, 12(3):145–163, 2002.

[27] Huan-Yun Wei, Shih-Chiang Tsao, and Ying-Dar Lin. Assessing and improving TCP rate shaping over edge gateways. *IEEE Trans. Comput.*, 53(3):259–275, 2004.

[28] RFC 5681. https://tools.ietf.org/html/rfc5681.

[29] Lampros Kalampoukas, Anujan Varma, and KK Ramakrishnan. Explicit window adaptation: A method to enhance TCP performance. *IEEE Infocom*, 1998.

[30] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM*, 1999.

[31] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. *ACM CoNEXT*, 2012.

[32] Mirja Kühlewind, Sebastian Neuner, and Brian Trammell. On the state of ECN and TCP options on the Internet. *International Conference on Passive and Active Measurement*, 2013.

[33] Yin Zhang and Lili Qiu. Understanding the end-to-end performance impact of RED in a heterogeneous environment. Technical report, Cornell, 2000.

[34] VMware vSphere DVFilter. https://pubs.vmware.com/vsphere-60/index.jsp?topic=%2Fcom.vmware.vsphere.networking.doc%2FGUID-639ED633-A89A-470F-8056-5BB71E8C3F8F.html.

[35] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Bjorn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM*, 2015.

[36] Microsoft Hyper-V Extensible Switch. https://msdn.microsoft.com/en-us/library/windows/hardware/jj673961%28v=vs.85%29.aspx.

[37] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. Senic: Scalable NIC for end-host rate limiting. *USENIX NSDI*, 2014.

[38] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. FasTrak: enabling express lanes in multi-tenant data centers. *ACM CoNEXT*, 2013.

[39] Jeffrey C Mogul, Jayaram Mudigonda, Jose Renato Santos, and Yoshio Turner. The NIC is the hypervisor: bare-metal guests in IaaS clouds. 2013.

[40] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, UC Berkeley, 2015.

[41] Stephen Kent and Karen Seo. Security Architecture for the Internet Protocol. RFC 4301, RFC Editor, December 2005.

[42] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazières, and Dan Boneh. The case for ubiquitous transport-level encryption. *USENIX Security*, 2010.

[43] Haiqing Jiang, Yaogong Wang, Kyunghan Lee, and Injong Rhee. Tackling bufferbloat in 3G/4G networks. *IMC*, 2012.

[44] F5 Networks. Optimize WAN and LAN application performance with TCP Express. 2007.

[45] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing traffic shapers for practical resource allocation. *USENIX HotCloud*, 2013.

[46] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. *ACM SIGCOMM*, 2011.

[47] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. *ACM CoNEXT*, 2010.

[48] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. *USENIX NSDI*, 2011.

[49] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Changhoon Kim, and Albert EyeQ: Practical network performance isolation at the edge. *USENIX NSDI*, 2013.

APPENDIX

A. NOTATIONS AND DEFINITIONS

We begin by making the following notations. W_c is the congestion window of a TCP sender. W_r is the latest advertised receive window of a TCP receiver that was received by a TCP sender. *fs* is the flight-size of the TCP sender, namely, the number of sent but not yet cumulatively acknowledged packets. W_{eff} is the effective send window of the TCP sender and is defined as follows:

$$W_{\mathit{eff}} = \max\{\min\{W_c, W_r\} - \mathit{fs}, 0\} \qquad (2)$$

Namely, the effective send window is the number of *additional* packets that the sender is allowed to send. Let *ssthresh* be the slow-start threshold of a TCP sender and MSS be the maximum segment size agreed between the sender and the receiver during the three-way handshake.

We denote the TCP guest by *gst*, the hypervisor as *vcc* and the TCP receiver as *rcv*. We use a superscript to denote a corresponding congestion parameter owner. For example, W_c^{gst} is the congestion window of the guest and fs^{vcc} is the flight-size of the hypervisor.

We denote by n the n'th (input) event experienced by the TCP congestion control state machine owner. Specifically, an event can be (a) the arrival of an ACK, (b) resuming transmission after an idle period, or (c) a timeout. Let f be any TCP congestion variable, then we denote by $f(n)$ the size of f right after the n'th event. For example, $W_c^{gst}(n)$ is the congestion window of the guest and $fs^{vcc}(n)$ is the flight-size of the hypervisor right after event number n.

Next, we present the detailed actions that are taken in each state for each event.

