Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport

Shiyu Liu and Ahmad Ghalayini, *Stanford University;* Mohammad Alizadeh, *MIT;*
Balaji Prabhakar and Mendel Rosenblum, *Stanford University;*
Anirudh Sivaraman, *NYU*

https://www.usenix.org/conference/nsdi21/presentation/liu

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport

Shiyu Liu[1], Ahmad Ghalayini[1], Mohammad Alizadeh[2], Balaji Prabhakar[1], Mendel Rosenblum[1], and Anirudh Sivaraman[3]

[1]Stanford University, [2]MIT, [3]NYU

Abstract

Recent datacenter transport protocols rely heavily on rich congestion signals from the network, impeding their deployment in environments such as the public cloud. In this paper, we explain this trend by showing that, without rich congestion signals, there is a strong tradeoff between a packet transport's equilibrium and transience performance. We then propose a simple approach to resolve this tension without complicating the transport protocol and without rich congestion signals from the network. Our approach factors the transport into two separate components for equilibrium and transient handling. For equilibrium handling, we continue to use existing congestion control protocols. For transients, we develop a new underlay algorithm, On-Ramp, which intercepts and holds any protocol's packets at the network edge during transient overload. On-Ramp detects transient overloads using accurate measurements of one-way delay, made possible in software by a recently developed time-synchronization algorithm.

On the Google Cloud Platform, On-Ramp improves the 99th percentile request completion time (RCT) of incast traffic of CUBIC by 2.8× and BBR by 5.6×. In a bare-metal cloud (CloudLab), On-Ramp improves the RCT of CUBIC by 4.1×. In ns-3 simulations, which model more efficient NIC-based implementations of On-Ramp, On-Ramp improves RCTs of DCQCN, TIMELY, DCTCP and HPCC to varying degrees depending on the workload. In all three environments, On-Ramp also improves the flow completion time of non-incast background traffic. In an evaluation at Facebook, On-Ramp significantly reduces the latency of computing traffic while ensuring the throughput of storage traffic is not affected.

1 Introduction

Datacenter packet transport has been an active area of research within the networking community for over a decade. The primary goals of datacenter transport protocols are to achieve high throughput and low latency and to effectively deal with bursty traffic, especially incast [51]. To achieve these goals, the research community has pursued two broad lines of work. The first is a series of congestion control algorithms that have relied on progressively richer forms of congestion signals from the network. These signals run the gamut from single-bit ECN marking [13] to multi-bit signals [53] and all the way to queue size and link utilization information [44]. The second line of work has focused on packet scheduling mechanisms that proactively prevent congestion from occurring in the first place [21, 34], or explicitly optimize for objectives like flow completion times [16, 17, 20, 49]. These schemes often require more elaborate network and application support, such as in-switch priority queues and application hints about flow sizes or deadlines.

Taking a step back from recent developments, we ask: *is it possible for a congestion control algorithm to achieve good behavior without rich congestion signaling or packet scheduling support from the network?* This question is not merely of academic interest; it has significant practical implications. In many environments, there is no way for the network to export rich signaling information or perform sophisticated packet scheduling. A particularly important example is the environment in which public cloud customers find themselves today. Cloud customers have access to the edge of the network, whether it is within a VM or in a bare metal server or potentially within SmartNICs in the future. However, they do not have access to the network infrastructure.

Motivated by the above question, we show that the trend towards rich congestion signals in state-of-the-art schemes is rooted in an inherent tension between the two main functions of a datacenter transport protocol: (i) converging quickly to a fair and stable equilibrium point as large flows arrive and depart, and (ii) handling transients of the incast-type effectively. Specifically, we consider two widely-deployed algorithms (TIMELY [47] and DCQCN [53]) and show that a parameter setting that works well in equilibrium performs poorly in transience and vice versa. On the other hand, protocols such as HPCC [44] which use richer congestion signaling from the network (e.g., queue size information and link utilization from INT [41]) can improve performance in both equilibrium and transience, as discussed in [44, §2.3 and §5.2].

Informed by these results, we ask a second question: *can congestion control be modularized into two simpler components, one each to deal with equilibrium and transient concerns?* This paper answers the question by proposing a new approach to congestion control that breaks the nexus between transience and equilibrium behavior. It delegates transient handling to a protocol, called *On-Ramp*, which is tailor-made for transients, leaving the congestion control algorithm to deal with equilibrium behavior.

On-Ramp (OR) can be coupled with any datacenter congestion control protocol to improve its performance during transients; indeed, implementing On-Ramp requires only end-host modifications and we have combined On-Ramp with CUBIC [33], BBR [19], DCQCN [53], TIMELY [47], DCTCP [13] and HPCC [44]. The core idea behind On-Ramp is extremely simple: when the congestion control protocol at a sender decides to transmit a packet, the sender temporarily holds back the packet if the sender-to-receiver one-way delay of the most-recently acknowledged packet exceeds a threshold. Thus, On-Ramp lets congestion control algorithms do what they are good at—determining transmission rates or window sizes in dynamic settings to improve metrics such as throughput and fairness—while giving them a leg up with functionality they struggle with: transience.

As a very useful by-product, On-Ramp also enhances the performance of existing congestion control algorithms in equilibrium by making them more robust to the choice of algorithm parameters. This is because of a phenomenon that we term *state-space compaction*. The state space of a network is the size of the queues within the network at any time. It spans a large dynamic range of queues sizes from transient (high queue) to equilibrium (low, but non-zero) to unstable (zero queues and link underflows). By compacting the state space, On-Ramp reduces the dynamic range of the state space observed by the congestion control algorithm and keeps it in a tight band around the desired operating point. This attenuates the large and frequent congestion signals from the network which, in turn, can cause an overreaction by the congestion control algorithm.

Key to making On-Ramp practical is an accurate measurement of one-way delay, which requires synchronized clocks between the sender and receiver. Since the one-way delays in a datacenter can be a few tens of microseconds or lower, this implies that the sender and receiver clocks must be synchronized to within a few microseconds. Such high-accuracy clock synchronization has traditionally required hardware-intensive protocols like Precision Time Protocol (PTP) [37]. But a recently developed system, Huygens [30], showed that it is possible to achieve nanosecond-level clock synchronization without special hardware or dedicated priorities. On-Ramp leverages the Huygens algorithm, making it easier to deploy.

We evaluated On-Ramp in three different environments: the public cloud, a CloudLab cluster (bare-metal cloud) and ns-3 simulations. We find:

1. **Performance improvements.** On Google Cloud, On-Ramp improves the 99$^{\text{th}}$ percentile request completion time (RCT) of incast traffic of CUBIC by 2.8× and BBR by 5.6×. In CloudLab, On-Ramp improves the tail RCT of CUBIC by 4.1×. In ns-3 simulations, which model more efficient NIC-based implementations, On-Ramp improves RCTs to varying degrees depending on the workload under DCQCN, TIMELY, DCTCP and HPCC. In all three environments, On-Ramp also improves the flow completion time of non-incast background traffic.
2. **CPU overhead of On-Ramp.** When running at 40% network load on a 10 Gbps NIC and an 8-core CPU, the total CPU utilization is 15.1% without On-Ramp, and 18.7% with On-Ramp. If On-Ramp is implemented in the NIC, this overhead can be eliminated.

2 Transience–Equilibrium Tension

Congestion control algorithms execute one of the following update equations:

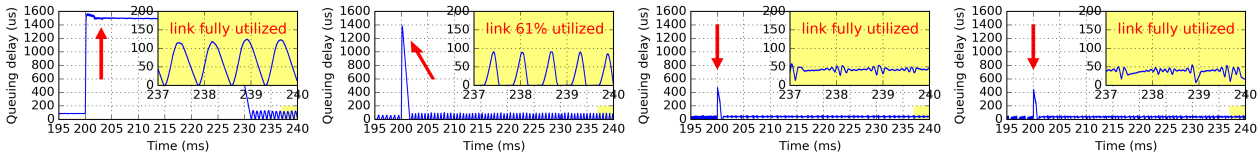
$$\begin{aligned} W(next) &= f(W(now),\ congestion\ signals,\ K), \text{ or} \\ R(next) &= g(R(now),\ congestion\ signals,\ K). \end{aligned} \tag{1}$$

That is, based on congestion signals received from the network, the algorithm updates the window size W or the transmission rate R. This is taking place constantly and iteratively, as flows arrive or depart or the path bandwidth changes (e.g., due to prioritization). K here is the gain in the control loop, which is typically very carefully chosen to provide stability in equilibrium and quick reaction to congestion in transience.[1] However, in the high bandwidth and small-buffered environment of data centers, we shall see that it is quite hard to pick the gain K so as to get great performance in *both* transience and equilibrium: a high value of K gives the responsiveness needed to react quickly to congestion but suffers from bad performance and instability in equilibrium when there are lags in the control loop; conversely, a low value of K can provide good performance in equilibrium but make the source sluggish during transience. We refer to this as the *transience-equilibrium tension* and we shall see that industrial-grade and commercially-deployed algorithms like TIMELY and DCQCN suffer from the transience-equilibrium tension.

With elaborate and frequent congestion signals from the network (e.g., queue depths, link utilization and flow rate, sent on a per-packet basis using in-band telemetry [41]), the congestion control algorithm can be improved simultaneously in transience and equilibrium (e.g., HPCC [44]). Unfortunately, such elaborate signals are not available in virtual environments such as the public clouds.

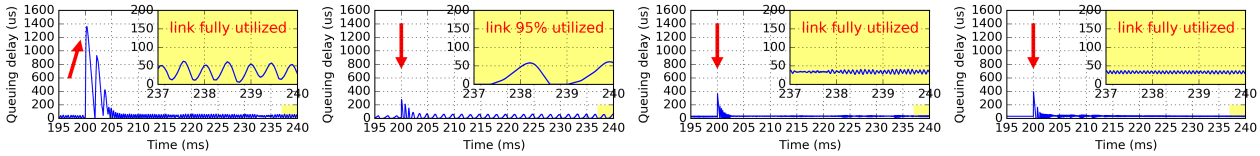
An alternative is to bake in two different modes of congestion handling within the same algorithm, e.g., slow start for transients and congestion avoidance for equilibrium. However,

[1] Indeed, an extensive literature in congestion control theory is devoted to the careful choice of the gain parameters, e.g. [14, 15, 24, 27, 40, 45, 52].



