Virtualized Congestion Control (Extended Version)

Updated August 11, 2016

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 datacenters by reducing latency, overcoming incast, increasing throughput and improving fairness. Ideally, the operating system in every server and virtual machine is updated to support 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 datacenters, without having to worry about TCP-friendliness with non-participating virtual machines. This paper presents a solution we call *virtualized congestion control*. The datacenter 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.

CCS Concepts

•**Networks** → **Transport protocols;** *Network architectures; Programmable networks; Data center networks;*

Keywords

Virtualized congestion control; algorithmic virtualization; datacenters; hypervisors; ECN; DCTCP; TCP.

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 the network's behavior to best serve their large distributed applications.

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 [7]: their tenants implement their *own* congestion control algorithms. Yet, the multitenant datacenter owner must ensure that all the algorithms still play well together in the shared datacenter network, so as to meet

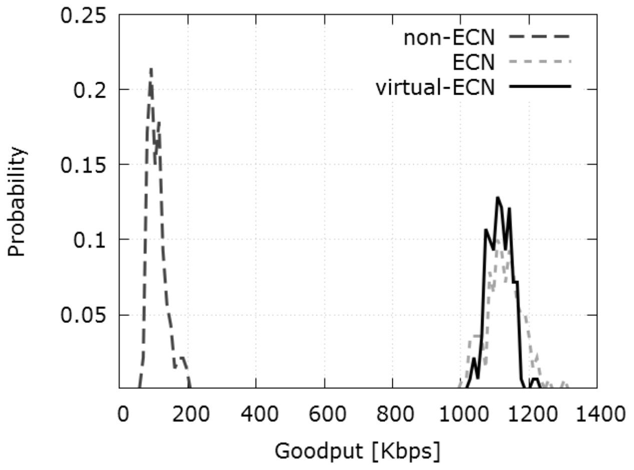


Figure 1: 10 flows share the same bottleneck link: an ECN-unaware flow (*non-ECN*), 8 ECN-enabled flows (*ECN*), and a non-ECN flow augmented by vCC translation (*virtual-ECN*). The figure plots the frequency, over many runs, of the average goodput of each flow. The non-ECN flow is starved, reaching only 10% of the ECN goodput on average. After translation to virtual-ECN, the average goodput is near identical to that of ECN.

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 [7, 16]), what is a multitenant datacenter owner to do?

A few approaches come to mind. For instance, the multitenant datacenter owner can strictly divide the bandwidth among the tenants, giving each a fixed allocation [17]. 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 example by implementing separate queues or applying different marking thresholds within the same queue [7, 16]. 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. 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? (as illustrated in Figure 2.) In other words, the guest congestion control algorithm is an *overlay*

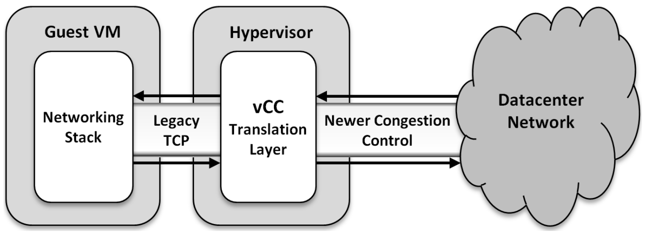


Figure 2: High-level illustration of vCC (virtualized congestion control). The vCC translation layer sits in the hypervisor, where it translates the guest's legacy congestion control to a target congestion control algorithm.

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 legacy application runs on a legacy guest VM operating system (OS) that uses an old TCP version (*e.g.,* an ECN-unaware TCP NewReno). 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 (*e.g.,* 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 this 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, in this case, 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 receive window to make the overlay TCP guest sender 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 existing algorithms in multitenant datacenters. 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 guest VMs. Finally, since each hypervisor can determine the application and tenant that generated each of the hypervisor 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 a concept that we denote *algorithmic virtualization*. While *resource virtualization* is about sharing a common resource and making each guest believe that it keeps using the 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.[2] 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 receive 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 underlay 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 overlay 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 the guest-VM legacy applications will continue to use their legacy TCP implementations, the hypervisor will translate this legacy TCP into a newer congestion control algorithm *under-the-hood*. As a result, the hypervisor 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 similar expected performance and therefore in increased fairness. 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 a complex implementation. Likewise, by breaking a TCP connection into several sub-connections, a TCP proxy-like [19,20] solution can implement nearly any congestion control algorithm, but may violate TCP end-to-end semantics by acknowledging packets that were not received by the destination receiver.

We also suggest more lightweight approaches that provide a more limited set of benefits. For example, if the hypervisor can update the receive window field in ACKs, then we show that it can provide ECN-like or DCTCP-like properties to an ECN-unaware TCP congestion control. In fact, in specific cases, we prove that it can exactly emulate either ECN or DCTCP.

Fairness in mixed-ECN environments.[3] In Section 3, we show that a minority of non-ECN legacy flows can get starved by a majority of ECN flows. This is in part because when a switch buffer becomes congested, packets from the ECN flows continue to enter the buffer for at least an RTT, keeping the buffer congested. As a result, the switch may drop long sequences of non-ECN packets, causing timeouts in non-ECN flows.

[1]The data presented here represents 140 runs of the experiment. Each run lasted 37 seconds; the first 5 and last 2 seconds were not included in the goodput 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. All links have a bandwidth of 10 Mbps and a delay of 250 μs, so the round-trip time (RTT) is 1 ms. ECN and non-ECN flows rely on TCP NewReno. 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 set presented in Table 1.

[2]*Fibbing* can be seen as another recent example of algorithmic virtualization in the routing layer [18].

[3]All of our Linux code and experimental settings are publicly available on Github [21].

We subsequently 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 vSwitch 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 provides 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. These techniques are roughly ordered from the most to least intrusive. The first two techniques are specific to hypervisors, while those following can also be used in network edge middleboxes, including several existing techniques that were proposed in the literature to regulate the rate of TCP flows [22–26]. In this paper, we focus on the simpler and least intrusive techniques, since they are the most appealing and practical to implement.

Additionally, we explain the drawbacks of each technique, including how each may violate networking architecture principles. *Not all lies are created equal*: breaking the end-to-end principle can be considered more severe than merely reducing the receive window.

Write into guest memory. Modern virtualization techniques such as active memory introspection [27, 28] and industry products such as VMware's VMSafe [29] 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. It could provide access to the congestion window without the need to maintain state to track it in the hypervisor.

Cons. Again, tenants may not accept that the hypervisor gets 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 [19, 20]. It can acknowledge packets to the guest VM at some 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 requiring 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, potentially 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 [20, 30].

Example. In order to solve TCP incast, it can be useful to reduce the retransmission timeout value $\mathrm{RTO}_{\mathrm{min}}$. vCC can buffer in-flight packets and retransmit according to its own $\mathrm{RTO}_{\mathrm{min}}$ buffer, even when the guest OS does not support the desired value or does not support changing the $\mathrm{RTO}_{\mathrm{min}}$ at all.

Cons. The hypervisor needs to manage packet buffers. Generally, packet buffers may also increase latency when they are used to store packets coming in too quickly (instead of copies of sent packets). Large buffers significantly increase the memory footprint of vCC.

Buffer ACKs. The hypervisor can similarly buffer received ACKs [23–25]. If an ACK is piggybacked on data, the acknowledged sequence number is reduced and the remaining bytes to acknowledge are later sent as a pure ACK.