B. TCP-RENO CONGESTION CONTROL

For our proofs, we use a conceptual TCP-RENO congestion control state machine that corresponds to RFC 5681 [28]. Specifically, we consider the TCP-RENO state diagram in Figure 14. Then, the congestion parameters obey the following rules:

Slow Start: Upon a new acknowledged packet:

$$W_c(n) = W_c(n-1) + MSS \tag{3}$$

Congestion Avoidance: Upon a new acknowledged packet:

$$W_c(n) = W_c(n-1) + MSS \cdot \frac{MSS}{W_c(n-1)} \tag{4}$$

Fast Recovery: Upon any acknowledged packet:

$$W_c(n) = W_c(n-1) + MSS \tag{5}$$

Upon entering Fast Recovery:

$$\textit{ssthresh}(n) = \max\left\{\left\lfloor \frac{1}{2} \cdot \textit{fs}(n-1) \right\rfloor, 2 \cdot MSS\right\} \tag{6}$$

$$W_c(n) = \textit{ssthresh}(n) + 3 \cdot MSS \tag{7}$$

Upon leaving Fast Recovery:

$$W_c(n) = \textit{ssthresh}(n) \tag{8}$$

Upon Timeout:

$$W_c(n) = MSS \tag{9}$$

$$\textit{ssthresh}(n) = \max\left\{\left\lfloor \frac{1}{2} \cdot \textit{fs}(n-1) \right\rfloor, 2 \cdot MSS\right\} \tag{10}$$

Upon Idle period: (i.e., when the connection is idle for more than an RTO). In this case RFC 5681 recommends to resume the transmission from slow start with the following congestion restart window:

$$W_c(n) = \min\{W_{init}, W_c(n-1)\} \tag{11}$$

and

$$\textit{ssthresh}(n) = \textit{ssthresh}(n-1), \tag{12}$$

where W_{init} is the initial window size after the connection establishment.

So far we have described our considered TCP-RENO congestion control. We next define a TCP-RENO-based ECN support that corresponds to RFC 3168 and a TCP-RENO-based DCTCP that corresponds to [**?**].

ECN support: in order to support ECN, the additions to a TCP sender are:

- A received ACK that arrives with an ECN-Echo (ECE) flag set, is treated by the sender as if a packet was lost (i.e., Equations (6) and (7)). However, this ECE flag *does not affect the TCP sender state*. Specifically, the TCP sender does not enter Fast Recovery but maintains its current state.
- The sender will not decrease its congestion window and slow start threshold as a reaction to additional ECE flags that arrive within the same window of data as the previous one.
- The sender will not increase its congestion window as a response to an ACK with ECE flag.

DCTCP support: the sender maintains a running estimate of the fraction of its packets that are marked with a congestion flag. This estimate, which is denoted by α, is updated once for every window of data. Then, DCTCP uses α to reduce its window in response to congestion as follows:

$$W_{c,\,\textit{new window}} = W_{c,\,\textit{last old window}} \cdot \left(1 - \frac{\alpha}{2}\right) \tag{13}$$

where

$$0 \leq \alpha \leq 1. \tag{14}$$

Namely, the window is reduced, at most, by a factor of 2 approximately each RTT.

In addition, for both ECN and DCTCP, the arrival of congestion flags does not change the state of the congestion control state machine and does not change the reaction to other events experienced by the sender.

Next we continue by presenting the main idea that allows full emulation of ECN and DCTCP and the assumption that we make to be able to conduct a formal proof.

C. SYNCHRONIZATION AND INVARIANTS

We continue to show that we can provide an exact translation from the considered TCP-RENO to a TCP-RENO with ECN support and TCP-RENO-based DCTCP with a hypervisor using *only three-way handshake modification* and *receive-window throttling* techniques and under the following assumptions:

Assumption 1. All the processing and communication times within the guest and hypervisor are negligible.

Assumption 2. The guest follows the TCP-RENO congestion control that is considered in Section A.