(a) β=0.2, no OR **(b)** β=0.8, no OR **(c)** β=0.2, with OR **(d)** β=0.8, with OR

Figure 1: 12 100G servers sending traffic to one receiver using TIMELY. 2 of them start at t=0, the other 10 start at t=200ms. From ns-3 runs using the setup in §5.1.1. The red arrow points to transience, and the yellow box is a zoom-in of equilibrium. OR threshold T=30μs.



(a) T_i=55μs, T_d=50μs, no OR **(b)** T_i=300μs, T_d=4μs, no OR **(c)** T_i=55μs, T_d=50μs, with OR **(d)** T_i=300μs, T_d=4μs, with OR

Figure 2: DCQCN study. OR threshold T=30μs.

this approach is fragile because it requires defining precise conditions to trigger switches between modes. Further, when a congestion control algorithm in equilibrium encounters severe congestion and drastically cuts its window or rate, it must make a difficult choice between remembering and forgetting its previous equilibrium state (rate/window). This choice is difficult because it depends on the duration of congestion, which is hard to predict. If the congestion is transient, such as an incast, the algorithm must remember its previous state so as to rapidly restore the old equilibrium without under utilization, once the incast ends. On the other hand, if the congestion is sustained, such as the simultaneous arrival of many long-lived flows, the algorithm must forget its previous state so that it can rapidly find a new equilibrium.

In this context, On-Ramp makes two key contributions: (i) when the one-way delay (OWD) on the path exceeds a given threshold, T, On-Ramp can quickly and forcefully react by pausing transmissions at the source, reducing congestion; and (ii) On-Ramp also reduces the sensitivity to the choice of the gain parameter K by ensuring good transient and equilibrium performance over a wide range of values of K.

We shall show that On-Ramp achieves the above by *compacting the state space* in the network. By that we mean that On-Ramp maintains network state variables such as queuing delays around their desired operating points, preventing large excursions (high or low) which occur during transience or in equilibrium when the value of K is high. Essentially, state space compacting *attenuates* (but does not eliminate) the congestion signals, preventing an overreaction at a source with high gain K while providing the throttling necessary for a source with low gain K.

Let us now illustrate the above points by considering the following scenario: 12 100G servers sending traffic to one receiver; all machines are connected to a single switch. 2 servers start sending at time 0; the other 10 start at 200 ms.

Figures 1a and 1b show the queuing delays at the switch during transience (t=200ms) and equilibrium (t=237-240ms) when TIMELY is used with its gain parameter $\beta = 0.2$ and $\beta = 0.8$, respectively. Note that $\beta = 0.8$ is the value recommended in [47, 54], and $\beta = 0.2$ is the largest value to achieve full utilization in equilibrium.

At $\beta = 0.2$, TIMELY performs well in equilibrium: the queue stays above zero and the link rate is maintained close to 100G. However, it performs poorly in transience: it takes a long time to react to the congestion caused by the newly arriving 10 flows—the queuing delay converges very slowly. At $\beta = 0.8$, it reacts much more quickly to transience, but due to aggressive congestion control, the queue underflows during equilibrium and leads to a link utilization of just 61%.

The trend is similar for DCQCN. The rate-increase timer T_i and rate-decrease timer T_d are varied to change its control gains. Note that the values T_i = 55 μs and T_d = 50 μs are the settings recommended in [53] and T_i = 300 μs and T_d = 4 μs are the default settings recommended by a vendor of network hardware (cf. [44]). At less aggressive gains (Fig. 2a), DCQCN performs well in equilibrium but reacts slowly during transience. Aggressive gain settings (Fig. 2b) give the opposite behavior.

Figures 1c and 2c respectively consider the scenario where the low (equilibrium-friendly) gain parameters for TIMELY and DCQCN are used in conjunction with On-Ramp. We observe that both algorithms react very quickly to transient congestion and converge smoothly to a stable equilibrium with full link utilization. Indeed, the equilibrium performance is actually improved by On-Ramp: the oscillation of queues during equilibrium is reduced! Figures 1d and 2d consider the high gain parameter scenario. We see that On-Ramp helps here as well by preventing severe queue undershoots and providing a high link utilization.

In conclusion, On-Ramp helps to cope with severe transient congestion by lowering queuing delays rapidly; conversely, it also prevents queues from underflowing, hence keeping a high link utilization. It achieves this by compacting the state space to a region around the desired queue size. As a useful by-product of this, it also reduces the sensitivity of the congestion control algorithm to gain parameters. It is critical to note here that On-Ramp *does not* perform the window or rate updates at Equation 1. This is left to the congestion control algorithm.

One might wonder if a more sophisticated delay-based pro-

tocol could make better use of precise one-way delay measurements. While an interesting possibility, we did not pursue this route for two reasons. First, On-Ramp allows us to decouple transient and long-term congestion management, and handle each at its own appropriate timescale without burdening a single protocol with both. Second, On-Ramp also composes very naturally with existing congestion control algorithms. It lets congestion control algorithms do what they are good at—improving long-term metrics such as throughput and latency, while taking care of transients for them. This factorized approach has allowed us to combine On-Ramp with several existing congestion control algorithms [13, 19, 33, 44, 47, 53].

3 On-Ramp Design

On-Ramp is a simple end-to-end flow control algorithm, sitting as a shim between the congestion control algorithm and the network (see Figure 3). On-Ramp aims to bring down the path queuing delays as quickly as possible by pausing the flow at the sender's end of the network when the measured OWD (which we denote as O) exceeds a threshold T. For a congestion control algorithm that does not control queuing delays on its own (e.g., TCP CUBIC), On-Ramp adds this functionality. For a congestion control algorithm that does control queuing delays on its own (e.g., DCTCP), On-Ramp works like a safeguard, for example, by reducing queue spikes during transience.

We first present a simple version of the On-Ramp algorithm that is intuitive but has queue oscillations and the possibility of under-utilization in the presence of feedback delay. We update On-Ramp by amending the rule for pausing, resulting in the final version of the algorithm.

3.1 Strawman Proposal for On-Ramp

As shown in Figure 3, On-Ramp is implemented underneath the congestion control protocol (CC) between the sender S and receiver R. It consists of two parts:

(i) Receiver side: Upon receiving a packet, the receiver sends an OR-ACK to the sender, which contains (1) the flow indicator (i.e., the 5-tuple representing the flow), (2) the sequence number of the received packet, and (3) the time at which the packet was received.

(ii) Sender side: The sender maintains two data structures for each flow: (1) a queue of outstanding packets belonging to that flow waiting to be transmitted; and (2) a value called $t_{NextPkt}$ representing when the next packet from the flow will be transmitted. Initially, $t_{NextPkt}$ is set to 0 and, upon the receipt of an OR-ACK, it is updated as follows:

$$t_{NextPkt} \leftarrow \begin{cases} t_{Now} + O - T, & \text{if } O > T \\ t_{NextPkt}, & \text{else.} \end{cases} \quad (2)$$

Here t_{Now} is the current system time. The flow will be paused until $t_{NextPkt}$ if that is larger than t_{Now}. If an OR-ACK is dropped, the sender will not be able to measure OWD O at this time, so $t_{NextPkt}$ will not be changed. The sender dequeues packets of the active (non-paused) flows in round-robin order.

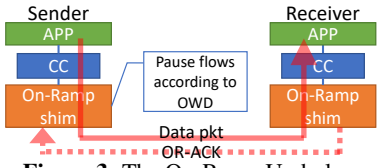


Figure 3: The On-Ramp Underlay.

The strawman proposal is simple and intuitive: upon receiving an OR-ACK with an OWD value of O exceeding threshold T, pause for $O-T$. The goal is to drain the queue such that the OWD after the pause is under T. This reasoning would have been correct *if there were no delay* in getting acks from the receiver. In the presence of feedback delay, however, the sender will actually pause for a significantly longer time than necessary.

To understand what is going on, suppose that the sender receives an ack with OWD O exceeding T for the first time at time t, and it immediately pauses for duration $O-T$. Notice that it will take at least one additional round-trip-time (RTT) after t for the sender to see the impact of this pause on the OWD values carried in acks. In particular, acks received for packets that were transmitted before pausing are likely to also carry OWD values exceeding T. Hence a sender using the strawman design will actually pause for at least the next RTT, even if the OWD exceeds T by a small amount. By the time it resumes sending traffic, the queue will have significantly undershot T, which risks under-utilization.

3.2 The Final Version of On-Ramp

To fix the above problem, we propose a simple mechanism to compensate for the sender's feedback delay in receiving the OWD signal. The key is to observe that it is possible the sender was paused when the green packet (see Figure 4) was in flight and before the sender received its ack. The update equation (2) doesn't take these previous pauses into account, and therefore it overestimates how much additional time it needs to pause for the OWD to drop down to T.

One approach to correct for previous pauses would be to subtract the total time the sender was paused while the green packet was in flight from the OWD value O obtained in the ack. This approach assumes that if the sender was paused for some duration P, then the current value of OWD is no longer O but rather $O-P$. However, this ignores the contribution of other senders to the OWD. The reduction in OWD due to a pause of duration P is *at most* P, but it may be less if other senders transmit in that time.