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[4]. 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 [22, 23, 25] 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, a delicate point to note is that the receive window should not be decreased to less than the current number of in-flight packets. This may conflict with common implementations of the TCP buffer management. Therefore, the hypervisor needs to manage a gradual decrease while closely monitoring the connection state. Finally, a significant shortcoming is that while the technique

[4]Although it does not seem to directly go against RFC 5681 [31], which mentions the possibility of the replication of ACK segments by the network.

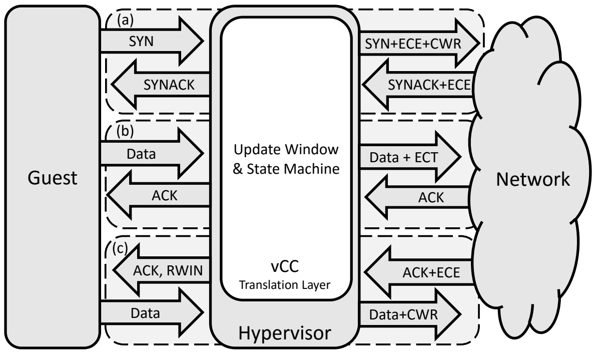


Figure 3: Interactions of the vCC translation layer in the hypervisor with TCP traffic. From top to bottom: (a) Connection negotiation, where the translation layer enables the ECE and CWR flag in the SYN packet to indicate ECN support, but hides the ECE field in the returning SYNACK; (b) normal data 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 bit is received, the translation layer masks the ECE bit, modifies the RWIN in the TCP header, and sets CWR on the next outgoing packet.

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 without additional techniques.

These techniques can translate the congestion control most accurately when 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, guest introspection, or other communication with the guest. However, if the hypervisor either does not know or does not want to trust the information [32], it could simply limit the sender; *e.g.,* when applying the receive window throttling technique, it could drop anything beyond the allowed receive window.

In addition, note that these techniques can be implemented either on a single side of the flow (*i.e.,* receiver or sender), yielding a *virtual-to-native* communication, or on both sides, yielding a *virtual-to-virtual* communication. When the guest already implements the target modern congestion control algorithm, vCC can either tunnel its traffic transparently, or still translate the traffic to make sure it obeys the exact same protocol implementation as 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 (we later implement a proof-of-concept of this solution in Section 3). 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 vCC 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 either is ECN-enabled, or also has a vCC translation layer. In addition, in all these cases, we need to recompute the checksum when the fields change. We can do so by looking at the changed bytes only.

Formally, when using the three-way handshake and receive window techniques, we are able to prove that we can *exactly emulate ECN and DCTCP* (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 NewReno state machine that is based on RFC 5681 [31] and RFC 6582 [33] 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 NewReno. We can then prove the following emulation theorems:

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

THEOREM 2. *The translation layer can exactly emulate DCTCP given a non-ECN TCP NewReno guest.*

The full formal proofs of these two theorems are in the Appendix. 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 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 at an ECN or DCTCP sender, it may reduce its congestion window, while we assume that the notification should be ignored by a non-ECN TCP sender. The second insight 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 in the vCC translation layer: *e.g.,* we could translate TCP NewReno with ECN to DCTCP by concatenating (a) TCP NewReno with ECN to TCP NewReno without ECN (simply modify the three-way handshake); and (b) TCP NewReno without ECN to DCTCP (as shown above).

3. EVALUATION: SOLVING ECN UNFAIRNESS

In Sections 3 and 4, 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 in this Linux vCC system are carried out with Mininet [34] for reproducibility. Our experiments use a virtual machine running Ubuntu 14.04 with Linux kernel version 3.19, except for the experiments with 1 Gbps links, which were 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 vCC can provide bandwidth sharing in this hypervisor environment.

3.1 ECN Unfairness

ECN allows flows to react to congestion before any data has been lost [35]. ECN can be a valuable tool to increase network performance, but it has not been widely supported in operating systems until recently [36]. 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. To demonstrate the ability of vCC-augmented guests to interact with any guest in a *virtual-to-native* communication, we set the receiver to be a simple native Linux guest without vCC. As a result, it can be seen as a non-ECN receiver for non-ECN flows, and an ECN receiver for ECN and virtual-ECN flows. All links have a bandwidth of 100 Mbps and a delay of 0.25 ms, so the RTT is 1 ms. The switch queue uses RED with parameter set RED1 as detailed in Table 1. We use TCP NewReno 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 [37]. 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 instead. 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 Table 1. The results are presented in Figure 5. The same trend is evident.

We next analyze the impact of different ratios of ECN to non-ECN flow numbers and of various RED parameter sets (Table 1) on this ECN unfairness. 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 [1]. The $\text{RED}_{\text{burst}}$ parameter is set to the minimum allowed value in tc-red for RED1 parameters. RED2 is the recommended setting for RED in the tc-red man page example. RED3 is a modification of RED2 (modified RED_{prob}) to test a more aggressive marking/dropping policy.

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

Table 1: RED Parameters used in the experiments.

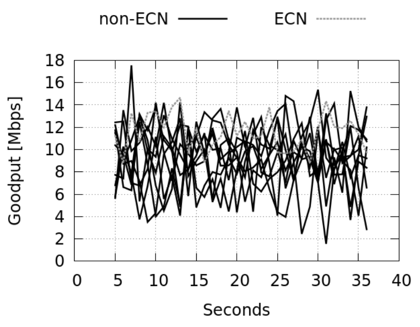
Figure 6 plots the ratio of the mean non-ECN flow goodput to the mean ECN flow goodput, *i.e.,* a measure of this unfairness, as a function of the number of ECN flows, given a total of 10 flows. It illustrates how for all tested parameter sets, introducing even a small number of ECN flows into the mix breaks fairness between ECN and non-ECN flows. Moreover, when there is only one non-ECN flow left out of the 10 flows, its goodput is at most 45% of the goodput of the ECN flows.

Figure 7 explores how modifying the RED_{min} parameter in the RED1 parameter set affects fairness. We set $\text{RED}_{\text{max}} = \text{RED}_{\text{min}} + 1$, and set $\text{RED}_{\text{burst}}$ to the minimum allowed by tc-red. The figure depicts the goodput ratio for different values of RED_{min}. In general we see that as the proportion of ECN flows in the mix increases and as RED_{min} decreases, the unfairness worsens and the non-ECN flows suffer from increasing goodput loss.

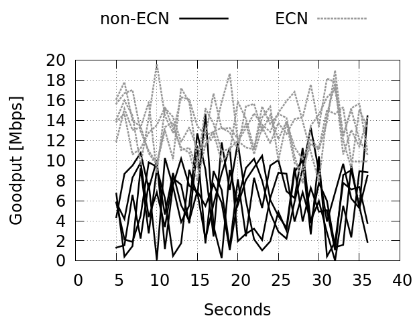
What is causing this unfairness? Figure 8 presents a 100 ms binned histogram of the time between consecutive acknowledgments sent by the receiver to a non-ECN and to an ECN flow, where the non-ECN flow is competing with 9 ECN flows on a 100 Mbps link using RED1 parameters. The non-ECN flow suffers from repeated retransmission timeouts, as seen by the 200 ms and the 600 ms 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}. It 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 losing a packet or experiencing a timeout.