Why do we need those assumptions? We observe that in order to provide exact emulation capabilities, we would like the hypervisor to behave according to the TCP we want to emulate (specifically, ECN or DCTCP) and the guest to provide the required packets to the hypervisor whenever they are needed (i.e., not to *starve* the hypervisor unless there are no packets to send). On the other hand, since we want to

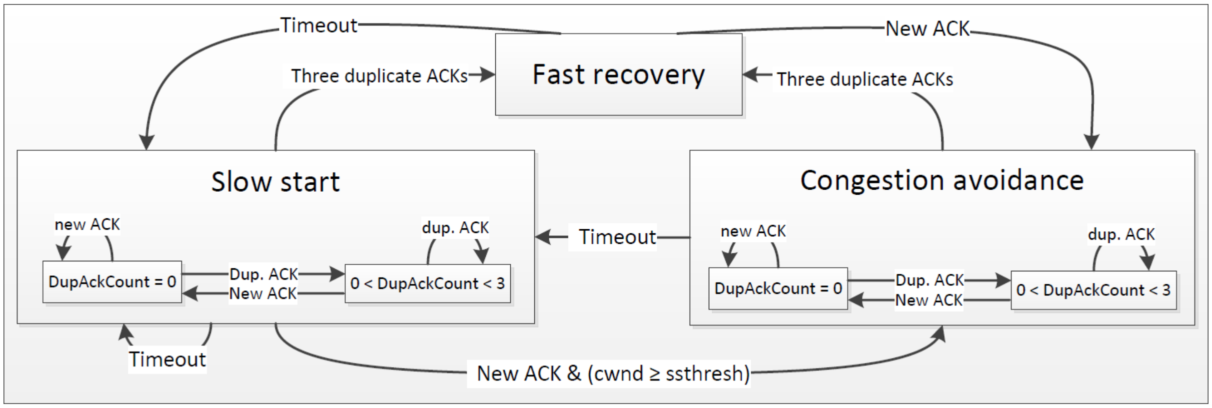


Figure 14: TCP-RENO congestion control state machine that corresponds to RFC 5681.

avoid buffering, we do not want the guest to provide packets to the hypervisor if the later cannot send those packets to the network according to its congestion control.

In addition, we observe that TCP congestion control algorithms are deterministic. Thus, if the TCP used by the hypervisor was identical to the one used by the guest and fed by the same input (i.e., the same received ACKs at the same time), then both will produce the same output (i.e., both will send the same packets at the same time). Specifically, we would obtain:

$$W_c^{gst} = W_c^{vcc}, \tag{15}$$

and thus achieve the desired result. However, we want to emulate a *different* congestion control algorithms. Our solution is to control the advertised receive window that is forwarded to the guest by the hypervisor and enforce,

$$W_{eff}^{gst} = W_{eff}^{vcc}. \tag{16}$$

This way, we achieve two guarantees. **(1)** The guest will forward packets to the hypervisor only when the latter is allowed to forward them according to its congestion control. Thus, no buffering at the hypervisor is needed (we shall refer this as the *in-flight invariant*). **(2)** The guest will not starve the hypervisor due to a *smaller* congestion window.

(1) To archieve this, we use the following assignment:

$$W_r^{gst} = fs^{vcc} + W_{eff}^{vcc} \tag{17}$$

This assignment never goes below the flight-size of the guest and additionally ensures that each packet that is sent by the guest can also be immediately sent by the hypervisor.

This assignment also reveals the need for Assumption 1. Specifically, we want to avoid a situation in which the following order of events will violate the in-flight invariant:

- at time t Equation (16) holds and in addition the congestion window of the guest is significantly bigger than its flight size due to temporary starvation by the application.
- at time $t + \varepsilon$ two events take place. (1) The hypervisor receives a congestion notification. (2) The application pushes to the guest additional data to send.

Namely, if the processing delay of the hypervisor and the guest is not negligible, then we might encounter a situation in which the hypervisor already decreased its congestion window but the guest sent new packets before receiving the new advertised window. Assumption 1 essentially states that all events (i.e., arrived ACKs and timeouts) are simultaneous for both the guest and the hypervisor with no possibility of interleaving events between the guest and the application. Thus, such order of events as described is not possible.

(2) This reveals the need for Assumption 2. Specifically, we will show that when the hypervisor (a) uses the *same exact* TCP-RENO congestion control as the guest (including RTT estimation technique), (b) uses Equation (17), and in addition (c) supports ECN or DCTCP, then we obtain:

$$W_c^{vcc} \leq W_c^{gst}. \tag{18}$$

Finally, combining Equations (17) and (18) will yield Equation (16). Namely, no buffering is needed and the hypervisor is never starved by the guest (unless both are starved by the application) - *an exact emulation*.

D. EXACT EMULATION

Next, we prove that indeed when the hypervisor supports ECN or DCTCP and uses Equation (17) within our considered model we obtain Equation (18). Then, using this result we will show that it is a sufficient condition for an exact emulation.