To account for other senders, On-Ramp estimates the relationship between the actions it takes (i.e., its own pause duration) and the effect of these actions (i.e., reduction in OWD). To this end, it dynamically measures a parameter β_m, which equals the change in OWD per unit of On-Ramp pause time; in other words, the *gain* of the On-Ramp control mechanism. Refer to Figure 4, and let O_B and O_G be the OWDs

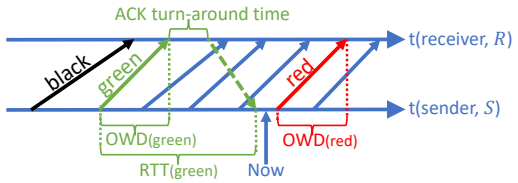
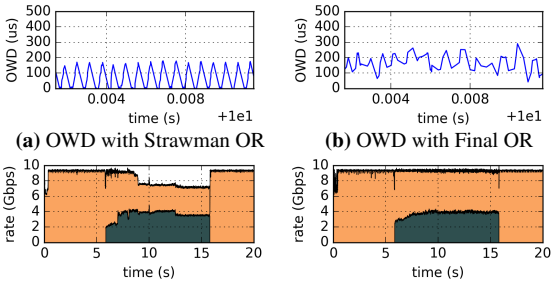


Figure 4: Timing Diagram of Packets Between S and R



(a) OWD with Strawman OR **(b)** OWD with Final OR

(c) Flow rates, Strawman OR **(d)** Flow rates, Final OR

Figure 5: Two long-lived CUBIC flows sharing a link

of the black and green packets, where the black packet is the one which has been acked immediately prior to the green packet being acked. Let P_{BG} be the total time On-Ramp had paused the sender in between the times it transmitted these two packets. If $P_{BG} > 0$, then $\beta_m = (O_B - O_G)/P_{BG}$ captures the effect of the sender pausing on the reduction in the OWD. We threshold β_m to be between 0 and 1. A value of β_m close to 1 indicates that the reduction in OWD is roughly the same as the sender's pause time. This would occur, for example, if there is little traffic from other senders, or if the senders are synchronized and all pause together. On the other hand, β_m near 0 indicates that On-Ramp must pause for a long amount of time to reduce OWD.

On-Ramp obtains one such β_m measurement for each new packet that it transmits for which $P_{BG} > 0$. It computes a moving average of these values with each new measurement:

$$\beta \leftarrow (1-g)\cdot\beta + g\cdot\beta_m \qquad (\text{if } P_{BG} > 0) \tag{3}$$

Here g is the EWMA gain. Finally, upon receiving an OR-ACK, the sender replaces the first line in Equation (2) with

$$t_{NextPkt} \leftarrow t_{Now} + O - T - \beta \cdot P_{LastPktRTT} \qquad (\text{if } O - \beta \cdot P_{LastPktRTT} > T) \tag{4}$$

Here, $P_{LastPktRTT}$ is the total time pause was asserted during the time period spanned by the RTT of the most recently acked packet (the green packet).

Figure 5 demonstrates the effectiveness of lag compensation. It shows the OWDs and the flow rates when two CUBIC flows share a bottleneck link in a bare-metal environment (Cloudlab [26]). The second flow is on during 6–16 seconds. The threshold T is set to 50μs. It's clear that the strawman On-Ramp leads to significant queue undershoot under T and causes under-utilization, and the final On-Ramp fixes this problem. Figure 24 in Appendix shows the case of 12 flows, where final On-Ramp removes most fluctuations in queue lengths and achieves better fair sharing among all flows.

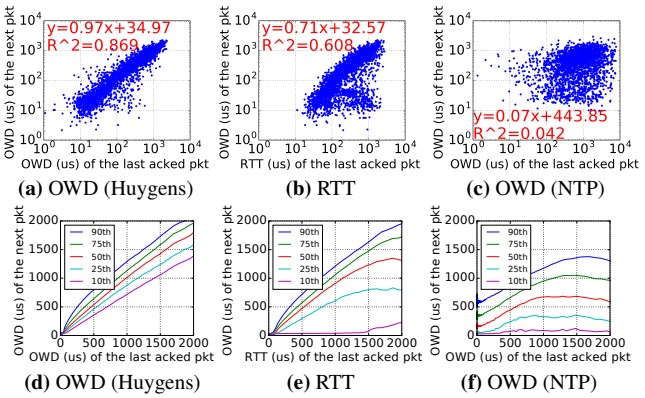
Parameter Selection. The threshold T should clearly be higher than the minimal OWD on the path, plus some headroom to tolerate errors in the OWD measurement. §5.1.4 describes how to pick T in practice. We choose the EWMA gain $g = 1/16$ and find that the end-to-end performance of On-Ramp is relatively insensitive to g, shown in §7.4.

3.3 Importance of Accurate One-way Delay

To measure OWD accurately, On-Ramp uses Huygens [30], a recently developed system for highly-accurate clock synchronization. The Huygens algorithm uses a random probe mesh among all the clocks; the mesh is formed by each clock probing typically 10 other clocks. The clocks exchange probes and acks on this mesh. We note that probes and acks are UDP packets *not* using higher priorities in switches. The send and receive timestamps of each probe and ack are processed through a combination of local and central algorithms every 2–4 secs. The local algorithm performs filtering using coded probes and support vector machines to estimate the discrepancies between two clocks. These techniques make Huygens robust to network queuing delays, random jitter, and timestamp noise. Then, the central algorithm, dubbed "network effect" in the paper, determines errors in the accuracy of clock sync using the transitivity property: the sum of the clock offsets A-B, B-C, and C-A is zero; else, errors exist. By looking at clock offset surpluses over *loops* of the probe mesh, Huygens pins down clock sync errors and provides corrections which can be applied offline or online.

Since the probe mesh is set up end-to-end at the hosts, there is no need for special hardware support (in contrast to other high-accuracy algorithms like PTP [37], DTP [43], or DPTP [39]). With NIC hardware timestamps, [30] reports a 99th percentile synchronization accuracy of 20–40 nanoseconds even under network loads of 80%. The probe mesh makes Huygens robust to high loads and link or node failures. Further, the local-central processing distributes effort across all nodes, making the algorithm scalable to 1000s of nodes. In the present paper, we use Huygens with CPU (software) timestamps because software timestamps are universally available in both VMs in public clouds and bare-metal machines. In this case, we see a median accuracy of a few hundred nanoseconds and 99th percentile accuracy of 2–3 microseconds under high network loads such as 80% in a single data center.

To see the importance of accurate one-way delay, we compare three signals that On-Ramp could use to measure path congestion: (i) OWD measured with accurately synchronized clocks using Huygens, (ii) round-trip time (RTT), and (iii) OWD measured with less accurately synchronized clocks using NTP [46]. Referring to Figure 4, we evaluate how well each of these signals, measured for the green packet, correlates with the OWD of the red packet. If the correlation is high, then the fate of the green packet is a good predictor of the congestion to be experienced by the red and immediately succeeding packets. As Figure 6a and 6d show, the OWD of the green packet measured using Huygens is highly corre-



(a) OWD (Huygens) **(b)** RTT **(c)** OWD (NTP)

(d) OWD (Huygens) **(e)** RTT **(f)** OWD (NTP)

Figure 6: The OWD of the next packet vs. the OWD (Huygens), RTT, and OWD (NTP) of the *last-acked packet*. (a)–(c) are samples plotted in log scale. (d)–(f) are percentiles plotted in linear scale. The test is conducted on 100 machines in a bare metal cloud with *WebSearch* [13] traffic and 40% network load.

lated with the OWD of the red packet. However, neither the RTT nor the OWD measured using NTP correlates well with the actual OWD experienced by the red packet. This makes On-Ramp much less effective with the latter two signals.

4 Implementation

In this section, we describe the Linux implementation of On-Ramp using kernel modules, which allows us to install On-Ramp by just loading a few kernel modules rather than reinstalling (or worse still, recompiling and then reinstalling) the whole kernel. We will also comment on the benefits—both implementation and performance—of implementing On-Ramp in Smart NICs in the future.

4.1 Linux Kernel Modules

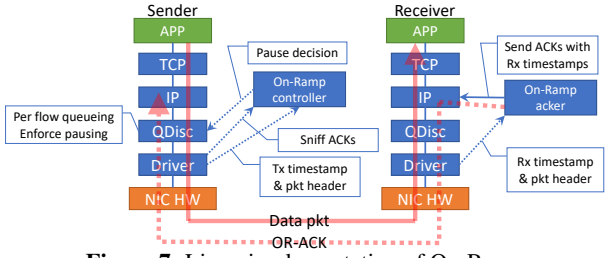


Figure 7: Linux implementation of On-Ramp

Figure 7 shows our On-Ramp implementation. It consists of four parts, described below. For each of them, we mention whether it involves changes to the sender, receiver, or both.

1. **On-Ramp controller.** This module runs at the sender and calculates OWDs and makes decisions to pause or not on a per-flow basis.
2. **NIC driver.** The NIC driver at the receiver is modified to timestamp packets (before GRO). The NIC driver at the sender is modified to timestamp packets, sniff OR-ACKs and forward them to the On-Ramp controller for implementing the On-Ramp algorithm. The NIC driver could be either a physical or a virtual NIC's driver.
3. **QDisc.** We modify the fair queueing (FQ) Qdisc [25] at the sender to queue packets into per-flow queues and exert pause on a per-flow basis.
4. **On-Ramp acker.** This module at the receiver sends OR-ACKs. These are UDP packets that use the same Ethernet priority as the received data packets, so they don't require any priority queues in Ethernet switches. We avoid piggybacking receive timestamps onto TCP ACKs, because they may be delayed by the TCP stack, and modifying the TCP stack requires recompiling the kernel.

Note that On-Ramp does not modify the existing data packets, and OR-ACKs are standard UDP packets.

There are three important details regarding the implementation that pertain to three critical aspects of On-Ramp: (i) the accuracy of measuring the OWD, (ii) the granularity of control (exerting pause), and (iii) the behavior after a pause ends. Accordingly, the On-Ramp implementation may vary depending on the deployment scenario, e.g., public cloud (Google Cloud Platform), bare-metal cloud (CloudLab), or bare-metal cloud with SmartNICs. We expand on these details below.