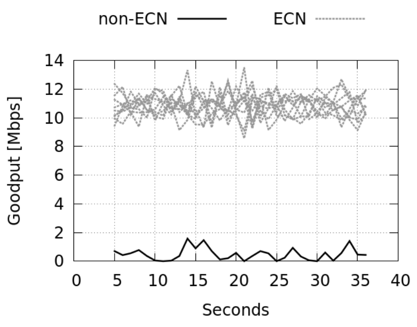
ECN to non-ECN flows ratio. Why does the unfairness to non-ECN flows become more severe as the proportion of ECN flows increases? Assume the switch buffer becomes congested, *i.e.,* crosses the marking threshold beyond which ECN packets are marked and non-ECN packets are dropped. Then packets from the ECN flows continue to enter the buffer for at least an RTT, potentially keeping the buffer congested. As a result, the switch may drop long sequences of non-ECN packets, causing timeouts in non-ECN flows. This effect is particularly pronounced with a higher proportion of



(a) 9 non-ECN vs. 1 ECN flows

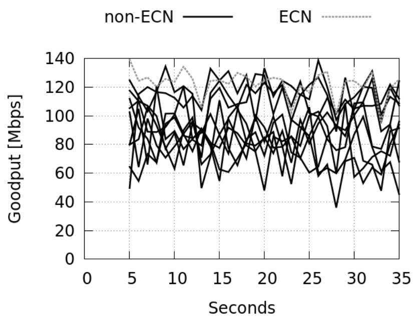


(b) 5 non-ECN vs. 5 ECN flows

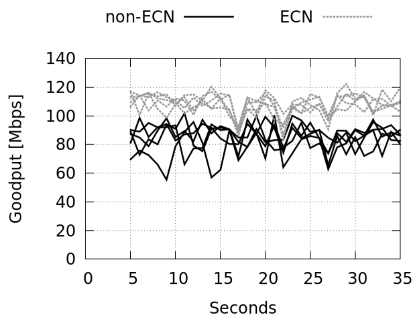


(c) 1 non-ECN vs. 9 ECN flows

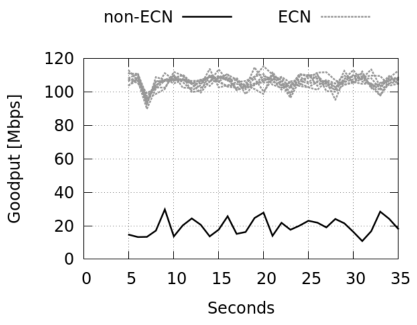
Figure 4: Unfairness between ECN and non-ECN flows, given a constant total number of 10 flows going through a shared 100 Mbps bottleneck link. As the ratio of ECN to non-ECN flows increases, the non-ECN flows suffer from increasing starvation and can send fewer and fewer packets.



(a) 9 non-ECN vs. 1 ECN flows

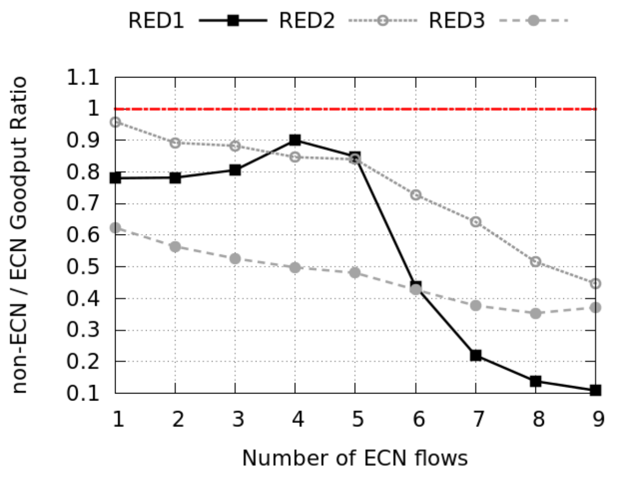


(b) 5 non-ECN vs. 5 ECN flows

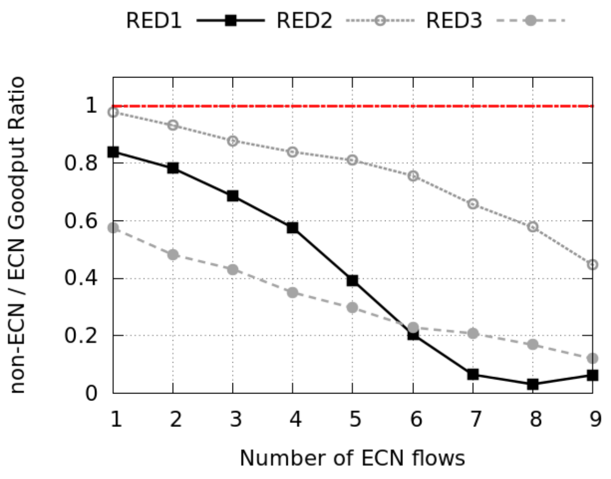


(c) 1 non-ECN vs. 9 ECN flows

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



(a) 10 Mbps links



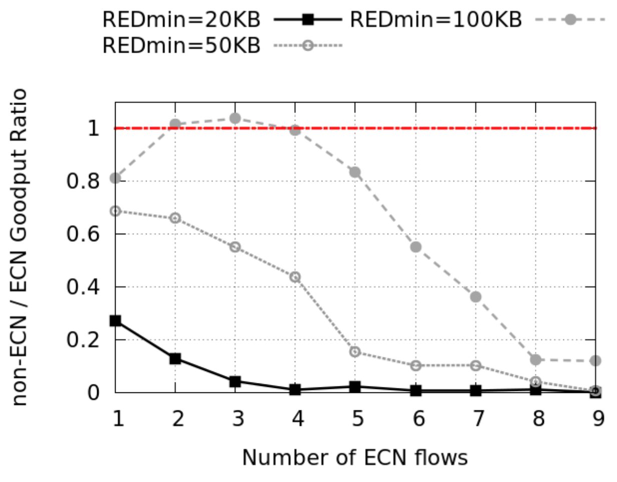
(b) 100 Mbps links

Figure 6: Unfairness between ECN and non-ECN flows for several flow-type mixes 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.

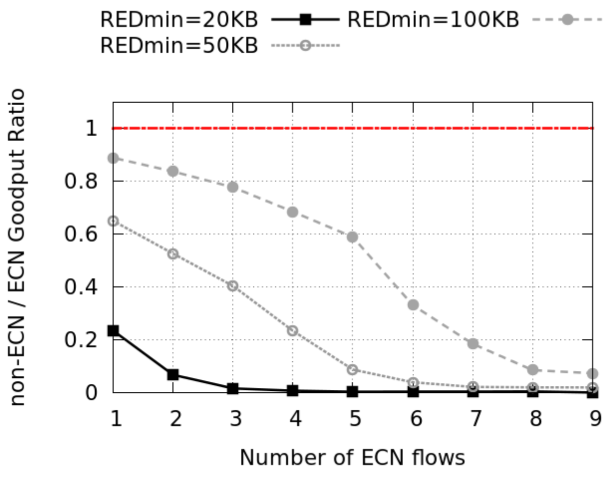
ECN flows, which typically leads to a higher ECN traffic rate. As a result, it will take longer to drain the queue below the marking threshold as more ECN traffic keeps arriving, and therefore may cause a longer congestion period.

