# Reducing Tail Latency for Multi-Bottleneck in Datacenter Networks: A Compound Approach

Yuxiang Zhang[a], Lin Cui[a,*], Fung Po Tso[b], Xiaolin Lei[a]

[a]*Department of Computer Science, Jinan University, Guangzhou, China*
[b]*Department of Computer Science, Loughborough University, UK*

**Abstract**

The effectiveness of network congestion control fundamentally depends on the accuracy and granularity of congestion feedback. In datacenter networks, precise feedback is essential for achieving high performance. Most existing approaches use either Explicit Congestion Notification (ECN) or network delay (e.g., RTT) independently as congestion indicators. However, in multi-bottleneck networks, the limitations of these signals become more pronounced: ECN struggles with large cumulative end-to-end latency, while RTT lacks the precision needed to control queuing delays at individual hops. To address these challenges, we propose *Cocktail*, a simple yet effective transport protocol for datacenter networks that combines both ECN and RTT congestion signals to more effectively handle multi-bottleneck scenarios. By leveraging the ECN signal, *Cocktail* bounds per-hop queue lengths, enhancing its ability to control single-hop latency and prevent packet loss. Additionally, by estimating RTT, *Cocktail* effectively manages end-to-end delay, resulting in lower Flow Completion Time (FCT). Extensive experimental evaluations in Mininet demonstrate that *Cocktail* significantly reduces the average and 99th-percentile completion times for small flows by up to 20% and 29%, respectively, compared to current practices under production workloads.

*Keywords:* ECN, RTT, Congestion control, Datacenter networks, Transport

## 1. Introduction

Datacenter networks run tightly coupled computing tasks that must be responsive to clients. This means up to thousands of backend servers need to exchange information just to serve a single request, and all of the transfers must be completed quickly enough to meet strict deadline [3][13][19]. To meet these requirements, congestion control mechanisms must simultaneously deliver high network throughput and utilization at low latency. However, when there are multiple network bottlenecks (multi-bottleneck), fulfilling these contradictory demands becomes much more difficult.

Multi-bottleneck is common in today's datacenters as their multi-root tree typologies with oversubscribed aggregation layer link capacities which are prone to creating congestion hotspots [10][25]. Such multi-bottleneck constrains overall performance since it creates congestion hotspots and significantly increases transfer latency. Moreover, it is challenging for end servers to detect the occurrence of multi-bottleneck due to lacking comprehensive congestion signals. Most current schemes rely on a single congestion signal, such as ECN or RTT, leading to performance degradation in multi-bottleneck scenarios.

The main drawback of ECN-based schemes is that they use a fixed base RTT value to calculate the marking threshold, without accounting for RTT variations. However, since queuing delay constitutes the largest part of RTT, significant RTT variations are common in multi-

*Corresponding author
*Email addresses:* samuelzyx0924@gmail.com (Yuxiang Zhang), tcuilin@jnu.edu.cn (Lin Cui), p.tso@lboro.ac.uk (Fung Po Tso), leixl@jnu.edu.cn (Xiaolin Lei)

bottleneck scenarios with varying numbers of bottlenecks. This means the end-to-end latency can increase rapidly as the number of bottlenecks increases, attributed to accumulative single bottleneck latency (refer to Figure 2).

Moreover, latency-based schemes, whether they use RTT or one-way delay, focus solely on end-to-end delay while neglecting the latency of individual hops. This oversight can lead to performance degradation in multi-bottleneck scenarios because the latency threshold is preset based on multi-hop conditions. For example, DX [15] sets a fixed threshold (headroom) for computing the congestion window using queue latency, operating under the assumption that overall latency is roughly the sum of delays at each switch along the transmission path. However, this mechanism does not account for whether queue latency arises from a single switch or multiple hops. Given the dynamic nature of traffic loads, the number of bottlenecks in a multi-bottleneck scenario can change; for instance, it might shift from five bottlenecks to a single one in the next second. Consequently, the total path delay may decrease while latency at a single congested switch increases. In such cases, long queues or even packet loss may occur at this congested switch, negatively impacting overall network performance (refer to Figure 3).

Motivating experiments are conducted in Section 2, which show that these two signals have their own deficiencies, negatively affecting overall performance in multi-bottleneck scenarios. The results demonstrate that ECN-based schemes cannot well control accumulative latency when the number of bottlenecks goes up. For example, the RTT of an ECN-based scheme can be about 25% higher than a Delay-based scheme due to queuing delay buildup (refer to Section 2). Conversely, relying solely on delay as a congestion signal fails to achieve acceptable queue lengths and loss ratios on individual switches in multi-bottleneck scenarios.

Several recent proposals combine ECN and delay signals to enhance performance. However, how to fully leverage the complementary strengths of the two signals to address the compounded performance issues in multi-bottleneck scenarios still needs to be investigated. For instance, EAR [25] and ECN# [26] share similar design guidelines, which adopt two thresholds in switches for ECN marking. When queue length or single hop sojourn time exceeds the respective threshold, ECN marking is triggered. Although these two schemes incorporate delay into ECN marking, they fail to solve multi-bottleneck accumulative latency problem, where end-to-end latency becomes unacceptable as the number of bottlenecks goes up. FFC [27] is another scheme which uses two signals to adjust the congestion window, expanding it if no ECN marks are detected and reducing it based on the estimated RTT otherwise. Therefore, FFC uses ECN solely to detect congestion rather than combining ECN and delay signals. As a result, it might encounter the accumulative latency problem caused by multi-bottleneck especially when queues of all switches are below the marking threshold.

Motivated by the above challenges, in this paper, we strive to provide a solution which can effectively improve end-to-end latency and mitigate performance degradation on a single switch with the presence of multi-bottleneck. To this end, we present *Cocktail*, a simple yet effective solution to achieve this goal. *Cocktail* detects network congestion by monitoring ECN marks and RTT variations. By leveraging ECN signals, *Cocktail* alleviates performance degradation due to high queue lengths on single hops. Meanwhile, to eliminate cumulative latency caused by multiple bottlenecks, *Cocktail* adjusts its congestion window based on RTT estimations, particularly decreasing the congestion window when an increase in RTT is detected. This is a conservative operation in order to provide low latency delivery.