1. **Timestamp collection.** To compute OWD, we need to collect both sender and receiver timestamps. These timestamps are taken within the NIC driver and based on the system clock on both the sender and receiver sides. We choose the NIC driver because it is close to the wire and therefore minimizes additional software stack latency being added to the OWDs. This is important because stack latencies can be quite variable and confound the accurate detection of one-way delays in the *network* which are caused by congestion. Even though this makes On-Ramp's implementation NIC-driver-specific, the patch is only 20 lines of code and is quite easy to add to the NIC driver.[2] In bare-metal machines, if the NICs support hardware timestamping (e.g., through PHC [1]), NIC timestamps can also be captured, yielding less noisy OWD measurements, which, in turn, can lead to better control.
2. **The effect of generic send offload (GSO).** On the sender side, when GSO is enabled, the data segments handled by QDisc and the driver are GSO segments, which can be up to 64 KB (~43 packets). This limits the granularity of control by On-Ramp, as well as the accuracy of capturing transmit timestamps. §5 shows that On-Ramp already gives satisfactory performances with the default setting of GSO enabled and the max GSO size of 64KB. §7.2 shows that reducing max GSO size will further improve On-Ramp's performance, but increase CPU overhead, so there is a tradeoff here.
3. **Behavior after a pause ends.** One might wonder whether a burst of packets will be sent into the network after a pause ends, causing spikes in the queuing delay. This does not happen in practice. (1) For window-based CCs like CUBIC and DCTCP, although packets are also queued in the On-Ramp module, thanks to the TCP small queues patch [2] in Linux, the total number of bytes queued in the TCP stack and On-

[2] In a given public cloud, the vNIC implementations are the same, making the addition of the patch a one-time effort for each public cloud.

Ramp modules are limited. Furthermore, because On-Ramp is based on the FQ Qdisc, it dequeues in round-robin order across flows, smoothing out traffic after a pause ends. (2) For rate-based CCs like BBR, DCQCN and TIMELY, On-Ramp exerts pause by letting the rate pacer[3] hold back the next packet until $t_{Now} \geq \max(t_{NextTx}, t_{NextPkt})$, where t_{NextTx} is the time of sending the next packet determined by CC algorithm; $t_{NextPkt}$ is the value maintained by On-Ramp for each flow (see §3.1). When a pause ends, the rate pacer will resume releasing packets into the network according to the rate determined by CC, therefore the transmission will not be burstier.

Network and CPU overhead. The network overhead of On-Ramp comes from OR-ACKs; typically, 1 OR-ACK is sent per about 10 MTU-sized packets. Therefore, at 40% load on a 10 Gbps network and 78 Bytes per OR-ACK, the bandwidth consumed by OR-ACKs is about 21 Mbps, or 0.2% of the line rate. Huygens provides the clock synchronization service for On-Ramp. Its probe mesh adds negligible network overhead, which is about 3 Mbps, or 0.03% of the line rate.

For a typical scenario in §5, running 40% load of *Web-Search* traffic plus 2% load of incast on a 10 Gbps NIC and an 8-core Intel Xeon D-1548 CPU, the total CPU utilization is 15.1% without On-Ramp, and 18.7% with On-Ramp. If On-Ramp is implemented in the NIC as described later, this overhead can be eliminated. The CPU usage of Huygens is only around 0.5%.

We consider On-Ramp's overhead under higher network speeds. Table 1 shows overheads when two servers send iPerf flows to a third server simultaneously. Each server has a 25 Gbps NIC and a 10-core Intel Xeon E5-2640v4 CPU. We use the default GSO settings: enabled, max GSO size = 64KB. In this experiment, we get the same throughput with and without On-Ramp, both saturating the 25G link at the receiver. Most of the CPU overhead of On-Ramp is at the receiver, caused mainly by parsing, timestamping, and sending OR-ACKs. At the senders, On-Ramp operates at the granularity of GSO segments, so the overhead is small. Note that the On-Ramp implementation has not yet been optimized for CPU overhead.

	No On-Ramp[4]	With On-Ramp
Sender	1.01%	1.04%
Receiver	5.50%	6.99%

Table 1: The CPU usage in an iPerf experiment with 25G NIC

4.2 NIC Implementation

With the advent of programmable SmartNICs [3,7], On-Ramp can ideally be offloaded to the NIC in the future, conserving host CPU cycles. Moreover, a SmartNIC implementation can further improve On-Ramp's performance due to (1) shorter ack turn-around times at the receiver and (2) the exertion of pauses on MTU-sized packets, rather than GSO segments. §5.2.3 uses ns-3 [8] to emulate On-Ramp's performance using a NIC implementation.

[3] BBR uses FQ Qdisc to pace packets, which is compatible with the Linux kernel module implementation of On-Ramp. DCQCN's rate pacing is inside NICs, TIMELY's rate pacing can be in software or NICs, and we use ns-3 simulation to study On-Ramp's performance on top of them, as in §4.2.

[4] Since On-Ramp is built based on FQ QDisc, for a fair comparison of CPU overhead, we use FQ without On-Ramp as a baseline in this experiment. We note that FQ_CoDel [9] (the default QDisc in many modern Linux distributions such as Ubuntu 18.04) incurs a higher CPU overhead than FQ QDisc (when On-Ramp is not used): sender 1.41%, receiver 5.68%.

5 Evaluation

We evaluate the effectiveness of On-Ramp in a variety of environments and under different workloads and congestion control algorithms. In terms of performance, we consider:

1. Application-level performance measures: (i) request completion times for incast traffic, and (ii) flow completion times for non-incast background traffic.
2. Network-level performance measures: (i) number of packet drops and (ii) number of TCP timeouts.

5.1 Evaluation Setup

5.1.1 Evaluation environments

We consider three environments: public clouds, bare-metal clouds, and ns-3 [8] simulations. In the clouds we use On-Ramp's Linux implementation with different CC algorithms. The ns-3 simulations help us understand On-Ramp's performance in RDMA networks with different CC algorithms such as DCQCN, TIMELY, DCTCP, and HPCC.

VMs in Google Cloud. We use 50 VMs of type `n1-standard-4` [6]. Each VM has 4 vCPUs, 15 GB memory, and 10 Gbps network bandwidth. The OS is Ubuntu 18.04 LTS with Linux kernel version 5.0.

Bare-metal cloud in CloudLab. CloudLab [26] is an open testbed for research on cloud computing. We use the `m510` cluster [10] in CloudLab, which has 270 servers in 6 racks, 6 top-of-the-rack (ToR) switches, and 1 spine switch. The bandwidth is 10 Gbps between each server and ToR, and 4×40 Gbps between each ToR and the spine switch. Each ToR switch has a 9 MB shared buffer. Each server has an 8-core Intel Xeon CPU, 64GB memory, and a Mellanox ConnectX-3 10 Gbps NIC. For the On-Ramp evaluation, we rented 100 servers in this cluster (randomly chosen from these 6 racks by Cloudlab), and installed Ubuntu 18.04 LTS with Linux kernel version 4.15 on each server.

ns-3. We implement On-Ramp in ns-3 based on the open-source ns-3 simulator of HPCC [4,44]. We also use the same simulation setup as in [44, §5.1]. There are 320 servers in 20 racks, 20 aggregation switches and 16 core switches. The network topology is a 3-stage FatTree [12], consisting of ToR, aggregation and core switches. Each server has a 100 Gbps link connected to the ToR switch. All links between core, aggregation and ToR switches are 400 Gbps. Each link has 1 μs delay.[5] Each switch has a 32 MB shared buffer. Because

[5] Consisting of the link propagation delay and the packet processing delay in the corresponding switch.

the ns-3 simulations consider RDMA flows, we have priority flow control (PFC) [38] in effect. Note that neither the Google Cloud nor the CloudLab experiments has PFC; hence, buffers overflow and result in packet drops.

5.1.2 Traffic loads

We have two categories of traffic: incast type traffic [51] where requests generate bursts of small equal-sized flows simultaneously, and background traffic consisting of flows of varying sizes.

The incast traffic has a fanout of 40, where each of the 40 flows is either 2KB [13] or 500KB [44], so the total request size is 80KB and 20MB, respectively. The 2KB-sized flows are common in query-type scenarios while the 500KB-sized flows occur in query- and storage-type settings. The average load due to incast traffic is 2% or 20%.

For the background traffic, we use three data center workloads, namely: (1) *WebSearch* [13]: the web search traffic measured in Microsoft production clusters; (2) *FB_Hadoop* [50]: the traffic measured in Hadoop clusters at Facebook; (3) *GoogleSearchRPC* [5,49]: the RPC traffic generated by search applications at Google. Figure 8 shows the distribution of flow sizes. We adjust the average interval between adjacent flows to make the average traffic load to be 40%, 60% or 80%.

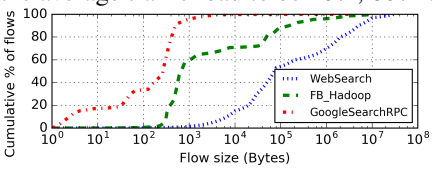


Figure 8: Distribution of flow sizes in the background traffic

5.1.3 Clock synchronization and packet timestamping

On-Ramp needs clock synchronization across servers to measure OWD. For bare-metal cloud and VMs in the public cloud, we use Huygens [30] as the clock synchronization algorithm. We find that the standard deviation of clock offsets after synchronization is around 200 ns in Google cloud, 100 ns in CloudLab, and the 99th percentile is less than 3 μs in both cases. In ns-3, by default the clocks are perfectly synchronized. However, to mimic clock inaccuracy in the real world even under good clock synchronization, we add a random offset to each server's clock according to a Gaussian distribution with a standard deviation of 200 ns.

For VMs in Google Cloud, the packet timestamps are taken inside the VMs with system clocks. In CloudLab, although the ConnectX-3 NIC supports hardware timestamping, we use software timestamps provided by the system clocks so that we can compare a public cloud and a bare-metal cloud by making the CloudLab setup as close as possible to a bare-metal cloud.

5.1.4 Selection of On-Ramp parameters

The EWMA gain g is set to $1/16$ (§3.2). The threshold T should be higher than the minimal OWD on the path, plus some headroom to tolerate errors in OWD measurements.

In Google Cloud, (i) as reported in [23], the VM-VM minimal RTT for TCP traffic is typically 25 μs, so the minimal OWD is less than that; (ii) the inaccuracy of Tx and Rx timestamps can be around 50 μs due to GSO in VM on Tx side and the merging of Rx segments in the hypervisor; (iii) the high percentile clock sync inaccuracy under Huygens is less than 3 μs. Taken together, we pick $T = 150\mu s$ to be safe.