3.2 Receive-Window Throttling

In order to address this unfairness 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



(a) 10 Mbps links



(b) 100 Mbps links

Figure 7: Average goodput ratio with varying values of RED$_{\text{min}}$, given 10 senders. Increased numbers of ECN flows lead to starvation of non-ECN flows.

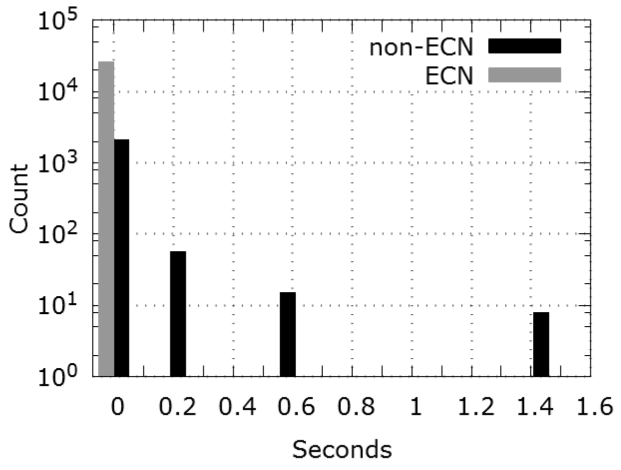


Figure 8: Histogram of time between consecutive acknowledgments sent by the receiver, divided into 100 ms bins, given a single non-ECN flow competing with 9 ECN flows on a 100 Mbps link. A representative ECN flow is plotted along with the non-ECN flow.

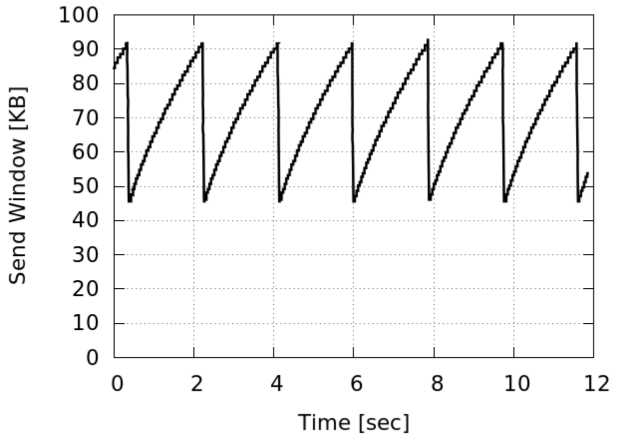


Figure 9: Send window time series for a virtual-ECN flow.

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 sender uses virtual-ECN provided by the vCC translation layer (wherein the ECE bits are hidden from the guest to simulate ECN-ignorance). The switch is configured with the RED1 parameter set from Table 1. The sender-to-switch link has a bandwidth of 12 Mbps, while the switch-to-receiver link has a bandwidth of 10 Mbps. The delay of each link is 250 μs (*i.e.,* RTT = 1 ms). The system is given 5 seconds to stabilize before data is collected for 12 seconds.

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 experiment, 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 run an experiment with 10 identical senders connected with 10Mbps links to a single receiver through a single switch.

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(c), once virtual-ECN is activated, the lost capacity is regained as virtual-ECN can react to congestion without dropping packets and retransmitting them (exactly like ECN's behavior in Figure 10(b)).

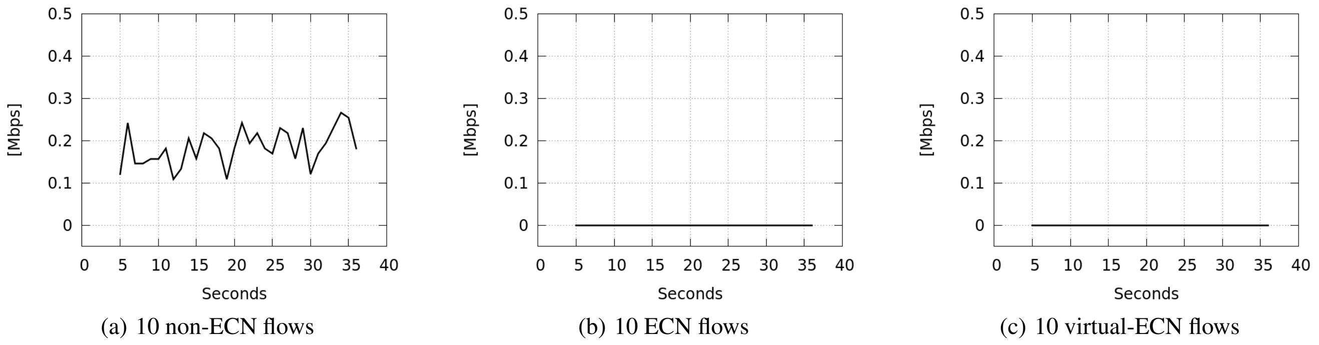

(a) 10 non-ECN flows (b) 10 ECN flows (c) 10 virtual-ECN flows

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

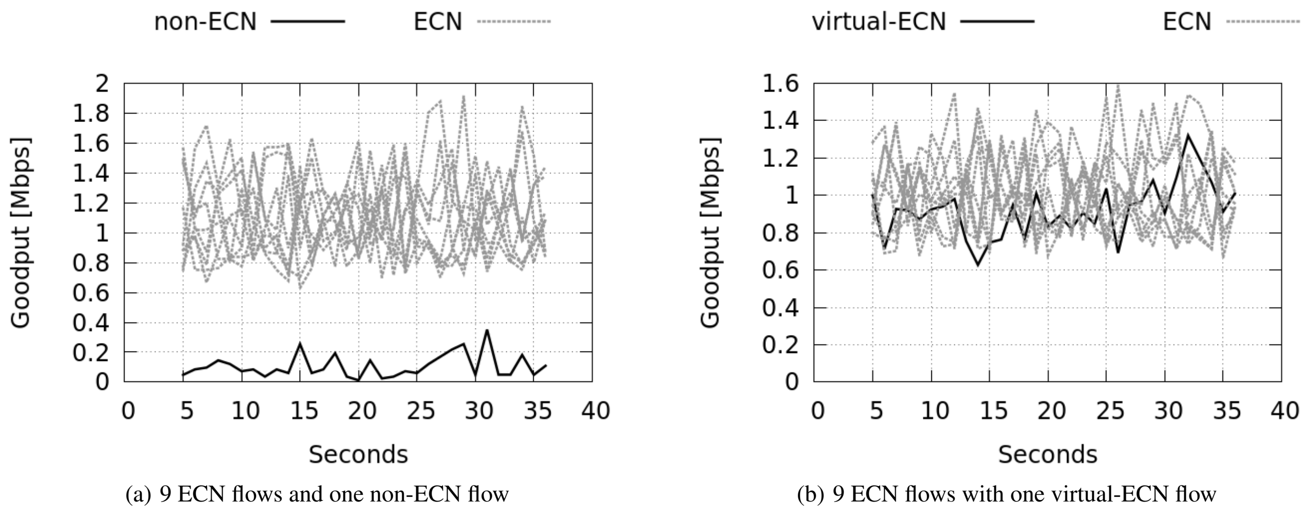

(a) 9 ECN flows and one non-ECN flow (b) 9 ECN flows with one virtual-ECN flow

Figure 11: 9 ECN flows share a 10Mbps bottleneck with either (a) one non-ECN flow; or (b) one virtual-ECN flow. (a) The non-ECN flow goodput is only 14.2% of the average goodput of the ECN flows. The fairness index is 0.921. (b) When virtual-ECN is used, the average goodput of the virtual-ECN flow is 103.8% of the average ECN flow goodput, and the fairness index is 0.994.