In summary, the major contributions of this paper are threefold:

- We conducted extensive studies to show both RTT and ECN lack effectiveness and may degrade performance with multi-bottlenecks. Notably, ECN and RTT have their own disadvantages but their advantages are complementary.

- We present a new congestion control mechanism, *Cocktail*, which combines both ECN and RTT. *Cocktail* uses ECN signal to sense the congestion in a single hop and leverage the RTT signal to maximize the network utility and minimize the end-to-end delay.

- We evaluate the performance of *Cocktail* by using Mininet simulations. The results indicate that *Cocktail* can reduce the flow completion time by an average of up to 26.9% compared to state-of-art schemes.

The remainder of this paper is organized as follows. Section 2 introduces the background and related works as well as the motivation for proposing a combining ECN and RTT scheme. Section 3 presents our solution, *Cocktail*, which leverages the strengths of both ECN and RTT signals. Extensive experiments results are shown in Section 5 and demonstrate that *Cocktail*'s effectiveness. Section 7 concludes the paper and discusses potential directions for future work.

## 2. Background and Motivation

### *2.1. Background*

This section will provide a brief introduction to multi-bottleneck and introduce schemes based on different congestion signals.

#### *2.1.1. Multi-bottleneck*

A flow may traverse many hops to arrive at its destination as shown in Figure 1. A bottleneck can be formed when a certain switch on flow's transmission path is fully utilized like switches $S_1$ and $S_2$ in Figure 1. This bottleneck throttles the performance of the flow. When a flow encounters multiple bottlenecks along its path rather than just one, this scenario is called a multi-bottleneck. In datacenter, multi-bottleneck is very common due to the high volume transmission. Each switch may fully utilize its capacity to transmit packets, inevitably forming packet queues to handle a large amount of traffic traversing the switch. Consequently, many switches may experience this situation, resulting in flows being delivered under multi-bottleneck conditions.

In datacenters, end-to-end latency is predominantly influenced by queuing delays due to their low-latency fabric design. This means that multi-bottlenecks can significantly impact latency performance because multiple switches with long queues may exist along transmission paths. Additionally, detecting such multi-bottleneck scenarios is challenging for congestion control mechanisms. Currently, most schemes adopt either ECN or delay as the congestion signal which may only reflect if there is congestion occurring. However, it is difficult for senders to determine whether multiple bottlenecks exist on the transmission paths. So it is crucial to investigate the issue of multi-bottleneck and develop effective strategies to solve it.

#### *2.1.2. ECN-based schemes*

Significant progress has been made in developing ECN-based transport mechanisms for datacenter networks [3][4][5][29]. Most of these approaches rely on a similar structure comprising two components: ECN-aware rate control at the end host and ECN marking at the switch [3]. Specifically, an ECN marking threshold $K$ (i.e., standard ECN marking threshold) is given in a switch. When the buffer occupancy at a switch exceeds $K$, an ECN mark is applied to passing packets. At the end hosts, the sending rate for each connection is adjusted based on the proportion of ECN-marked packets, allowing the buffer occupancy to remain near the marking threshold without sacrificing throughput.

In particular, ECN-based schemes aim to constrain the maximum queue length at a single hop to around the marking threshold of $K$ packets. As a result, flows may experience a latency equivalent to $K$ packets when traversing a fully utilized switch[1]. However, in multi-bottleneck scenarios, the total queuing delay can cumulatively increase with the number of bottlenecks. For instance, the whole queuing delay could reach $nK$ when a flow is traversing $n$ bottlenecks path, which violates the low latency requirement for datacenter network transmission.

#### *2.1.3. Delay-based schemes*

Delay based transports use latency measured at the hosts for sending rate control without touching switches in the network. Recent advances in NIC hardware enable RTT measurements with microsecond-level accuracy, which is sufficient to estimate switch queuing in datacenter networks. Motivated by this observation, several delay-based proposals have been proposed which treat the delay measured at end hosts as the congestion signal, like TIMELY [17] using RTT and DX [15] using one-way delay.

Different from ECN-based schemes, these two mechanisms focus on the delay of the transmission path, neglecting the latency control on a single switch. Take DX as an example. DX uses threshold (headroom) to estimate

[1] For the ease and generality of description, we use the number of packets as queuing delay, i.e., representing the latency by the quantity of packets traversing a switch, rather than specific time interval [3].

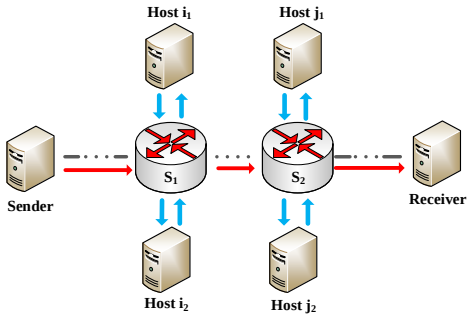



Figure 1: Multi-bottleneck scenario

if there is congestion in the networks, so it would increase its congestion window if the measured delay is less than threshold [15]. However, it can not sense whether the latency is generated from single or multiple switches. If the delay is caused by one switch with a long queue, a flow can easily encounter packet drops, degrading the performance.

For example, considering a path with five switches, if DX can tolerate 5 packets latency at each hop, the headroom can be set as 25 packets. However, the overall latency may come from certain switches (bottleneck). If other hops are being underutilized (i.e., no queues built up), the queuing delay of this switch queue can reach 25 packets. This significantly deviates from the original assumption that each bottleneck would experience up to 5 packets of latency. Furthermore, this phenomenon can be more severe in larger networks since the headroom might be set higher, potentially leading to packet loss at this single bottleneck.

### 2.2. Problem Exploration

To comprehensively validate the above analysis, we conducted a set of Mininet-based experiments to explore the deficiencies of schemes based on both signals. A multi-bottleneck topology is emulated as depicted in Figure 1. In this experiment, each link is configured with 1 Gbps link capacity and 25 $\mu$s latency. The marking threshold for DCTCP is set to 20 packets [3]. Besides, each bottleneck (switch) is saturated by 4 long-lived background flows which use the same congestion control as test scheme. In addition, DCTCP and TIMELY are used as the ECN-based and delay-based schemes for comparison, respectively.