In Cloudlab, following similar steps, we pick $T = 50\mu s$, because the minimal OWD is smaller, and there is no VM hypervisor involved in Cloudlab evaluations.

In ns-3, the minimal OWD is up to 6 μs inside the network. It emulates a NIC implementation of On-Ramp so the timestamps are accurate. To be safe, we pick $T = 6 + 10 = 16\mu s$.

§7.4 shows that the end-to-end performance of On-Ramp is only mildly sensitive to the value of T and g.

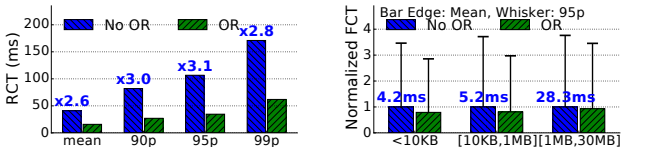
5.2 On-Ramp Performance

5.2.1 Google Cloud Platform (GCP)

We first consider the **following basic scenario** in GCP: *WebSearch* traffic at 40% load, and an incast load at 2% with a fanout of 40 where each of the 40 flows is 2KB (on each server, the average interval between two consecutive incast requests is 3.2 ms), and CUBIC congestion control.

Incast RCT. As can be seen in Figure 9a, the mean, 90th, 95th and 99th percentile RCTs of an incast request are reduced by 2.6×, 3.0×, 3.1× and 2.8×, respectively.

Background traffic FCT. We group background traffic flows into three buckets by size: small (≤10KB), medium (10KB–1MB), and large (1MB–30MB) flows. On-Ramp improves the mean FCT of the *WebSearch* background traffic flows by 21%, 19%, and 7% compared to the baseline for the flows of small, medium and large sizes respectively. The 95th percentile FCT shows similar improvements. This means that time-critical short flows get lower FCTs and the throughput of long flows does not degrade. Thus, On-Ramp *does not adversely affect* the congestion control of the long flows; indeed, it even improves it mildly by reducing packet drops and timeouts, so the long flows avoid unnecessary window cuts in CUBIC algorithm. See Figure 9b.[6] Finally, the percentage of packets retransmitted reduces from 0.0693% to 0.0314%.



(a) Incast RCT

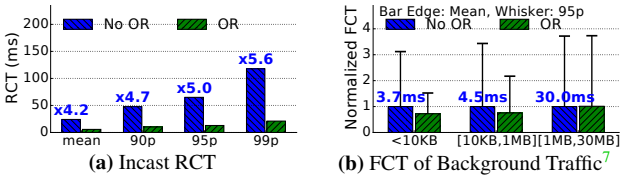
(b) FCT of Background Traffic, normalized by mean FCT without OR[7]

Figure 9: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast at 2% load (fanout=40, size of each flow=2KB).

[6] Note that, for this and all following experiments in GCP and CloudLab, the baseline "No OR" already includes the benefits of per-flow queueing and round-robin dequeueing, because the default QDisc (FQ_CoDel [9] for CUBIC and FQ [25] for BBR) in Ubuntu 18.04 provides this functionality. Therefore, the performance gain from "no OR" to "OR" purely comes from the On-Ramp algorithm.

Varying congestion control algorithms. We generalize the basic scenario by swapping out CUBIC for BBR [19]. BBR is an out-of-box alternative to CUBIC in VMs of public clouds. Figure 10a shows the mean and tail RCTs of an incast request are reduced by On-Ramp by 4.2× – 5.6×. On-Ramp also reduces the mean FCT of the *WebSearch* background traffic flows by 28% and 25% for the small and medium sizes respectively, and tail FCT by 51% and 37%, while maintaining the same performance as the baseline for the large flows, as shown in Figure 10b. In this scenario, the fraction of packets retransmitted reduces by 21×, from 0.0105% to 0.0005%.

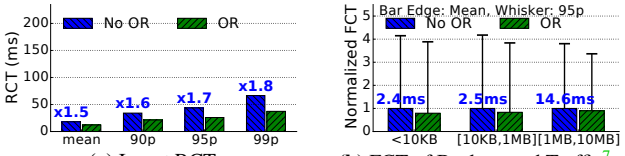
Note that BBR gives smaller RCTs of incast requests and FCTs of short flows than CUBIC because it controls delays. On-Ramp is able to further improve BBR's performance under the incast traffic.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 10: Cloud VM, **BBR**, *WebSearch* at 40% load + Incast at 2% load (fanout=40, size of each flow=2KB).

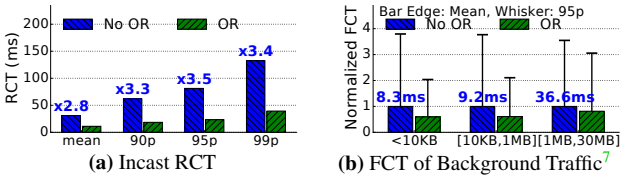
Varying background traffic pattern. Next, we consider the *FB_Hadoop* traffic at 40% load with an incast load as above and the CUBIC algorithm. Figure 11 shows the results.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 11: Cloud VM, CUBIC, **FB_Hadoop** at 40% load + Incast at 2% load (fanout=40, size of each flow= 2KB).

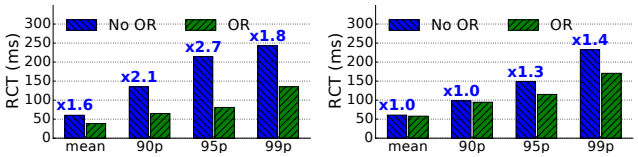
Varying the load level of background traffic. To test the robustness of On-Ramp under high load, we increase the background traffic to 80% load in the basic scenario, leaving everything else fixed. The results are in Figure 12. The performance gains achieved by On-Ramp are larger under higher loads. Figure 25 in Appendix shows the results at 60% load.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 12: Cloud VM, CUBIC, *WebSearch* at **80% load** + Incast at 2% load (fanout=40, size of each flow 2KB).

Varying the pattern and load level of incast traffic. Finally, in order to understand the effect of On-Ramp on RCT of larger incast requests (modeling storage-type traffic), we increased the size of incast flows from 2KB to 500KB in the basic scenario at incast loads of 20% and 2%. The findings are shown in Figure 13. Also, the total number of timeouts is reduced by 21× and 13× in the cases of 20% and 2% incast load respectively.



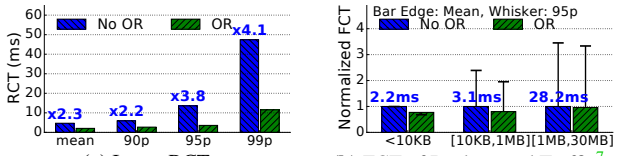
(a) Incast RCT (20% Incast Load) **(b)** Incast RCT (2% Incast Load)

Figure 13: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast (fanout=40, **size of each flow 500KB**).

This scenario highlights an interesting aspect of On-Ramp, namely, that the potential one RTT delay in obtaining OWD measurement when a new flow starts can be avoided by using the OWD measurements from previous flows. Specifically, under 2% load, the average interval between two consecutive incast requests is 800 ms, which is which is about one order of magnitude higher than the average RCT for requests of size 20MB (500KB × 40). Therefore, the gap between two request responses is large, and each request starts by getting new OWD measurements. When the incast load is 20%, the average inter-request interval (80 ms) is comparable to the average RCT. Therefore, a new request is able to leverage the OWD measurement of the previous one and help On-Ramp detect and throttle the incast episodes.[8]

5.2.2 CloudLab

For the evaluation on CloudLab, we consider the basic scenario described in the GCP evaluation except that the *WebSearch* traffic is at 60% load. As can be seen in Figure 14a, we observe similar improvements as GCP: the mean and tail RCTs of an incast request improves by 2.3× – 4.1×. The mean FCTs of the *WebSearch* background traffic flows are improved by 23%, 20%, 4% across flows of small, medium and large sizes respectively, as shown in Figure 14b.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

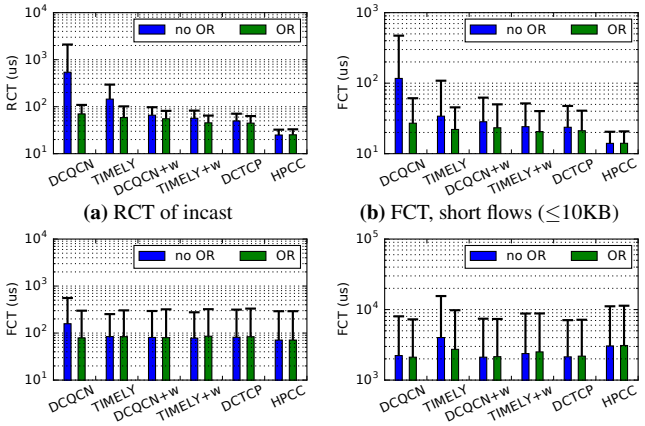
Figure 14: Bare-metal, CUBIC, *WebSearch* at 60% load + Incast at 2% load (fanout=40, size of each flow=2KB).

5.2.3 Large-scale ns-3 simulations

To understand the performance of On-Ramp when combined with recently developed congestion control schemes which use detailed network congestion information, we use ns-3 simulations. We use *WebSearch* traffic at 60% load, plus incast with a fanout of 40 and flow sizes of 2KB at 2% load. As

[7]Fig. 10b, 11b, 12b, 14b, 19b has the same normalization as Fig. 9b.

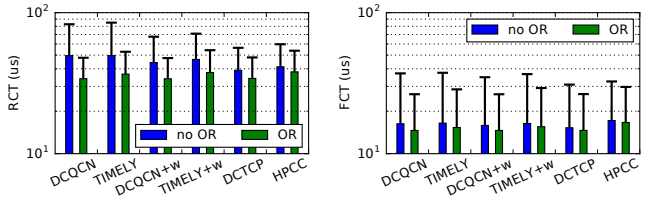
[8]Note that when the incast flow sizes are 2KB and the load is 2%, the average inter-request interval is 3.2 ms, 250 times smaller than when the flow sizes are 500KB. Therefore, in this case, an On-Ramp sender-side module *does* receive frequent-enough OWD measurements even at low load.