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 address the unfairness discussed in Section 3.1.

Figure 11(a) plots the goodput achieved with 9 ECN flows and one non-ECN flow sharing a 10 Mbps bottleneck link. It shows again how the non-ECN flow suffers from strong unfairness.

Figure 11(b) shows the goodput achieved in the same setting, except that the non-ECN flow has been replaced with virtual-ECN. The resulting goodput of the flow from the ECN-incapable guest is now similar to that of its ECN-capable peers, with goodput 103.8% of the average goodput of the ECN-capable flows.

To summarize, the translation layer uses receive-window throttling to cause the guest that does not support ECN to mimic its ECN 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 [38] 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 can 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 [39]. 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. The hypervisor detects the signature of a tenant, port or packet, and restricts 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 two 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 time-sensitive or higher-paying tenants, for example. Three ports are given to unpreferred flows, and two to preferred flows. The total amount of window space, *i.e.,* 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 12 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.

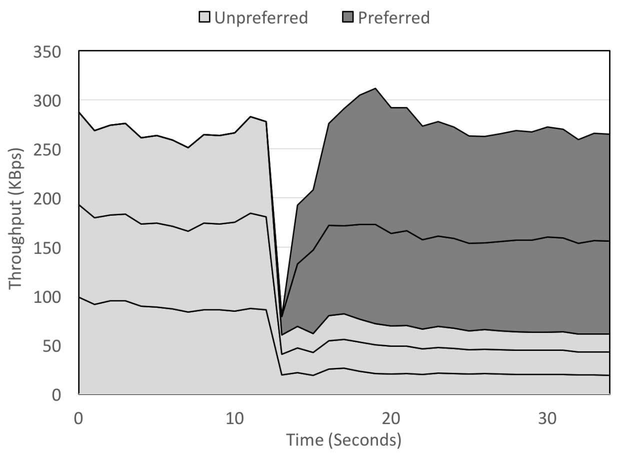


Figure 12: Stacked throughputs for three unpreferred and two preferred flows. The flows are receive-window throttled by the ESXi vCC layer. The sum of the windows of all the live flows is kept constant throughout the experiment, but the vCC throttles unpreferred flows once preferred flows start in order 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 [40], VMware's ESXi DVFilter [38], 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 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 such a case, the delay would be on 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 [41–44]. 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 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, or 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. Note that it is easy to configure vCC to not modify certain traffic. Future work could include automatic detection of flows that need translation, reducing the need for administrator configuration.

Delay-based congestion control. We believe vCC can translate to/from delay-based TCP algorithms like TCP Vegas and TIMELY [8]. To do so, it would need to use the more heavyweight techniques at its disposal, such as split connections and buffers.

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 translation 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.

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.

Encryption: Our analysis suggests that the vCC architecture can similarly be used to offer encryption services, such as TCPCrypt and IPSec [45, 46], to legacy unencrypted TCP flows. If the guest is already encrypting communications, vCC would need to access session keys in order to operate, for instance by reading guest memory.

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.

6. RELATED WORK

AC/DC. AC/DC [47] has independently and concurrently introduced a similar approach to ours. It shares many of the main goals and ideas of this paper. AC/DC suggests that datacenter administrators could take control of the TCP congestion control of all the VMs. In particular, it demonstrates this approach by implementing a vSwitch-based DCTCP congestion control algorithm. AC/DC provides a thorough evaluation, including a demonstration of the effectiveness of AC/DC in solving the incast and fairness problems identified in [7] and CPU overhead measurements.

We view our virtualized congestion control (vCC) solution as a general framework for translating between congestion control algorithms. For example, our framework allows the translation of only the legacy flows, or the translation of only the sender (or receiver) side. Our experiments show that vCC allows virtually-translated legacy flows to fairly co-exist with modern non-virtualized flows. On the conceptual side, we survey additional translation techniques, such as introspection and split-connection. In addition, we provide emulation proofs for ECN and DCTCP in specific cases. Beyond vCC, we also introduce the general concept of algorithmic virtualization for legacy algorithms.

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

Several papers have also suggested that the congestion control algorithm could adapt to the datacenter network conditions, e.g. by using 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 attempt to regulate the sending rate of each TCP flow [22–26]. These papers present the techniques of buffering TCP packets and ACKs, as well as throttling the receive window. Our vCC approach uses similar approaches. While these papers typically attempt to reach a fixed sending rate, the goal of vCC is to translate between legacy congestion control algorithms and *any* modern congestion control algorithm.

Link-level retransmissions. In wireless communications, the Snoop protocol [20, 30] 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 [19, 20, 48]. In contrast, vCC need not break the end-to-end principle; it can keep the original connection and does not need to create ACKs for data not received by the destination receiver.

Fairness between VMs. An alternative approach for providing fairness between VMs is to use datacenter-wide isolation-based techniques that are able to enforce bandwidth guarantees while attempting to maintain work-conserving link usage [49–53]. Also, a related approach for the multitenant datacenter owner is to strictly divide the bandwidth among the tenants, giving each a fixed allocation [17]. Rate limiters and shapers in hypervisors and in NIC hardware [41, 54] can also help enable better fairness between local VMs.

7. CONCLUSION

Our goal was to make it easy to deploy new and improved congestion control algorithms into multitenant datacenters, without having to worry about TCP-friendliness with non-participating virtual machines. This paper presents vCC, which enables the datacenter 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 show 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, Gal Mendelson, Asaf Samuel, as well as our shepherd, Vishal Misra, and our anonymous reviewers.

This work was partly supported by the Platform Lab and ONRC (Open Networking Research Center) at Stanford, funding from Intel and AT&T, the Hasso Plattner Institute Research School, the Gordon Fund for Systems Engineering, the Technion Fund for Security Research, the Israeli Consortium for Network Programming (Neptune), and the Israel Ministry of Science and Technology.

9. REFERENCES

[1] 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.

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

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

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

[5] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for data center networks. *ACM CoNEXT*, 2012.

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

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

[8] 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.

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

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

[17] 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.

[18] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central control over distributed routing. *ACM SIGCOMM*, 2015.

[19] 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.

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

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

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

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

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

[25] 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.

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

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

[28] 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.

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

[30] 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.

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

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

[33] RFC 6582. https://tools.ietf.org/html/rfc6582.

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

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

[36] 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.

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

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

[39] 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. *ACM SIGCOMM*, 2015.

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

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

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

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

[44] 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.

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

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

[47] Keqiang He, Eric Rozner, Agarwal Kanak, Yu Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. *ACM SIGCOMM*, 2016.

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

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

[50] 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.

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

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

[53] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM*, 2013.

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

[55] TCP Congestion Control for Datacenters. https://tools.ietf.org/html/draft-bensley-tcpm-dctcp-05.

[56] Proportional Rate Reduction for TCP. https://tools.ietf.org/html/rfc6937.