**End-to-end latency:** Although delay-based schemes excel at controlling end-to-end latency, ECN-based schemes often struggle in this regard. To illustrate this, we measured the RTTs and FCTs achieved by DCTCP and TIMELY, and the results are presented in Figure 2.

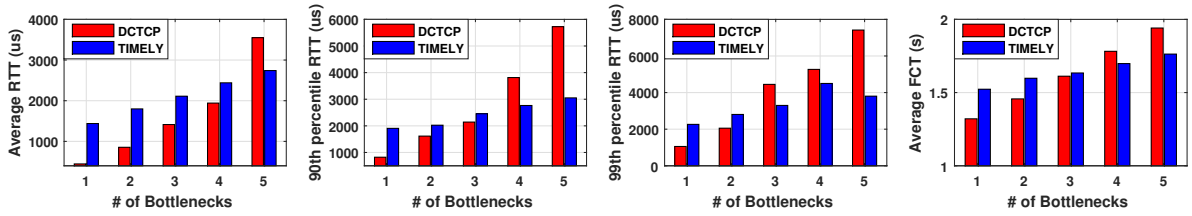
The results indicate that as the number of bottlenecks increases, the RTTs of both schemes also increase because flows traverse more bottlenecks and potentially encounter more congestion points and queuing (see Figure 2(a) ~ Figure 2(c)) Specifically, the RTTs of DCTCP increase more rapidly than that of TIMELY, though DCTCP gets better latency when the number of bottleneck is relatively small. As shown in Figure 2(d), TIMELY achieves lower FCT in the presence of multiple bottlenecks due to its effective control of end-to-end latency Since TIMELY is sensitive to RTT, it reacts to excessive queuing delay by reducing its sending rate, whereas DCTCP focuses on one hop delay and can not more effectively constrain end-to-end latency. This aligns with the conclusions presented above.

**Observation 1.** *In multi-bottleneck scenarios, ECN-based schemes have limited ability to control accumulative end-to-end latency.*

**One hop queuing:** ECN-based and delay-based schemes response differently to queuing at a switch. To understand how queuing affects both types of schemes, we investigated the queuing behavior of a single switch.

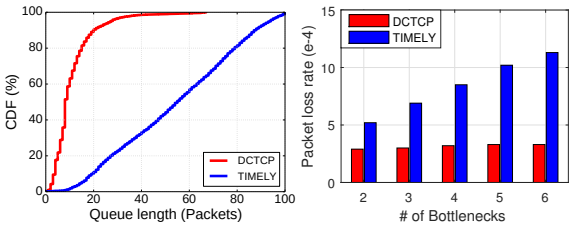
Figure 3 illustrates the queuing length at the switch connected to the sender switch and the packet loss rate of flows for both schemes. Clearly, TIMELY can have up to 4X more packet loss as the number of bottlenecks increases, as shown in Figure 3(b). This trend is further explained by the CDF of a single switch queue shown in Figure 3(a). Although TIMELY has a lower average queuing delay, the tail is significantly larger than DCTCP, leading to higher packet loss. This indicates that delay-based schemes cannot well control per-hop queuing, leading to substantial packet loss, especially in the multi-bottleneck scenario.

**Observation 2.** *Delay-based schemes are infeasible for tackling long queues or packet loss in multi-bottleneck scenarios with dynamics, e.g., the number of bottlenecks is changing.*




(a) Average RTTs (b) 90th percentile RTTs (c) 99th percentile RTTs (d) Average FCTs

Figure 2: End-to-end RTTs and FCTs in multi-bottleneck scenarios




(a) CDF of queue length (b) Packet loss rate

Figure 3: The comparison of sender switch queuing and loss rate in multi-bottleneck scenarios

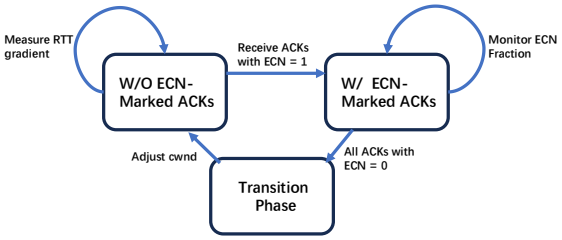



Figure 4: The state machine of *Corktail*

**Summary:** Single congestion signal cannot ensure acceptable end-to-end or single hop delays in multi-bottleneck scenario. Performance deteriorates more as the number of bottlenecks increases. These principles derived from our analysis have been summarized in Table 1.

## 3. Cocktail Design

*Cocktail* aims at utilizing the information provided by both ECN and RTT. Inspired by models used in DCTCP and TIMELY, *Cocktail* consists of three parts: retrieving RTT and the mechanism of sending ACKs, congestion condition identification, and the approaches to control sending rate. Figure 3 shows the design overview of *Cocktail* and notations used in this section are summarized in Table 2.

### *3.1. Design Overview*

Based on the analysis in Section 2, *Cocktail* aims to achieve optimal performance by maximizing throughput, minimizing latency, and approaching zero packet loss, even in a multi-bottleneck environment. The core objectives of *Cocktail* are summarized as follows:

- **High throughput:** *Cocktail* should operate with high efficiency, enabling data flows to fully utilize available network bandwidth whenever demand is sufficient.
- **Low latency:** To ensure minimal queuing delays and robust handling of traffic bursts, *Cocktail* must keep buffer occupancy low.
- **Near zero loss:** *Cocktail* should carefully manage queue lengths at switches to prevent packet buildup from exceeding switch capacity, thereby achieving minimal packet loss.

The previous section demonstrates that neither ECN nor delay signals alone can achieve both low packet loss and low RTTs, while they complement each other well. Thus, using both signals as congestion indicators is a promising approach. Therefore, *Cocktail* can use ECN signal to control per-hop queuing and thus packet loss while utilizing RTT signal to control end-to-end queuing and thus bounded packet round trip times.