(a) RCT of incast **(b)** FCT, short flows (≤10KB)

(c) FCT, mid flows (10KB-1MB) **(d)** FCT, long flows (1-30MB)

Figure 15: ns-3, *WebSearch* of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*



(a) RCT of incast **(b)** FCT, GoogleSearchRPC flows

Figure 16: ns-3, **GoogleSearchRPC** of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*

a reminder, to mimic realistic deployments, PFC is in effect and each server's clock is jittered by an additional random offset according to a Gaussian distribution with a standard deviation of 200 ns, so OWD measurement is not precise. The congestion control algorithms used are DCQCN, TIMELY, DCTCP and HPCC. Following [44], a sending window is added to DCQCN and TIMELY to limit the bytes-in-flight. These algorithms are called DCQCN+w and TIMELY+w.

As seen in Figure 15, the mean and tail RCT of incast traffic is reduced significantly for DCQCN, TIMELY, DCQCN+w, TIMELY+w and DCTCP. The performance of HPCC is not significantly improved because it utilizes recent and detailed congestion information from the network elements and is already highly performant. Further, the FCT of the *WebSearch* flows is improved across all categories, including the large flows (1MB-30MB). Again, the extent of improvement is algorithm-specific.

Then, we change the background traffic to *FB_Hadoop*, leaving the other settings the same. The improvement given by On-Ramp is similar to the above. See Appendix Figure 26.

Next, we consider the *GoogleSearchRPC* workload. Since this traffic has mostly (>99.85%) small flows (≤10KB), we simply consider the mean and high percentile FCT across all flows rather than categorizing by flow size, as shown in Figure 16. This workload is challenging for most congestion control algorithms because nearly 80% of bytes are due to flows under 10 KB in size—too short for congestion control algorithms to address.[9] However, just by more efficiently controlling the transient events caused by the remaining 20% of bytes from the relatively long flows, On-Ramp improves the performance of all algorithms, including HPCC.

6 Evaluation in Facebook's Network

We have also evaluated On-Ramp at Facebook, where On-Ramp was used to throttle large, high-bandwidth storage file transfers so that they don't eat up all the switch buffers, causing severe packet drops for latency-sensitive compute application traffic. This scenario is canonical in data centers where multiple types of traffic with different objectives share the same network fabric. Our goal is to use On-Ramp to ensure storage traffic gets the bandwidth it needs while not affecting the latency of the compute applications.

Environment. Two racks inside a Facebook production cluster are used in the evaluation. As a typical setup, in the first rack, 15 machines work as application clients and 12 work as storage clients. In the second rack, 30 machines work as application servers and 6 work as storage servers. The two ToR switches are connected to 3 spine switches. The link bandwidth is 100 Gbps for each storage server, 25 Gbps for all other machines, and 100 Gbps between each ToR and each spine switch. Huygens runs on all machines to synchronize their clocks.

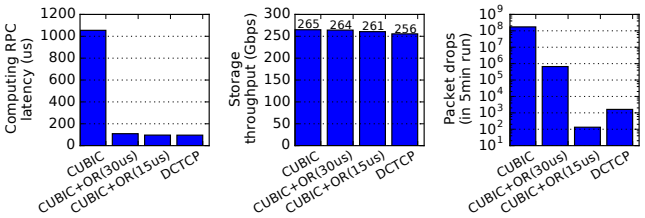
Traffic loads. Two types of traffic are run simultaneously: (1) Computing traffic: They are latency-sensitive short flows carrying RPCs generated by computing jobs. They run between the application clients and servers. (2) Storage traffic: Each pair of storage clients read files from one storage server via NVMe-over-TCP [11], which generates throughput-sensitive long flows consisting of 16–128KB bursts. When 12 or more storage clients are reading, the total amount of storage traffic requested will be $25 \times 12 = 300$ Gbps or more, enough to saturate the uplinks from the second ToR switch to the spine switches. Severe congestion happens at this load.

Results. Figure 17 shows the results when 12 storage clients are reading from storage servers.[10] When using the default CC CUBIC, the latency of computing RPCs is severely hurt by the storage traffic, and numerous packets are dropped. Deploying On-Ramp with CUBIC reduces the latency of computing RPCs by 10× while maintaining the throughput of storage traffic. The number of packets dropped is reduced by about 260×. Here we pick On-Ramp threshold T=30μs following the guideline in §3.2. The more aggressive setting of T=15μs reduces the packet drops even more while only marginally reducing the storage throughput, as shown in the figure.

On-Ramp's performance was compared with DCTCP as well, and we found that, under saturation loading of storage traffic (300 Gbps), both achieve a similar performance in

[9] DCQCN, TIMELY and HPCC have no slow start phase. Following [44], DCTCP's slow start phase is removed for fair comparisons.

[10] See Appendix §10.3 for results with 0–12 storage clients.



(a) Latency **(b)** Throughput **(c)** Packet drops

Figure 17: On-Ramp's performance in a Facebook cluster

terms of compute application latency,[11] but DCTCP achieves a slightly lower storage throughput compared to On-Ramp + CUBIC. A drawback of DCTCP is that it needs ECN marking at all the switches which is not only operationally burdensome, but challenging when many non-DCTCP flows (e.g., flows whose source or destination is an external server) share switches with the DCTCP flows. Being purely edge-based, On-Ramp sidesteps these burdens and challenges.

7 On-Ramp Deep Dive

7.1 The Accuracy of OWD Signals

To study the effect of the accuracy of OWD signals on On-Ramp's end-to-end performance, we repeat the ns-3 scenario described in §5.2.3 and consider DCQCN, TIMELY and HPCC. We add a constant random Gaussian offset to each clock of standard deviation σ_{clk}, and vary σ_{clk} to model different levels of clock inaccuracy.

Recall that $0.2\mu s$ is the default σ_{clk} we use throughout our ns-3 evaluation, with a corresponding threshold $T = 16\mu s$. Here, we increase σ_{clk} up to $100\mu s$ and, following the guidelines in §3.2, we change the threshold T according to the formula $T = 2\sigma_{clk} + minOWD$ ($minOWD = 6\mu s$ in ns-3). Essentially, inaccurate clocks lead to inaccurate measurements of OWD which become confounded with path congestion. Choosing a value of T as per the formula above allows for inaccurate clocks. In practice, Huygens reports an estimation of clock inaccuracy via the network effect [30], so we can adapt the threshold T according to it using this formula.

Figure 18 shows the RCT of incast requests under different σ_{clk}. We observe that for DCQCN and TIMELY, as σ_{clk} increases, the mean and tail incast RCT also increases. The performance degradation becomes more significant when σ_{clk} becomes comparable to the OWDs under congestion (roughly $20-100\mu s$). Since HPCC maintains very small queues (due to its bandwidth headroom), it operates well under the threshold T of queuing delay needed to trigger On-Ramp. Hence, On-Ramp doesn't affect its performance.

7.2 The Granularity of Control

As discussed in §4.1, GSO affects the granularity of control by On-Ramp. We study its effect by reducing the max GSO size from the default value of 64 KB to 16 KB. As shown in Figure 19, the mean and 90th, 95th, 99th percentile of incast traffic RCT are *further* reduced by 36%, 37%, 41%, 54%, respectively. The FCT of short and mid-sized flows in the *WebSearch* traffic are further reduced by 8-14% (mean) and 13-22% (95th percentile). The throughput of long flows is well-maintained. However, reducing max GSO size adds more CPU overhead to the sender, so we let the user decide on it.

Remark. This experiment explains the significant performance improvement of On-Ramp in ns-3 when compared to the Google Cloud and CloudLab implementations. In ns-3, we are effectively simulating a NIC implementation where On-Ramp has per-packet control and highly accurate time synchronization, which leads to better performance.

[11] Indeed, both DCTCP and On-Ramp + CUBIC achieve an RPC latency under saturation loading nearly equal to the case when there is no storage traffic, which is the best possible. See Appendix §10.3 for details.

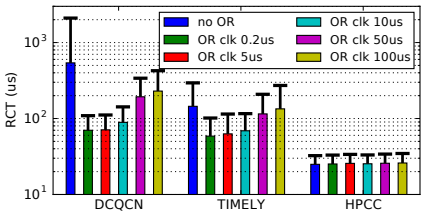
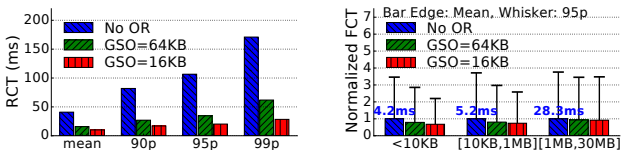


Figure 18: RCT of incast traffic under different levels of clock inaccuracy. Bars: mean, whiskers: 95th percentile.



(a) Incast RCT **(b)** FCT of Background Traffic[7]

Figure 19: Cloud VM, CUBIC, *WebSearch* at 40% load + Incast at 2% load (fanout=40, each flow 2KB).

7.3 Co-existence

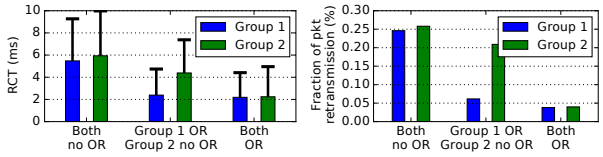
A public cloud user is unaware of other users and the amount of traffic (not controlled by On-Ramp) they insert into the network. Hence, it is possible that the effect of On-Ramp can be blunted when other traffic is present. In fact, if non-On-Ramp traffic shares links with On-Ramp traffic, the latter may unilaterally do worse because On-Ramp may pause transmission when congestion due to the non-On-Ramp traffic increases. The results in §5.2.1 show that this dire situation may not happen: a cloud user can achieve better performance by enabling On-Ramp in their own VM cluster even though there *may* be non-On-Ramp traffic on their paths. In this section, we revisit this question in the controlled environment of CloudLab.