APPENDIX

A. EMULATION PROOFS

Our goal in this appendix is to prove the emulation results of Theorems 1 and 2.We start by proving emulation results for TCP-Reno guests. Then, we build on these proofs to demonstrate the emulation results in Theorems 1 and 2 for TCP-NewReno guests.

A.1 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 bytes sent but not yet cumulatively acknowledged. W_{eff} is the effective send window of the TCP sender and is defined as follows:

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

Namely, the effective send window is the number of *additional* bytes 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 nth (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 nth event. For example, $W_c^{gst}(n)$ is the congestion window of the guest and $\mathit{fs}^{vcc}(n)$ 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.

A.2 TCP-Reno congestion control

For our proofs, we use a conceptual TCP-Reno congestion control state machine that corresponds to RFC 5681 [31]. Specifically, we consider the TCP-Reno state diagram in Figure 13. Then, the congestion parameters obey the following rules:

Slow Start: Upon receiving m newly acknowledged bytes:

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

Congestion Avoidance: Upon receiving m newly acknowledged bytes:

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

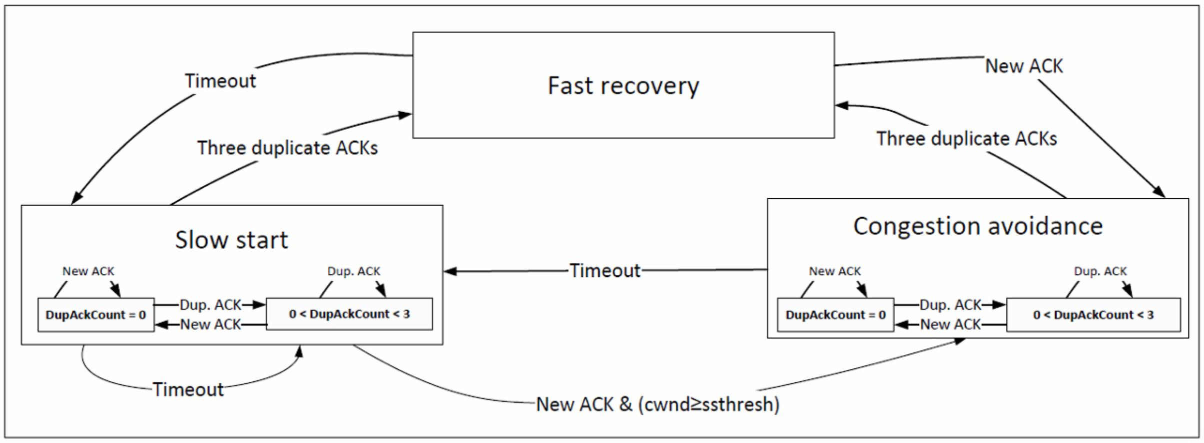


Figure 13: TCP-Reno congestion control state machine that corresponds to RFC 5681.

Fast Recovery: Upon any acknowledgment:

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

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 [55].

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 newly acknowledged bytes 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 the updated α value to reduce its window as follows:

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

where $W_{c,\,\textit{last}}$ is the size of the window just before applying Equation (13) and,

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

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.

A.3 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.1.

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 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 achieve 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.*

A.4 Reno 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.

Throughout our proofs we consider Assumptions 1 and 2. In addition, as mentioned, we assume that the hypervisor uses the same congestion control as the guest (including RTT estimation technique), uses the assignment in Equation (17), and supports ECN as described in Section A.2.

LEMMA 1.

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

and

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

PROOF. Since the hypervisor uses the assignment that is presented in Equation (17) 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$_{\text{min}}$ 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

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

Induction hypothesis: Assume that:

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

and

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

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:

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

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

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

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.

In addition, when ACKs get lost or delayed, we might have a cumulative ACK. Namely, the congestion window of the guest and the hypervisor grow according to the number of acked bytes. This does not affect the above analysis.

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.

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

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

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

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

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

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

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

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

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

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

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.

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

and

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

PROOF. DCTCP reaction to congestion flags is different than ECN as described in Section A.2. However, it still can only further reduce the congestion parameters of the hypervisor 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.

PROPOSITION 2. *The translation layer can exactly emulate a TCP Reno-based DCTCP given a TCP Reno guest.*

PROOF. Again, the proof is identical to the proof of Theorem 1, with the only exception that we rely on Lemma 2 instead of Lemma 1. ∎

A.5 The NewReno modification to Reno's fast recovery algorithm

We next extend our results to the TCP NewReno congestion control that corresponds to [33]. Essentially, the NewReno modification of Reno describes a specific algorithm for responding to partial acknowledgments. NewReno applies to the fast recovery procedure that begins when three duplicate ACKs are received, and ends when either a retransmission timeout occurs or an ACK arrives that acknowledges all of the data up to (and including) the data that was outstanding when the fast recovery procedure began. Figure 14 depicts a high-level diagram of the NewReno modification.

We first introduce a new variable that is used in TCP NewReno.

New (sender's side) state variable – *recover*: When in fast recovery, this variable records the send sequence number that must be acknowledged before the fast recovery procedure is declared to be over. Let R^{gst} and R^{vcc} be the recover variables of the guest and the hypervisor, respectively.

We next list our considered changes to the TCP Reno state machine that is presented in Figure 13 in order to obtain a corresponding TCP NewReno state machine.

Upon initialization: recover is set to the first sequence number sent. Trivially, we obtain:

$$R^{gst}(0) = R^{vcc}(0). \tag{34}$$

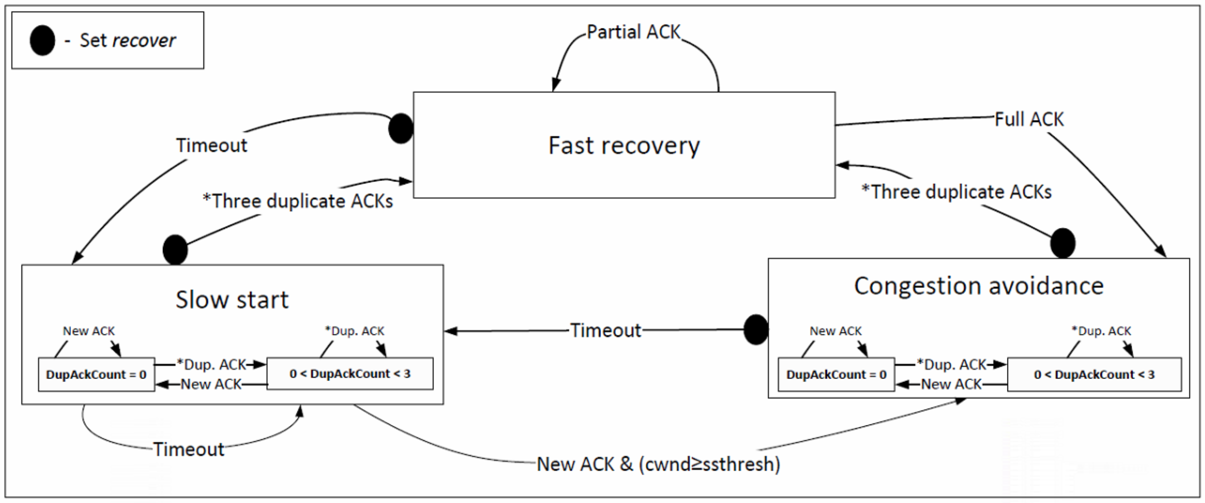


Figure 14: High-level illustration of the TCP NewReno modification of TCP Reno Fast recovery that corresponds to RFC 6582. A new congestion parameter denoted *recover* is introduced. Fast recovery continues until a Full ACK is received (i.e., it covers *recover*). The black circles indicate the events upon which *recover* is set to the highest outstanding sequence numbers. Duplicate ACKs are accounted for only if they cover more than *recover*.