Table 1: Path conditions using ECN and RTT

| ECN | RTT | Possible Cause | Operation |
|---|---|---|---|
| Exists | Increase | Heavy congested | Shrink |
| Exists | Decrease | Single hop congested | Shrink |
| Not exists | Increase | End-to-End congested | Shrink |
| Not exists | Decrease | Underutilized/No congestion | Expand |

Table 2: Notations in design description

| Notation | Description |
|---|---|
| $W_j$ | Congestion window at timeslot $j$ |
| $RTT_i$ | Round trip time at timeslot $i$ |
| $R_i$ | The sending rate at timeslot $i$ |
| $F_i$ | The fraction of ECN marked packets at timeslot $i$ |
| $K$ | The ECN marking threshold at switch |
| $Q_j$ | The measured queuing delay at timeslot $j$ |
| $\lambda$ | A self-updated coefficient parameter |
| $\alpha$ | A parameter to estimate marked packets |
| $\beta$ | A multiplicative decrement factor |
| $\sigma$ | The congestion window increase step |
| $RTT_{base}$ | The minimum RTT observed |
| $q_j$ | The queue length at timeslot $j$ |
| $C$ | The link capacity |
| $D$ | The transmission delay of single packet |
| $N$ | The number of flows in the network |
| $flag$ | Entry mark for entering transition phase |
| $T_{low}$ | Filter of RTT spikes |

Moreover, *Cocktail* is a window-based congestion control algorithm which launches a *Fast Start* phase to minimize FCTs of short flows. *Cocktail*'s congestion avoidance follows the popular Additive Increase Multiplicative Decrease (AIMD) rule. Particularly, *Cocktail* uses both ECN and RTT to make a decision on whether to increase or decrease the congestion window in the next round (roughly every RTT). If and only if there are marked packets, the sender reacts by reducing the window by a factor that depends on the fraction of marked packets. On the other hand, if there are no marked packets, the sender adjusts its window based on the RTT gradient. A positive delay gradient due to increasing RTTs indicates rising queues, which means that the sender should decrease the congestion window. On the other hand, a negative gradient indicates receding queues in the network, so the window size should be increased. Other features of TCP such as slow start, fast retransmission when getting three duplicate ACKs or fast recovery from packet loss are left unchanged.

**Algorithm 1** *Cocktail* Mechanism

**Input:** $new\_rtt$, $end\_frac$
**Output:** $new\_cwnd$

$new\_rtt\_diff = new\_rtt - prev\_rtt$
$prev\_rtt = new\_rtt$
$rtt\_diff = (1-\alpha) \cdot rtt\_diff + \alpha \cdot new\_rtt\_diff$
$rtt\_gradient = rtt\_diff / minRTT$
**if** $ecn\_frac > 0$ **then**
  $flag = 1$
  $\alpha \leftarrow (1-g) \times \alpha + g \times F$
  $cwnd \leftarrow cwnd \times (1 - \alpha/2)$
**else if** $flag == 1$ **then**
  $flag = 0$
  $Transition\_Phase()$
**else if** $new\_rtt < T_{low}$ **then**
  $cwnd \leftarrow cwnd + \sigma$
**else**
  $weight \leftarrow W(rtt\_gradient)$
  $error \leftarrow \frac{new\_rtt - RTT_{ref}}{RTT_{ref}}$
  $cwnd \leftarrow \theta(1 - weight) + cwnd(1 - \beta \cdot weight \cdot error)$
**end if**

### 3.2. Cocktail Algorithm

The *Cocktail* congestion control mechanism is detailed in Algorithm 1. Guided by the design objectives outlined above, the core principles of *Cocktail* can be summarized as below:

$$W_{j+1} = \begin{cases} W_j + \sigma, \text{ if } g \le 0, M = 0 \\ W_j \times d_1, \text{ if } g > 0, M = 0 \\ W_j \times d_2, \ M = 1 \end{cases} \quad (1)$$

where $g$ is the delay gradient, and $M$ represents ECN marks. $\alpha$ and $\beta$ are decreasing factors.

There are two conditions: $M$ equals 0 or 1, which means absence and presence of ECN-marked packets respectively. When there is no ECN-marked ACKs, *Cocktail* uses the RTT gradient to update the congestion window. If the gradient is negative or equals zero, *Cocktail* probes for more bandwidth by performing an additive increment for the connection: $W_{j+1} = W_j + \sigma$. When the gradient is positive, the total in-flight is greater than network capacity, *Cocktail* performs a multiplicative window decrement $d_1$, which relates to the RTT gradient (which is depicted as W/O ECN-Marked ACKs in Figure 3). Otherwise, when ECN-marked packets are present, the congestion window is shrunk by $d_2$, which relates to the fraction of ECN-marked packets. The well-known AIMD property ensures that our algorithm achieves fairness across connections [9][17] (which is depicted as W/ ECN-Marked ACKs in Figure 3). The detailed operations of these two conditions will be elaborated below.

**With ECN-marked ACK:** When the sender receives ECN-marked ACKs, it confirms that the queues at certain switches exceeds the marking threshold. In this case, *Cocktail* decreases the window proportional to the ECN fraction. Specifically, *Cocktail* follows the control law of DCTCP, where the sender accumulates the ECN fraction as a congestion measurement and adjusts the congestion window accordingly. In particular, the sender maintains an estimation of the fraction of packets that are marked, called $\alpha$, which is the weighted average of the fraction of marked packets [3][25]. In response to a marked ACK, the window is reduced as $W_{j+1} = W_j \times (1 - \alpha/2)$. This is how *Cocktail* maintains low queue length, while still ensuring high throughput.

**Without ECN-marked ACK:** If there is no marked ACK in the current congestion window, the sender employs gradient tracking and adjusts the congestion window using a smoothed delay gradient. To compute the delay gradient, *Cocktail* calculates the difference between two consecutive RTT samples, which is shown in line 1 ∼ 3 of Algorithm 1. *Cocktail* then normalizes this difference by dividing it by the minimum RTT, obtaining a dimensionless quantity [30]. This approach ensures that all flows have knowledge of the bottleneck's queue length, which provides two key benefits. First, the system can have a unique fixed point determined by the RTT (refer to Section 4). Second, all flows can converge to the same rate, since they share the knowledge of the bottleneck RTT. In practice, the exact value of the minimum RTT does not matter since the sender only needs to determine if the queue is growing or shrinking. Therefore, a fixed value is used to represent the wire propagation delay across the datacenter network, which is known ahead of time. A continuous weighting function $w(g)$ is then designed to make the transition between window increase and decrease smooth, avoiding the on-off behavior that causes oscillation [30]. Here, we simply use a linear function of $w_i$:

$$g_i = \begin{cases} 0, \ g_i \leq -\frac{1}{4} \\ 2g_i + \frac{1}{2}, -\frac{1}{4} < g_i < \frac{1}{4} \\ 1, \ g_i \geq \frac{1}{4} \end{cases} \tag{2}$$

Then, Equation 1 is transformed to Algorithm 1.