We consider the scenario in §5.2.2 and divide the 100 servers in CloudLab randomly into two groups with 50 servers each. The same workload as in §5.2.2 is run inside each group, but we don't run cross-group traffic. This models 2 users renting servers in a cloud environment unbeknownst to each other. We evaluate the performance in the following cases: (i) both groups do not use On-Ramp, (ii) Group 1 uses On-Ramp but not Group 2, and (iii) both groups use On-Ramp.

Figure 20 summarizes the results. Interestingly, when Group 1 uses On-Ramp and Group 2 does not, *both groups* do

better than when neither uses On-Ramp. Essentially, during periods of congestion, On-Ramp enables Group 1 senders to transmit their traffic at moments when Group 2 traffic is at low load. Conversely, the improvement in Group 2's performance is due to a reduction in overall congestion.

When Group 2 also uses On-Ramp, the improvement in Group 1's performance is only slight. Thus, Group 1 obtains almost the same benefit from using On-Ramp whether or not Group 2 uses it; this is desirable for incremental deployment.

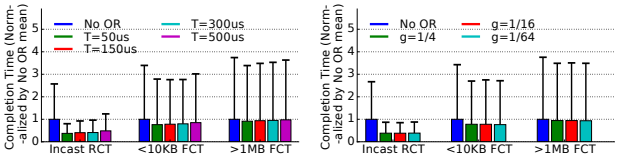


(a) RCT of incast **(b)** Pkt retransmission

Figure 20: Co-existence of traffic with and without OR.

7.4 On-Ramp Parameters T and g

To explore the sensitivity of On-Ramp on the threshold T and EWMA gain g, we run the same basic evaluation scenario described in 5.2.1 in GCP, but now vary T between 50μs and 500μs, and g between $\frac{1}{4}$ and $\frac{1}{64}$. Recall that for GCP, the default parameter values are $T = 150\mu$s and $g = \frac{1}{16}$. Figure 21a shows that the performance of On-Ramp worsens noticeably as T increases beyond 300 μs. Figure 21b shows that On-Ramp's performance is relatively insensitive to the value of g in the range between $\frac{1}{4}$ and $\frac{1}{64}$.



(a) Threshold T **(b)** EWMA gain g

Figure 21: Changing T and g. RCT and FCT normalized by the mean values of No OR. Bars: mean, whiskers: 95th percentile.

8 Related Work

Congestion control (CC). CC algorithms can be broadly categorized into two groups: (i) those which need no network assistance, e.g., drop-based schemes (TCP NewReno [28], CUBIC [33]) or delay-based schemes (TCP Vegas [18], TIMELY [47], and Swift [42]); and (ii) those which rely on network assistance, e.g., use ECN signals (DCTCP [13], DCQCN [53]), leverage in-band network telemetry (HPCC [44]), or rely on the network's ability to perform some functions like scheduling or trimming packets [16,17,29,31,34,47,49]. On-Ramp is complementary to CC. It is meant to be deployed underneath any CC algorithm, providing a fast and accurate response to transient congestion purely from the edge of the network. Swift [42] is a recent algorithm that uses RTT measurements and carefully chosen delay targets with support for fractional congestion windows to obtain good performance across a wide range of deployment scenarios. Swift couples the handling of equilibrium (on ACK) and transient (on timeout). This coupling is likely to lead to the equilibrium-transient tension mentioned in §2. By contrast, On-Ramp explicitly decouples the two, providing more robust performance.

In-network pause. Schemes such as Priority-based Flow Control (PFC) [38] have been widely deployed to eliminate packet drops in switches. However, PFC causes several safety and performance challenges including PFC deadlocks and congestion spreading [32,36,48,53]. By pausing flows at the edge, On-Ramp avoids these challenges.

Congestion Control (CC) in cloud environments. Previous works like AC/DC TCP [35] and Virtualized Congestion Control (VCC) [22] give cloud admins control over the CC of the users' VMs by translating between the target CC and the VM's CC. Their architectures are similar to On-Ramp: they also operate as a shim layer between the VM applications and the physical network, intercepting packets without requiring network infrastructure changes. However both AC/DC TCP and VCC rely on ECN support from the network infrastructure to implement the target CC (DCTCP) and need to be implemented by the cloud provider within the hypervisor. On-Ramp makes no assumptions of the underlying network infrastructure and can be implemented by cloud users within their VMs.

Flow scheduling in data centers. On-Ramp performs a form of flow scheduling because it pauses packets at the edge for a short period of time. Previous work in this space like pHost [29], NDP [34], and Homa [49] propose algorithms with varying levels of network support to enable flow scheduling within the network and improve end-to-end performance. On-Ramp requires no network support and is done purely at the edge. It can therefore be readily deployed, especially by users of public cloud environments.

9 Conclusion and Future Work

Datacenter packet transport over the last decade has relied increasingly on network support (e.g., ECN marking, queue size information), making it hard to deploy in environments such as the public cloud. We show empirically that the move towards increasingly rich network support is rooted in a tension between equilibrium and transience performance. Motivated by these results, we take a step back and modularize congestion control into two separate components, one responsible for equilibrium and the other for transients. We leave equilibrium handling to existing congestion control algorithms and design a new underlay scheme, On-Ramp, for transient handling. On-Ramp uses one-way delay measurements enabled by synchronized clocks to hold back packets transmitted by any congestion control algorithm at the edge of a network during transient congestion. Intellectually, On-Ramp contains two ideas that are of independent interest. First is the use of synchronized clocks to improve network performance. Second is the factoring of datacenter congestion control—traditionally a single control loop—into two separate control loops, one each for transience and equilibrium. We hope this paper is the beginning of a more in-depth investigation of both ideas.

Acknowledgments

We thank Vaibhavkumar Patel and Manoj Wadekar at Facebook for their tremendous work and support in conducting the Facebook evaluations. We thank the sponsors of the Platform Lab at Stanford University for generously supporting this research. Ahmad Ghalayini is supported by the Caroline and Fabian Pease Stanford Graduate Fellowship.

References

[1] PTP hardware clock infrastructure for Linux. https://www.kernel.org/doc/Documentation/ptp/ptp.txt, 2011. [Online; accessed 16-Sept-2020].

[2] TCP small queues. https://lwn.net/Articles/507065/, 2012. [Online; accessed 16-Sept-2020].

[3] Mellanox BlueField SmartNIC 25Gb/s Dual Port Ethernet Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf, 2019. [Online; accessed 16-Sept-2020].

[4] GitHub - alibaba-edu/High-Precision-Congestion-Control. https://github.com/alibaba-edu/High-Precision-Congestion-Control, 2020. [Online; accessed 25-January-2020].

[5] HomaSimulation/Google_SearchRPC.txt at omnet_simulations - PlatformLab/HomaSimulation - GitHub. https://github.com/PlatformLab/HomaSimulation/blob/omnet_simulations/RpcTransportDesign/OMNeT%2B%2BSimulation/homatransport/sizeDistributions/Google_SearchRPC.txt, 2020. [Online; accessed 16-Sept-2020].

[6] Machine types | Compute Engine Documentation | Google Cloud. https://cloud.google.com/compute/docs/machine-types, 2020. [Online; accessed 25-January-2020].

[7] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/, 2020. [Online; accessed 16-Sept-2020].

[8] ns-3 Network Simulator. https://www.nsnam.org/, 2020. [Online; accessed 16-Sept-2020].

[9] tc-fq_codel. http://man7.org/linux/man-pages/man8/tc-fq_codel.8.html, 2020. [Online; accessed 16-Sept-2020].

[10] The Cloud Lab Manual. Chapter 11: Hardware. http://docs.cloudlab.us/hardware.html, 2020. [Online; accessed 16-Sept-2020].

[11] NVMe-oF Specification. https://nvmexpress.org/developers/nvme-of-specification/, 2021. [Online; accessed 08-Feb-2021].

[12] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[13] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[14] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: Stability, Convergence, and Fairness. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 73–84, New York, NY, USA, 2011. ACM.

[15] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability Analysis of QCN: The Averaging Principle. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, page 49–60, New York, NY, USA, 2011. ACM.

[16] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. ACM.

[17] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 455–468, USA, 2015. USENIX Association.

[18] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM.

[19] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control. *Commun. ACM*, 60(2):58–66, January 2017.

[20] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187, 2016.

[21] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 239–252, 2017.

[22] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 230–243, New York, NY, USA, 2016. ACM.

[23] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, and et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 373–387, USA, 2018. USENIX Association.

[24] Nandita Dukkipati, Masayoshi Kobayashi, Rui Zhang-Shen, and Nick McKeown. Processor Sharing Flows in the Internet. In *Proceedings of the 13th International Conference on Quality of Service*, IWQoS'05, page 271–285, Berlin, Heidelberg, 2005. Springer-Verlag.

[25] Eric Dumazet. pkt_sched: fq: Fair Queue packet scheduler, 2013. [Online; accessed 25-June-2020].

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, and et al. The Design and Operation of Cloudlab. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 1–14, USA, 2019. USENIX Association.

[27] S. Floyd. RFC3649: HighSpeed TCP for Large Congestion Windows, 2003.

[28] S. Floyd and T. Henderson. RFC2582: The NewReno Modification to TCP's Fast Recovery Algorithm, 1999.

[29] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1–12, New York, NY, USA, 2015. ACM.

[30] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 81–94, Berkeley, CA, USA, 2018. USENIX Association.

[31] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 1–14, USA, 2015. USENIX Association.

[32] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. ACM.

[33] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[34] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 29–42, New York, NY, USA, 2017. ACM.

[35] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 244–257, New York, NY, USA, 2016. ACM.

[36] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 92–98, New York, NY, USA, 2016. ACM.

[37] IEEE. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)*, pages 1–300, July 2008.

[38] IEEE. IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 17: Priority-based Flow Control. *IEEE Std 802.1Qbb-2011*

(Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011), pages 1–40, Sep. 2011.

[39] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs. In *Proceedings of the 2019 ACM Symposium on SDN Research*, SOSR '19, page 8–20, New York, NY, USA, 2019. ACM.

[40] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. *SIGCOMM Comput. Commun. Rev.*, 32(4):89–102, August 2002.

[41] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *SIGCOMM '15*. ACM, 2015.

[42] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. ACM.

[43] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally Synchronized Time via Datacenter Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 454–467, New York, NY, USA, 2016. ACM.