Upon a third duplicate ACK:

Case 1: The cumulative acknowledgment field of this ACK covers more than recover. In that case:

$$R^{gst}(n) = k^{gst},\ R^{vcc}(n) = k^{vcc}, \tag{35}$$

where k^{gst} and k^{vcc} are the highest sequence numbers transmitted so far by the guest and the hypervisor, respectively. Enter fast recovery.

Case 2: The cumulative acknowledgment field of this ACK does not cover more than recover. In that case:

$$R^{gst}(n) = R^{gst}(n-1),\ R^{vcc}(n) = R^{vcc}(n-1). \tag{36}$$

Do not enter fast recovery.

Upon newly acknowledged data:

Case 1: Full acknowledgment – acknowledges all of the data up to and including recover. In this case, exit fast recovery.

Case 2: Partial acknowledgment – does not acknowledge all of the data up to and including recover. In this case, retransmit the first unacknowledged segment and stay in fast recovery.

Upon Timeout: record the highest sequence number transmitted in the variable recover, *i.e.,* Equation (35).

All updates to the congestion window and slow start threshold are assumed to correspond to TCP Reno, as detailed in Section A.2.

A.6 NewReno exact emulation

Since all updates to the congestion windows and slow start thresholds are assumed to be according to Section A.2, in order to obtain exact emulation it is sufficient to show that the recover value of the guest and the hypervisor are always identical.

LEMMA 3.

$$R^{vcc}(n) = R^{gst}(n) \quad \forall n, \tag{37}$$

PROOF. Again, we prove this lemma by induction on the number n of events at the guest and hypervisor senders.

Base: n=0. This holds according to Equation (34).

Induction hypothesis: Assume that:

$$R^{vcc}(n-1) = R^{gst}(n-1). \tag{38}$$

Inductive step: We split the proof into two possible cases (a case for each event followed by an update of recover).

Case 1: Timeout. The correctness in this case follows immediately from the *in-flight invariant* and the simultaneous timeouts of the guest and the hypervisor. Namely, both the guest and the hypervisor set recover to hold the highest sequence number transmitted, which is identical in both.

Case 2: Third duplicate ACK that acknowledges more than recover. The correctness in this case follows immediately from the *in-flight invariant* and Equation (35). ∎

COROLLARY 1. *The equations of Lemma 1 still hold when applied to a TCP NewReno-based guest and hypervisor.*

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

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

PROOF. Consider Corollary 1. Then, the proof is identical to the proof of Proposition 1. ∎

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

LEMMA 4.

$$R^{vcc}(n) = R^{gst}(n) \quad \forall n. \tag{39}$$

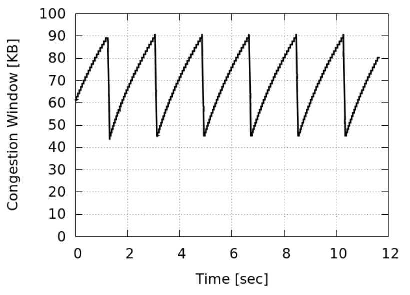
PROOF. The proof is the same as that of Lemma 3. ∎

COROLLARY 2. *The equations of Lemma 2 still hold when applied to a TCP NewReno-based guest and hypervisor.*

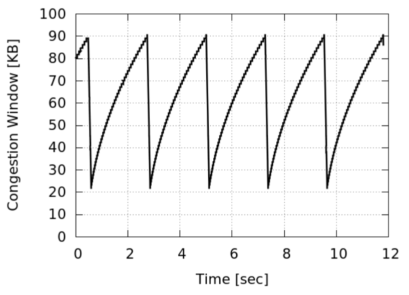
THEOREM 2. *The translation layer can exactly emulate DCTCP given a non-ECN TCP NewReno guest.*

PROOF. Consider Corollary 2. Then, the proof is identical to the proof of Proposition 2. ∎

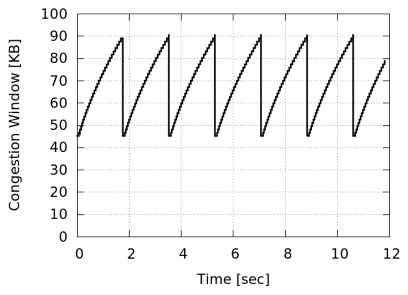
Interestingly, these proofs do not depend on the specific way to reduce the congestion variables of the hypervisor as a reaction to congestion flags. Thus, as long as the specific reaction does not trigger a change of state, a similar result can apply to arbitrary variations of the ECN and DCTCP congestion control algorithms.



(a) 1 non-ECN flow



(b) 1 ECN flow, PRR enabled



(c) 1 ECN flow, PRR disabled

Figure 15: Congestion window of a single non-ECN flow, a single ECN flow, and a single ECN flow with PRR disabled. The ECN flow experiences double drops, while the non-ECN flow and the ECN flow without PRR obey the textbook single drop.

A.7 Emulating ECN and DCTCP receivers

We have shown above how to exactly emulate ECN and DCTCP senders. We next briefly discuss the emulation of ECN and DCTCP receivers as well.

ECN receiver: When an ECN-capable receiver detects a CE flag in the header of a received packet, it sets the ECN-Echo flag in the TCP header of the subsequent ACK packet. The receiver continues setting the ECN-Echo flag on each outgoing ACK until it receives a CWR flag from the sender.

In the emulation, this functionality can be carried out by the hypervisor, which also resets the CE flags before forwarding the arriving packets to the guest receiver.

DCTCP receiver: To support delayed ACKs, the DCTCP receiver needs to maintain an additional congestion variable: *DCTCP Congestion Encountered* (DCTCP.CE). At the connection establishment, DCTCP.CE is initialized to false. Then, according to [55], when sending an ACK, the ECE flag is set if and only if DCTCP.CE is true. The updates to DCTCP.CE are as follows:

- If the CE flag is set and DCTCP.CE is false, send an ACK for any previously unacknowledged packets and set DCTCP.CE to true.
- If the CE flag is not set and DCTCP.CE is true, send an ACK for any previously unacknowledged packets and set DCTCP.CE to false.
- In any other case, ignore the CE flag.

To emulate this behaviour, the hypervisor needs to maintain DCTCP.CE and in addition to have knowledge about received but not yet acknowledged sequence numbers. This behaviour is regularly carried out by the hypervisor by maintaining two additional variables on top of DCTCP.CE that hold the last received (by the guest receiver) and the last acknowledged (by the guest sender) sequence numbers. While this may force the hypervisor to create and send an ACK packet, it does not require usage of any other techniques. In addition, in order to avoid redundant duplicate ACKs, the hypervisor needs to check the ACK sequence numbers sent by the guest sender. In case these sequence numbers predate the last sent ACK (by the hypervisor), the hypervisor updates the sequence numbers accordingly.