While we believe the above control mechanism is effective in each condition mentioned in Table 1, its performance during transitions between these conditions is uncertain. A special case needs to be considered: If ECN signals exist in $W_j$ while no ECN signals have been received in $W_{j+1}$, it suggests that the excessive queue lengths in all switches along this path have been eliminated. Since the $W_j$ must be a shrinking operation, the RTT gradient is likely negative in $W_{j+1}$. Hence, $W_{j+1}$ would equal to $W_j + i$. At this time, if $NW_j = CR$, in $W_{j+1}$ the queue length in certain switches along this path would exceed the marking threshold, resulting in fluctuation. Therefore, a new method is needed to address this reversed congestion window adjustment.

### *3.3. Transition Phase*

The transition phase is defined as follows: when the sender receives ECN signals in $W_j$ and gets no ECN signals in $W_{j+1}$, the computation of $W_{j+1}$ is defined as the transition phase. In this phase, *Cocktail* shrinks the congestion window in order to avoid fluctuation. Inspired by prior work [15][27], the following formula is adopted:

$$W_{j+1} = W_j \times (1 - \frac{Q_j}{\lambda}) \tag{3}$$

where $Q_j$ is the queuing delay at time $j$ and can be approximately computed as $RTT_j - RTT_{base}$.

The decrease should be just enough to prevent fluctuations without compromising utilization. An overly aggressive reduction of the congestion window can lead to under-utilization of network capacity. For *Cocktail*, the exact amount depends on the number of flows sharing the path, as the total in-flight packets should decrease to avoid exceeding the marking threshold. Coefficient $\lambda$ is used to account for the number of competing flows. The value of $\lambda$ is derived below.

Assume that at time $j$, the queue lengths of all bottleneck switches are equal to the marking threshold. Let $W_j^{(k)}$ and $Q_j^k$ denote the congestion window and the queuing delay of flow $k$ at time $j$, respectively. Therefore, the sum of the window size which shares the same path equals the bandwidth-delay product $C \cdot RTT_{base}$:

$$\sum_{k=1}^{N} W_j{}^{(k)} = C \cdot RTT_{base} \tag{4}$$

Since none of the $N$ flows experiences congestion, they all increase their window by $\sigma$ at time $j+1$:

$$\sum_{k=1}^{N} W_{j+1}{}^{(k)} = C \cdot RTT_{base} + N\sigma \tag{5}$$

At time $j+1$, the combined congestion windows of all flows cause the queue length to exceed the marking threshold. Assume that $Q_{j+1}^{(k)}$ is the part of the queuing delay that the $k^{th}$ flow has experienced due to the excess queue length beyond the marking threshold.. Thus, the total number of packets in the network at time $j+2$ can be calculated from the sum of the congestion window of all flows.

$$\sum_{k=1}^{N} W_{j+2}{}^{(k)} = \sum_{k=1}^{N} W_{j+1}{}^{(k)} \times (1 - \frac{Q_{j+1}^k}{\lambda}) \tag{6}$$

Assuming every flow experiences maximum queuing delay $N \cdot D$, where $D$ is the transmission delay of a single packet. Then, we get:

$$\begin{aligned} \sum_{k=1}^{N} W_{j+2}{}^{(k)} &= \sum_{k=1}^{N} W_{j+1}{}^{(k)} \times (1 - \frac{N \cdot D}{\lambda}) \\ &= (C \cdot RTT_{base} + N) \times (1 - \frac{N \cdot D}{\lambda}) \end{aligned} \tag{7}$$

The total number of in-flight packets at time $j+2$ should be equal to the bandwidth delay product:

$$(C \cdot RTT_{base} + N) \times (1 - \frac{N \cdot D}{\lambda}) = C \cdot RTT_{base} \tag{8}$$

Since $D = 1/C$, solving for $\lambda$ from Equation 8 results in:

$$\lambda = \frac{N \cdot D}{1 - \frac{C \cdot RTT_{base}}{C \cdot RTT_{base} + N}} \tag{9}$$

Among the variables required to calculate $\lambda$, the only unknown is $N$, which is the number of concurrent flows. The $N$ can be estimated according to the current congestion window since *Cocktail* achieves fair-share at a steady state. Then Equation 5 can be rewritten as:

$$\begin{aligned} \sum_{k=1}^{N} W_{j+1}{}^{(k)} &= N \times W_{j+1}^{(k)} = C \cdot R_{base} + N \\ \iff N &= \frac{C \cdot R_{base}}{W_{j+1}^{(k)} - 1} \end{aligned} \tag{10}$$

Using Equation 9 and replacing $D$, single packet transmission delay, with $1/C$, we get:

$$\lambda = \frac{R_{base} \cdot W_{j+1}^{(k)}}{W_{j+1}^{(k)} - i} \tag{11}$$

To compute $\lambda$, the sender only needs to know the base RTT $R_{base}$, and the previous congestion window. No additional measurement is required. Unlike purely ECN-based solutions, calculating $\lambda$ does not need to rely on external configuration or parameter settings. Moreover, variations in link capacity across the network do not affect the calculation of $\lambda$

### *3.4. Fast Start*

In current datacenter transports, many approaches transmit short messages without regard for potential congestion [18][24]. Hence, in *Cocktail*, new flows initiate by transmitting a sufficient amount of data (i.e., *Fast-byte*) to minimize the FCTs of short messages. The receiver subsequently returns explicit congestion information to manage future transmissions without introducing additional delays. Such blind transmissions mean that buffering can occur when multiple senders transmit to the

same receiver [18]. While some buffering is unavoidable to achieve low latency, it paradoxically contributes to increased latency when it occurs. Many previous designs have attempted to reduce buffering [3][4][5][15][17], but none of these approaches can completely eliminate the latency penalty of buffering. Thus, *Corktail* adopts the stringent scheme proposed above can help to reduce the buffering then control the transfer latency.