[44] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 44–58, New York, NY, USA, 2019. ACM.

[45] Lisong Xu, K. Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524 vol.4, 2004.

[46] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, Oct 1991.

[47] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. ACM.

[48] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. ACM.

[49] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. ACM.

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[51] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. *SIGCOMM Comput. Commun. Rev.*, 39(4):303–314, August 2009.

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

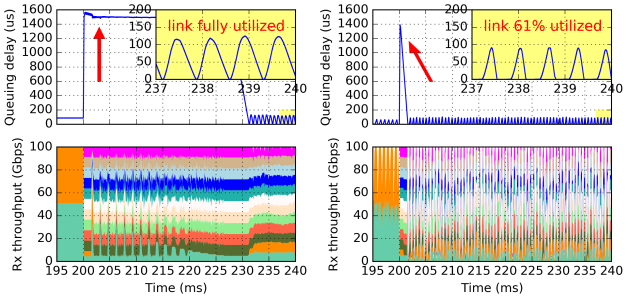
[53] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.

[54] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 313–327, New York, NY, USA, 2016. ACM.

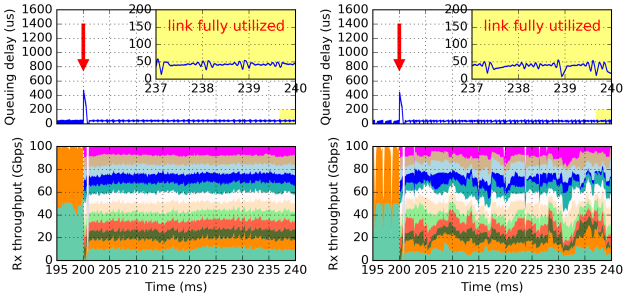
10 Appendix

10.1 Supplemental Material for §2 and §3

Figures 22 and 23 extend the results shown in Figures 1 and 2 of §2, by displaying the throughputs received by each flow as a stack. When the gain parameter is low (Figs. 22a and 23a), both TIMELY and DCQCN suffer from a long convergence time in transience, during which the flow throughputs are unstable and unfair. Under a high gain parameter, both algorithms under-utilize the link during equilibrium; see Figs. 22b and 23b. When On-Ramp is enabled, both CCs have shorter time in transience and smoother, fairer flow throughputs in equilibrium.



(a) β=0.2, no OR **(b)** β=0.8, no OR



(c) β=0.2, with OR **(d)** β=0.8, with OR

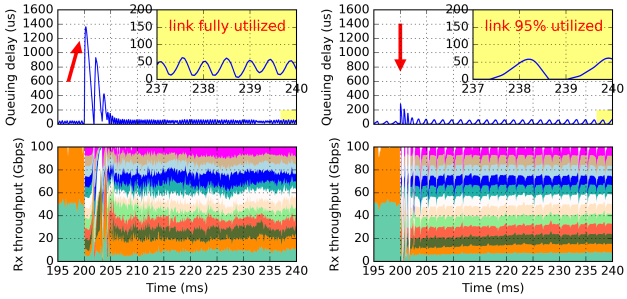
Figure 22: TIMELY study, the red arrow points to transience, and the yellow box is a zoom-in of equilibrium. OR T=30μs.

Figure 24 is a follow-up of Figure 5 in §3.2, which shows the OWDs and the flow rates when 12 CUBIC flows share a bottleneck link in a bare-metal environment (Cloudlab). The threshold T is also 50μs. When using strawman On-Ramp, similar to the case of 2 flows, the queue also suffers from significant undershooting. When using the final version of On-Ramp, most fluctuations in queue lengths are removed, and it achieves better fair sharing among all flows.

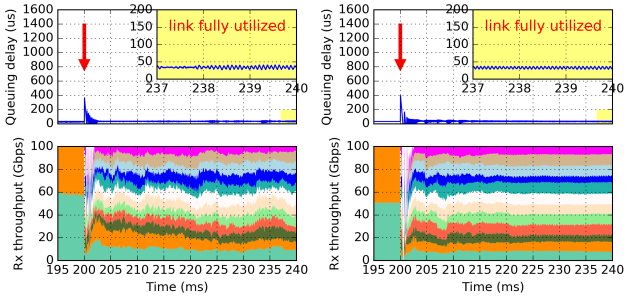
10.2 Supplemental Evaluation

Figure 25 corresponds to the evaluation scenario referred to in §5.2.1 of the main text, where the background *WebSearch* load is 60% for the base scenario in GCP.

Figure 26 corresponds to the ns-3 evaluation in §5.2.3. Here, we run FB_Hadoop traffic at 60% load, plus incast with

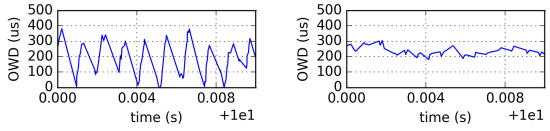


(a) T_i=55μs, T_d=50μs, no OR **(b)** T_i=300μs, T_d=4μs, no OR

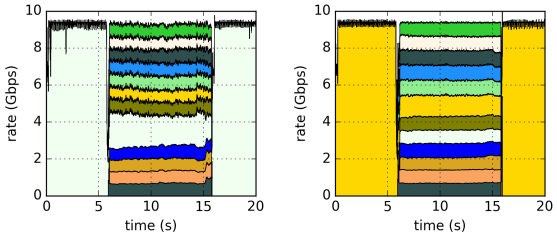


(c) T_i=55μs, T_d=50μs, with OR **(d)** T_i=300μs, T_d=4μs, with OR

Figure 23: DCQCN study. OR T=30μs.

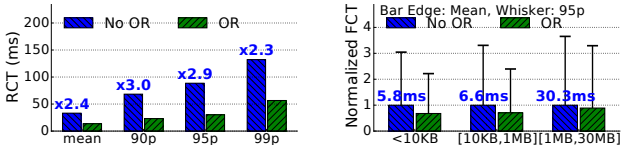


(a) OWD with Strawman OR **(b)** OWD with Final OR



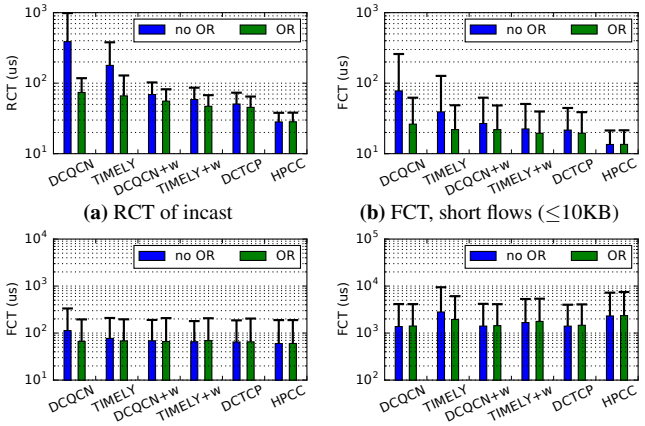
(c) Flow rates, Strawman OR **(d)** Flow rates, Final OR

Figure 24: 12 long-lived CUBIC flows sharing a link



(a) Incast RCT **(b)** FCT of Background Traffic, normalized by mean FCT without OR

Figure 25: Cloud VM, CUBIC, *WebSearch* at **60% load** + Incast at 2% load (fanout=40, each flow 2KB).



(a) RCT of incast **(b)** FCT, short flows ($\leq$10KB)

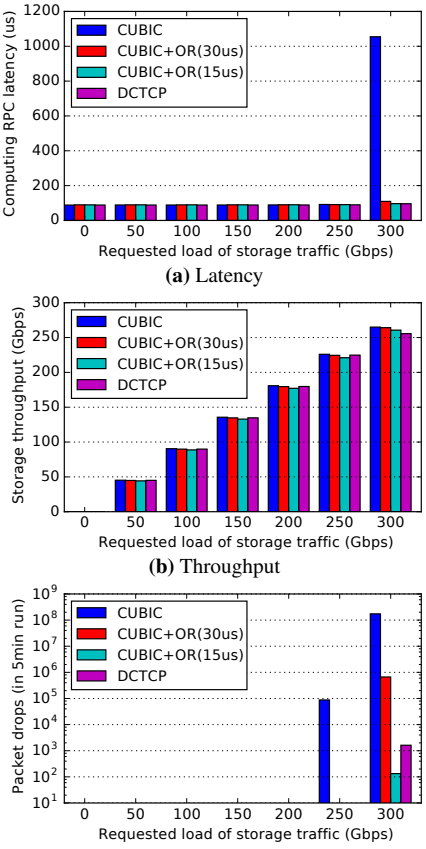
(c) FCT, mid flows (10KB-1MB) **(d)** FCT, long flows (1-30MB)

Figure 26: ns-3, **FBHadoop** of 60% load + incast of 2% load. Bars: mean, whiskers: 95th percentile. *Y-axis in log.*

a fanout of 40 and flow sizes of 2KB at 2% load. The findings are similar to the experiment with *WebSearch* workload (Figure 15). With On-Ramp, the RCT of incast requests and FCT of short flows in background traffic are significantly reduced, while the throughput of long flows is well-maintained (even improved e.g. in TIMELY). Again, the extent of improvement is algorithm-specific.

10.3 Supplemental Results: Facebook

Figure 27 corresponds to the evaluation in Facebook described in §6. The number of storage clients reading from storage servers is set to be 0, 2, 4, ..., 12, so the requested load of storage traffic is 0, 50, 100, ..., 300 Gbps respectively. When the requested load is less than or equal to 250 Gbps, the network is not congested yet, CUBIC, CUBIC + On-Ramp and DCTCP give similar performances. When the requested load hits 300 Gbps, the computing RPC latency shoots up dramatically under CUBIC, meaning it is severely hurt by the storage traffic. Using CUBIC + On-Ramp or DCTCP brings the latency down to the level similar to the non-congested case, while keeping the throughput of storage traffic well-maintained, as we discussed in §6.



(a) Latency

(b) Throughput

(c) Packet drops

Figure 27: On-Ramp's performance in a Facebook cluster