B. ECN + PRR = DOUBLE DROPS

In our experiments, we encountered a surprising phenomenon in regular ECN flows: *using both ECN and proportional rate reduction (PRR) [56] can cause double drops*. This is apparently the first mention in the literature of this phenomenon, which seems to go against the textbook single drop. Figure 15(a) depicts the congestion window time series for a single non-ECN flow in a Mininet experiment[5], while Figure 15(b) shows the congestion window for the ECN experiment; both are as reported by the tcp_probe kernel module. In the non-ECN case, as expected, the congestion window drops to half of its previous value. However, in the ECN case it drops to *a quarter*, *i.e.,* it drops twice upon congestion. We have verified this double drop by examining the packet traces on the receiver and sender. As Figure 16 illustrates, just after congestion starts, when the sender receives the first ACK with an ECE bit set (A_0), it almost immediately sends a packet P_1 with the CWR bit set. It will not respond with another window reduction during the remainder of the window of packets that were outstanding when the ECE bit was first received. However, P_1 arrives at the queue while congestion has not abated yet, so the switch sets the EC bit in the IP header of packet P_1 and of (at least) the next packet P_2 sent by

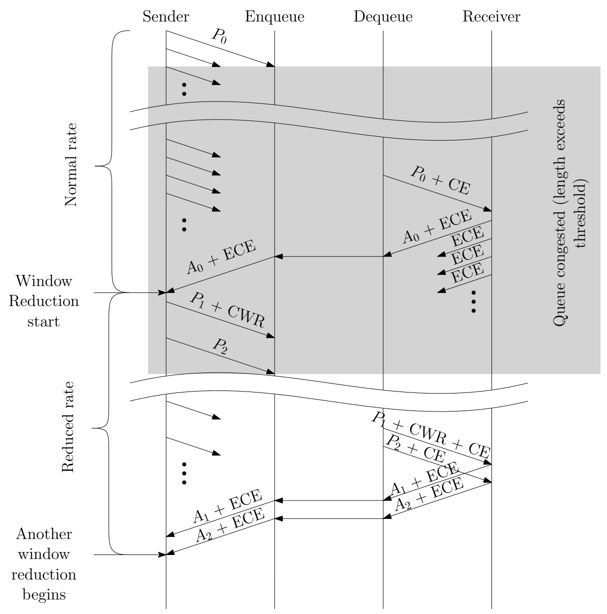


Figure 16: Double-drop diagram.

[5]The experiments here use the same topology and RED parameters as described in Section 3.2 and Figure 9.

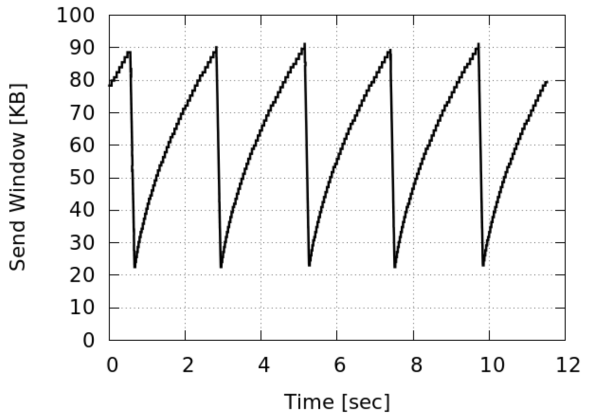


Figure 17: Send window time series for virtual-ECN.

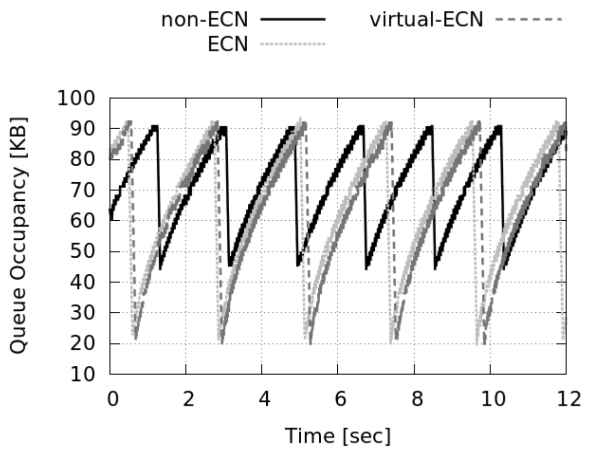


Figure 18: Queue occupancy for the non-ECN, ECN and virtual-ECN experiments.

the sender. Since this EC bit for P_2 arrives at the receiver after the CWR for P_1 is processed, it is interpreted as a *new* congestion notification and results in additional ECEs sent back to the sender, in ACKs acknowledging packets that were not outstanding when the first ECE was received. Therefore, the sender has to respond with another congestion window reduction.

The root cause for this double window reduction in ECN is the PRR mechanism implemented in Linux. PRR spreads out the window reduction across a whole RTT (or more specifically, across the ACKs on the outstanding window). It allows the sender to send packets right after receiving an ECE, and to send even more packets afterwards during the window reduction phase. This is different from a non-PRR implementation, where the congestion window is immediately halved upon receiving the first ECE, thereby preventing any packets from being sent until at least half of the outstanding window is acknowledged. By the time the transmitter can send a new packet, the queue size is reduced below the threshold. To verify this we modified Linux v3.19 and disabled its PRR mechanism. Figure 15(c) shows how we obtained again the usual single window reductions per congestion event.

Why do we not see the same effect for non-ECN flows? In this case, when the queue length exceeds the threshold, the newly arrived packet is discarded, and since packets are still dequeued and transmitted at the other end of the queue, the average queue length is reduced. In this experiment, this reduction is enough to ensure that no packet drop is detected (by receiving three duplicate ACKs) immediately after the window reduction phase is complete. Namely, the next packet drop is detected either (a) before the congestion window phase ends, which means it will not initiate another congestion window reduction, or (b) much after the window is reduced by half.

To make sure our virtual-ECN mimics the normal ECN behavior, we have incorporated the PRR mechanism into our receive-window throttling mechanism, which results in near-identical windows when comparing virtual-ECN's send window and ECN's congestion window, as seen in Figure 17. Compare this to Figure 9, where PRR is not implemented in the vCC mechanism. This is also evident in the comparison of the queue length occupancy in the three different experiments, as depicted in Figure 18. In simulations, we find that fairness between virtual-ECN and ECN flows is achieved both with and without PRR enabled in all flows.

We have also tested for this phenomenon with different link propagation delays (0.025 ms, 0.1 ms, 0.25 ms and 0.5 ms), different Linux kernels (3.13, 3.19), different link speeds (10 Mbps and 100 Mbps for the bottleneck link and 12 Mbps, 100 Mbps and 1 Gbps for the sender's link) and for the three RED parameter sets described in Table 1. The links speeds, kernel versions and link propagation delays have little to no effect on the double-drop phenomenon (except for links with 0.5 ms delay when using kernel version 3.13, where double-drops occur less frequently). The dominant factor affecting double-drops is the RED parameter set. The double drops are observed more frequently when the graph of the RED marking probability vs. queue occupancy gets closer to a step function (as in the RED1 parameter set). When using RED2 we see infrequent occurrences of double-drops. Such drops appear more frequently when using RED3 (which is a more aggressive version of RED2), and almost at each congestion event when using RED1.