Furthermore, Fast start can be overly aggressive in the case when a large number of flows arrive in a burst. A new flow that that initiates a fast start may experience excessive packet loss. Without a quick loss recovery scheme, a new flow may end up with a worse flow completion time than if fast start had not been used at all. To address this, *Cocktail* enables per packet ACK at the receiver to quickly inform the sender which packets have been received to help the sender detect packet loss quicker. Retransmitting lost Fastbytes in a fast start way results in a very slow loss recovery procedure, as retransmitted packets may get lost again in the network. To prevent this, *Cocktail* retransmits these lost packets under congestion window control .

### *3.5. Parameter Settings*

**RTT low threshold $T_{low}$.** Since bursts may create transient queues in the network, resulting in RTT spikes during occasional collisions. Without careful handling, the sender might interpret an RTT spike as a positive gradient, mistakenly inferring congestion and backing off unnecessarily. To mitigate this issue, *Cocktail* employs a low threshold $T_{low}$ to filter RTT spikes and the RTT gradient is calculated on RTT samples which are larger than $T_{low}$ [15][17].

**Fast start threshold** $Fastbyte$**.** Blind transmissions can cause buffering when multiple senders transmit to the same receiver, potentially overwhelming switch capacity and filling up buffers. To mitigate this, *Cocktail* sets a threshold, Fastbyte, to limit such occurrences and control overall latency. In current *Cocktail*, the *fastbyte* is set as 10 packets since most short flows can be transferred within such setting, preventing other latency caused by mechanism control [18].

## 4. Analysis of *Cocktail*

This section presents an analysis of *Cocktail*. First, we prove that *Cocktail* has a unique fixed point determined by RTT. Next, we demostrate that all *Cocktail* flows can converge to the same rate. Finally, the queue length of a single hop remains below the maximum observed of DCTCP.

Particularly, this section focuses on analyzing the steady behavior of *Cocktail* which relates to the operation of line 11∼13 in Algorithm 1. The line 6∼7 and line 9 operation in Algorithm 1 is a monotonic function , which means the state machine remains in these operations only briefly. For example, when the sender receives an ECN-marked ACK, it performs the window reduction operation specified in in Algorithm 1 line 6 & 7. The sender then transitions to other operations in Algorithm 1, without reaching stasis in this monotonic decreasing function. The same applies to line 11 of Algorithm 1.

Thus, steady-state behavior can only be achieved through the operations in lines 11∼13 of Algorithm 1, as these include both window increase and decrease functions. The operations of these three lines can be formulated as follows:

$$\frac{\mathrm{d}W_{j+1}}{\mathrm{d}t} = \frac{1 - W_j\delta}{RTT_{base}} - \frac{W_j\beta g_i}{RTT_{base}} \cdot \frac{q(t - RTT_{base}) - q')}{q'} \tag{12}$$

where $W_i$, the weight of window decreasing, is a function of $g_i$, and must satisfy $0 \leq W_i(g_i) \leq 1$ for any $g_i$. For ease of description, we assume the feedback delay of congestion signals as $RTT_{base}$ [30]. Intuitively, $W_i(g_i)$ is monotonically increasing with $g_i$, because a larger RTT gradient should lead to a larger window decrease. $W_i(g_i)$ is an indicator function of $g_i$, i.e., $W_i(g_i) = 1$ when $g_i \geq 0$, and $W_i(g_i) = 0$ when $g_i < 0$.

### *4.1. Fixed Point*

We first obtain the fixed point of the Algorithm 1.

**Theorem 1.** *The Cocktail scheme described in Algorithm 1 has a unique fixed point. All flows have the same congestion window at this fixed point, and the queue length is* $q^* = \frac{q'\delta}{W_j\beta} + q'$.

*Proof.* By setting the left-hand side of Equation 12 to 0.

$$\frac{\mathrm{d}W_{j+1}}{\mathrm{d}t} = 0 \tag{13}$$

Then any fixed points of the *Cocktail* (if they exist) must satisfy:

$$\frac{1 - W_j\delta}{RTT_{base}} = \frac{W_j\beta g_i}{RTT_{base}} \cdot \frac{q(t - RTT_{base}) - q')}{q'} \tag{14}$$

At any of the fixed points, let's assume the queue length at the fixed points is $q^*$ and it is shared by all flows and determined by Equation 14:

$$q^* = \left[ \frac{(1 - W_j)\delta}{W_j \beta W_i} + 1 \right] \cdot q' \tag{15}$$

Since it is a fixed point, all flows should be stable and converge to a fair share. Thus, the gradient of all flows equals to 0, which means the $w_i = 0$. Combining with Equation 15, the equation can be rewritten as :

$$q^* = \frac{q'\delta}{W_j \beta} + q' \tag{16}$$

Thus, *Cocktail* has a unique fixed point which leads to a unique fixed point of queue length $q^*$. □

### 4.2. Convergence

In this section, we demonstrate that two *Cocktail* flows with the same RTT can converge to their fair shares.

**Theorem 2.** *Two Cocktail flows converge to fair share under the network model as shown at the beginning of this section with the same round trip delay.*

*Proof.* Jain's fairness index is used, which is defined as follows, $F(x) = \frac{(\sum x_i)^2}{n(\sum x_i^2)}$. Besides, as mentioned before, flows would converge to the fixed share of network capacity only when they stay in the steady phase. Otherwise, they may increase or decrease the congestion window, which violates the flows' convergence.

Let's consider two *Cocktail* flows, whose window sizes are $x_1$ and $x_2$, respectively (assuming $x_1 < x_2$). We see $\Delta F \geq 0$, if and only if $\Delta x_1/x_1 \geq \Delta x_2/x_2$. There are three conditions,

- If $g_i \leq 1/4$, then $w_i$ equals to 0. Then $\Delta x_1 = \Delta x_2 = \delta$, since the increment of congestion window is $\delta$. As assumption denotes $x_1 < x_2$, it is easy to see $1/x_1 > 1/x_2$. Thus, the formula becomes $\Delta x_1/x_1 > \Delta x_2/x_2$.