LEMMA 1. *Consider Assumptions 1 and 2. Assume that the hypervisor uses the same congestion control as the guest (including RTT estimation technique), uses the assignment in Equation* (17) *and additionally supports ECN as described in Section B. Then:*

$$W_c^{vcc}(n) \leq W_c^{gst}(n) \quad \forall n, \tag{19}$$

and

$$ssthresh^{vcc}(n) \leq ssthresh^{gst}(n) \quad \forall n. \tag{20}$$

PROOF. As mentioned above, when the hypervisor uses the assignment that is presented in Equation (17), then Assumption 1 ensures the following *in-flight invariant*: all the packets sent by the guest are immediately forwarded by the hypevisor. This important invariant ensures:

- Provided that both the guest and the hypervisor use the same RTT estimation technique and have the same RTO_{min} value, all timeout events are simultaneous in both (i.e., both send packets at the same time and receive ACKs at the same time: (i) their RTO timers are set and reset at the same time. (ii) Both have the same RTT estimation).
- Both enter and leave Fast Recovery simultaneously (i.e., both receive any third duplicate ACK and any New ACK simultaneously).

With these observations at hand, we prove the Lemma by induction on the number of events at the guest and hypervisor senders (i.e., n).

Base: $n = 0$**.** Since both the guest and the hypervisor use the same initial window size and initial slow start threshold it trivially holds that:

$$W_c^{vcc}(0) \leq W_c^{gst}(0), \tag{21}$$

and

$$\mathit{ssthresh}^{vcc}(0) \leq \mathit{ssthresh}^{gst}(0). \tag{22}$$

Induction hypothesis: Assume that:

$$W_c^{vcc}(n-1) \leq W_c^{gst}(n-1), \tag{23}$$

and

$$\mathit{ssthresh}^{vcc}(n-1) \leq \mathit{ssthresh}^{gst}(n-1). \tag{24}$$

Inductive step: We split the proof into five possible cases (a case for each event):

Case 1: Timeout. According to Equation (9) we immediately obtain the result for the congestion windows. Formally:

$$W_c^{vcc}(n) = W_c^{gst}(n) = MSS. \tag{25}$$

Regarding *ssthresh* , since the hypervisor uses Equation (17), we obtain the *in-flight invariant*. Specifically it holds that:

$$\mathit{fs}^{gst}(n-1) = \mathit{fs}^{vcc}(n-1). \tag{26}$$

Next, applying Equation (26) to Equation (10) yields the desired result. Formally:

$$\begin{aligned} &\mathit{ssthresh}(n)^{gst} = \\ &\max\left\{\left\lfloor \frac{1}{2}\cdot \mathit{fs}(n-1)^{gst}\right\rfloor, 2\cdot MSS\right\} = \\ &\max\left\{\left\lfloor \frac{1}{2}\cdot \mathit{fs}(n-1)^{vcc}\right\rfloor, 2\cdot MSS\right\} = \\ &\mathit{ssthresh}(n)^{vcc}. \end{aligned} \tag{27}$$

Case 2: New ACK. Fist we notice that the slow start threshold does not change for both. For the analysis of the congestion windows, we split into five possible configurations:

- Both in slow start. In this case, applying the induction hypothesis and Equation (3) yields the results.
- Both in congestion avoidance. In this case, applying the induction hypothesis and Equation (4) yields the result.
- The guest is in slow start and the hypervisor is in congestion avoidance. In this case, applying the induction hypothesis and Equations (3) and (4) yields the result.
- The hypervisor is in slow start and the guest is in congestion avoidance. In this case, according to the induction hypothesis (specifically, the *ssthresh* inequity) it must hold that $W_c^{gst} > W_c^{vcc}$ (namely, strict inequity). Thus, applying Equations (3) and (4) yields the result.
- Both in fast recovery. In this case, applying the induction hypothesis and Equation (8) yields the results.

Case 3: First or second Duplicate ACK. It must hold that both in fast recovery or both are not in fast recovery, thus the claim holds. Specifically, if both in fast recovery, both windows grow by one MSS and if not, both windows do not change. *ssthresh* does not change for both.

Case 4: Third Duplicate ACK. Then, both enter fast recovery. In this case, applying the in-flight invariant and Equations (7) and (6) yields the results.

Case 5: Resuming after Idle period. In this case, applying the induction hypothesis and Equations (11) and (12) yields the result.

What about cases in which an ECE flag arrives (with an ACK) to which the hypervisor reacts? The correctness for these cases follows immediately since such a flag is treated like a loss by the hypervisor without affecting its state. Thus, such an event can only further decrease the congestion window and the slow start threshold of the hypervisor and the inductive step holds.