- If $g_i \geq 1/4$, then $w_i$ is set as 1. Thus, the formula can be rewritten as $\Delta x_1/x_1 = -\beta \cdot error = \Delta x_2/x_2$.

- Last, if $g_i \in (-1/4, 1/4)$, then we get $\Delta x_1 = \delta(1/2 - 2g_i) - \beta \cdot error \cdot (2g_i + 1/2) \cdot x_1 > \delta(1/2 - 2g_i) - \beta \cdot error \cdot (2g_i + 1/2)x_2 = \Delta x_2$. $\therefore \Delta x_1/x_1 > \Delta x_2/x_2$. When the two streams share the same bottlenecks and the propagation delay is the same, the gradient observed by the two streams should also converge to the same value. So the fairness index keeps increasing.

In summary, in each case, the index is strictly non-decreasing. Therefore, eventually, the two *Cocktail* flows converge to their fair share. □

### 4.3. Queue Length

Next, the worst case of queue length, in which the accumulative queue length arrives at the upper bound.

**Theorem 3.** *The maximum queue length has an upper bound: $nK + N$.*

*Proof.* We consider $N$ long-lived flows with identical round-trip times RTT, sharing bottleneck link of capacity $C$. We assume that the $N$ flows are synchronized, which is an assumption that is only realistic when $N$ is small where is the case *Cocktail* primarily addresses in datacenters. Under these conditions, the expected maximum queue length of single hop $Q_{max}$ equals $K$.

Hence, the queue lengths of all switches reach $K$ at time $T_i$ and the sum of latency of this path arrives at $nK$. Worse still, at $T_{i+1}$, all flows increase their congestion window by $i$. In this case, the latency arrives at $nK + N$. However, there are two conditions when the congestion window enforces adding $i$ when the overall latency in the network equals $nK$. First, the previous operation on the congestion window was a decrease, then at $T_{i+1}$, the congestion window is reduced. This makes it impossible for the worst-case scenario to occur. Another condition is adding $i$ to their congestion window as time $T_i$. Then, the RTTs of all flows have undoubtedly increased. Consequently, their congestion window would not increase at time $T_{i+1}$. In summary, the worst case cannot happen and the overall queue length does not reach $nK + N$. □

## 5. Implementation and Evaluation

In this section, we first present some implementation details. Then, Mininet based experiments are conducted to answer the following three key questions:

- **How does *Cocktail* perform in multi-bottleneck scenario?** In a static experiment, the results show that *Cocktail* reduces RTTs, maintains a low loss ratio, and ensures fairness between flows with long and short RTTs.
- **Does *Cocktail* scale to large datacenter fabric?** Using large-scale Mininet simulations, we show that *Cocktail* scales to multi-hop topologies and delivers the best overall performance. For example, it reduces the average FCT for large flows by up to 44.48% compared to TIMELY, while achieving up to 11% lower 99th percentile FCT for small flows compared to the DCTCP. Large-scale Mininet simulations demonstrate that Cocktail scales effectively to multi-hop topologies, delivering outstanding overall performance. For instance, it reduces the average Flow Completion Time (FCT) for large flows by up to 24.5% compared to TIMELY and achieves up to an 24.7% reduction in 99th-percentile FCT for small flows compared to DCTCP.
- **How robust is *Cocktail* to parameter settings?** Using a series of targeted simulations, we show that *Cocktail* is robust to $Fastbyte$ settings.

### *5.1. Implementation*

We have implemented *Cocktail* on the Linux kernel 4.10 by plugging a new congestion control module.

*Cocktail* relies on ECN-enabled switches to generate congestion notifications, a feature widely supported in modern datacenters [7][8]. Accurate RTT measurement is also essential for *Cocktail*, requiring servers equipped with high-resolution timers (up to the microsecond level) to support high-speed, low-latency datacenter networks. Fortunately, the option of a high-resolution timer has been provided in the Linux kernel 2.6 and later versions. Besides, one can install a NIC that supports the DPDK driver to use software NIC timestamp or directly equip hardware timestamp to obtain more accurate measurement [15][17][27].

To eliminate host processing delay, we perform TX timestamping just before sending packets to the NIC and RX timestamping immediately after packets are received, with the DPDK-based device driver [15]. We utilize *rdtsc* to capture CPU cycles and convert them into nanosecond precision. The memory overhead is proportional to the arrived data of which the corresponding ACK has not been sent yet. The memory overhead is negligible as it requires storing 8 bytes per in-flight packet.

The majority of *Cocktail*s congestion control logic is implemented in the function that processes ACK packets. Each ACK packet carries pre-calculated RTT information in its header, provided by the DPDK-based device driver. For each congestion window round, the received delay samples are averaged and used to update the congestion window.

### *5.2. Experiment Setup*

**Network Topology:** We employ a widely used fat-tree topology with 10 pods, in which there are 250 servers, 25 core switches, 50 aggregation switches, and 50 edge switches [27]. Each link is configured with a bandwidth of 40 Gbps and a latency of 10 $\mu$s. The network load is varied from 10% to 90% to evaluate performance across different conditions.

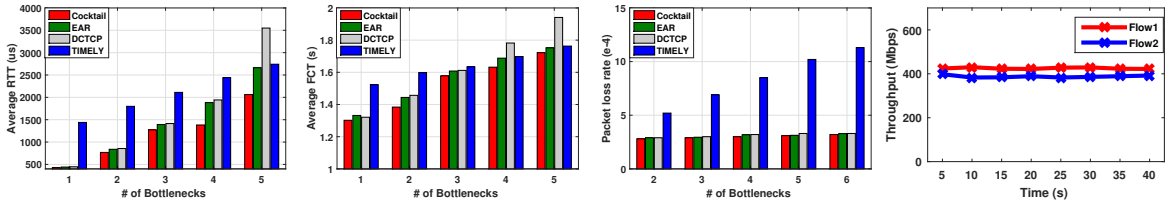
**Benchmark Workload:** We use four traffic distributions based on measurements from production datacenter networks, as summarized in Table 3 a web search workload [3], a data mining workload [12], a web server workload [20] and a cache follower workload [20]. In general, both workloads are heavy-tailed. Overall, these workloads exhibit heavy-tailed characteristics. For instance, in the web search workload, approximately 60% of all bytes come from flows smaller than 10MB, resulting in frequent concurrent flows on the same link, which increases network contention. The data mining workload, by contrast, primarily consists of tiny and large flows [15]; around 80% of flows are smaller than 10KB, while approximately 95% of the data volume is from large flows.