Additionally, in some implementation of TCP-RENO congestion control, it might be the case that when the limiting window of the sender is the receive window, then even upon the arrival of a new ACK the congestion window should not be increased. Such implementation does not harm the correctness of these proofs. The reason is that we have only two possible options: (1) the receive window is the limiting window for both the guest and the hypervisor - in that case both congestion windows do not grow. (2) the receive window is the limiting window only for the guest - in that case the congestion window of the guest in *strictly* bigger than the congestion window of the hypervisor. □

With this lemma at hand, we continue to show that an exact emulation of ECN is possible.

THEOREM 1. *The translation layer can exactly emulate an ECN-aware TCP Reno protocol given a TCP Reno guest. Namely, consider Assumptions 1 and 2. Assume that the hypervisor uses the same congestion control as the guest (including RTT estimation technique), uses the assignment in Equation* (17) *and additionally supports ECN as described in Section B. Then, we have an exact ECN emulation.*

PROOF. To prove an exact emulation it is sufficient to show that Equation (16) holds for all times.

Let $W_{\mathit{eff}}^{\mathit{vcc}}(n) = k$. According to Equation (17) and the in-flight invariant we obtain:

$$W_r^{\mathit{gst}}(n) = \mathit{fs}^{\mathit{vcc}}(n) + k = \mathit{fs}^{\mathit{gst}}(n) + k. \tag{28}$$

Applying Equation (28) on Equation (2) yields:

$$\begin{aligned} &W_{\mathit{eff}}^{\mathit{gst}}(n) = \\ &\max\{\min\{W_c^{\mathit{gst}}(n), \mathit{fs}^{\mathit{gst}}(n) + k\} - \mathit{fs}^{\mathit{gst}}(n), 0\}. \end{aligned} \tag{29}$$

Additionally, according to Lemma 1 and Equation (17) it must hold that:

$$\begin{aligned} &W_c^{\mathit{gst}}(n) \geq W_c^{\mathit{vcc}}(n) \geq \\ &\mathit{fs}^{\mathit{vcc}}(n) + k = \mathit{fs}^{\mathit{gst}}(n) + k. \end{aligned} \tag{30}$$

Applying Equation (30) on Equation (29) yields:

$$W_r^{\mathit{gst}}(n) = k. \tag{31}$$

This concludes the proof. □

Next, we continue to prove that DCTCP exact emulation is possible, where again, we begin by the following Lemma.

LEMMA 2. *Consider Assumptions 1 and 2. Assume that the hypervisor uses the same congestion control as the guest (including RTT estimation technique), uses the assignment in Equation* (17) *and additionally supports DCTCP as described in Section B. Then:*

$$W_c^{\mathit{vcc}}(n) \leq W_c^{\mathit{gst}}(n) \quad \forall n, \tag{32}$$

and

$$\mathit{ssthresh}^{\mathit{vcc}}(n) \leq \mathit{ssthresh}^{\mathit{gst}}(n) \quad \forall n. \tag{33}$$

PROOF. DCTCP reaction to congestion flags is different than ECN as described in Section B, however, still can only further reduce the congestion parameters of the hypevisor without affecting its state. Thus, the proof is identical to the proof of Lemma 1. □

With this lemma at hand, again, we continue to show that an exact emulation of DCTCP is possible.

THEOREM 2. *The translation layer can exactly emulate DCTCP given a TCP Reno guest. Namely, consider Assumptions 1 and 2. Assume that the hypervisor uses the same congestion control as the guest (including RTT estimation technique), uses the assignment in Equation* (17) *and additionally supports DCTCP as described in Section B. Then, we have an exact DCTCP emulation.*

PROOF. Again, the proof is identical to the proof of Theorem 1 with the only exception of using Lemma 2 instead of Lemma 1. □

Interestingly, these proofs do not depend on the specific way to reduce the congestion variables of the hypervisor as a reaction to congestion flags. Thus, as long as the specific reaction does not trigger a change of state, a similar result can apply to arbitrary variations of the ECN and DCTCP congestion control algorithms. In addition, we observe that in fact, under such a construction in which the in-flight invariant is ensured, ECN and DCTCP support cause the congestion control to be *less aggressive*. In other words, the observed aggressiveness of ECN and DCTCP follows from the asymmetric behaviors of the network (i.e., AQM) and not from the algorithms themselves.