**Transports Compared:** We evaluate the performance of five transports, *Cocktail*, DCTCP, TIMELY, EAR and HPCC. In addition, ECMP is used for routing and load balancing.

**Performance metric:** Unless stated otherwise, FCT serves as the primary performance metric. We evaluate both the overall average FCT across all flows and the average FCT for flows of different sizes (small, medium, and large). To assess tail latency, the 99th-percentile FCT for small flows is presented in the figures. All results reflect the average of five runs.

Table 3: Flow size distribution of realistic workload

| | Web Server | Cache Follower | Web Search | Data Mining |
|---|---|---|---|---|
| 0 - 100KB | 81% | 53% | 52% | 83% |
| 100KB - 1MB | 19% | 18% | 28% | 8% |
| > 1MB | 0% | 29% | 20% | 9% |
| Average flow size | 64KB | 701KB | 1.6MB | 7.41MB |




(a) Average RTTs (b) Average FCTs (c) Average packet loss ratio (d) Fairness of *Cocktail*

Figure 5: Static experiments: (a) Average RTTs; (b) Average FCTs; (c) Average packet loss ratio; (d) Fairness of *Cocktail*.

### *5.3. Static Flow Experiment*

We begin with a basic static flow experiment to evaluate whether *Cocktail* can address the challenges outlined in Section 2. By repeating the experiments from Section 2 with the addition of *Cocktail*, we observe that *Cocktail* effectively tackles multi-bottleneck issues.

Figure 5(a) shows that *Cocktail* achieves low RTTs comparable to those of DCTCP, TIMELY and EAR. Specially, *Cocktail* can smooth the increment of RTT when the number of hops goes up, indicating its effectiveness in reducing end-to-end latency. Figure 5(b) demonstrates that *Cocktail* can reduce FCT by an average of 11.2%. Thanks to its efficiency in minimizing RTT, *Cocktail* can outperform DCTCP and TIMELY in FCT performance and shows similar improvements to EAR. Additionally, Figure 5 confirms that EAR struggles with cumulative latency in multi-bottleneck scenarios, as previously observed.

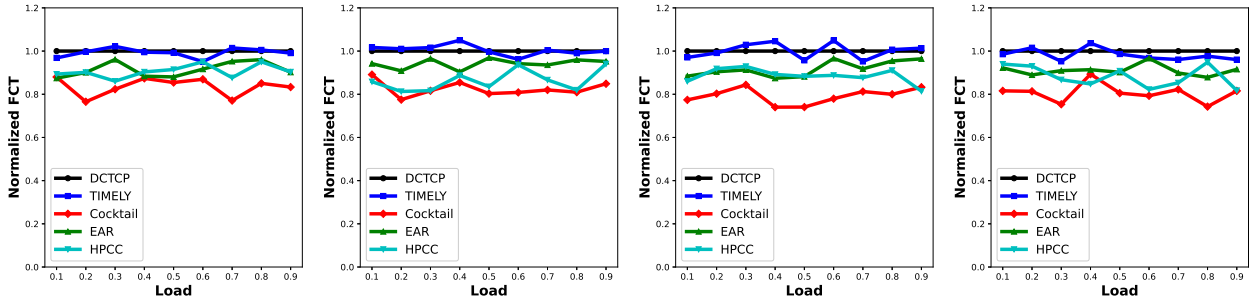
Further, Figure 5(c) presents the loss ratio achieved by *Cocktail*, which achieves a comparable loss ratio compared to DCTCP and EAR while getting better performance than TIMELY. Lastly, to assess throughput fairness, two *Cocktail* flows were generated to coexist. Figure 5(d) shows that the throughput difference between two flows is relatively small, which means *Cocktail* can offer fair performance to each *Cocktail* flow.

These results demonstrate that *Cocktail* performs well in multi-bottleneck scenarios and achieves the objectives set out in Section 3.

### *5.4. Large-scale Simulations*

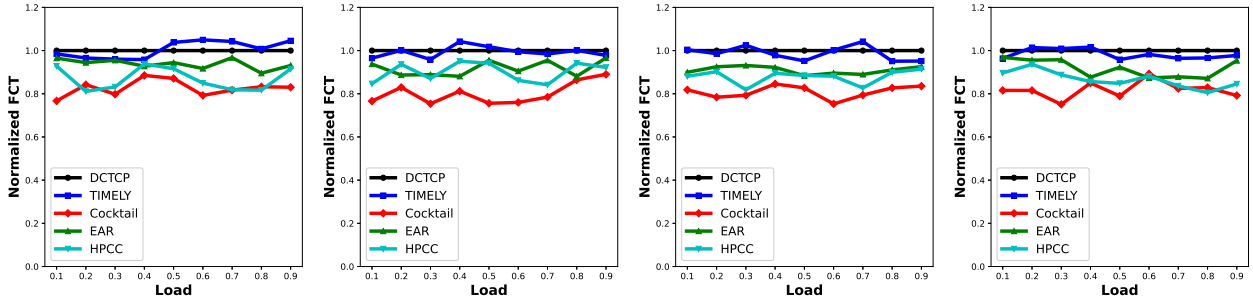
In this section, we evaluate the performance of *Cocktail*, DCTCP, TIMELY, EAR and HPCC through large-scale simulations. At the start of each simulation, the flow generator module randomly selects a sender and a receiver to initiate a new flow, with each flow generated using a Poisson process. Figure 5.2 to 5.2 present the FCT results across different workload and flow sizes. To better visualize the results, we normalize the FCT relative to that of DCTCP.

**Under the web search workload:** The web search workload mostly contains small and medium-sized flows from a few KB to tens of MB, with more than 95% of total bytes come from the flow smaller than 20MB [15]. From Figure 6(a) we observe that *Cocktail* generally achieves the best overall performance, consistent with our experiment results in Section 5.3. For the flows smaller than 100KB, *Cocktail* significantly reduces the average and the 99th percentile FCT, achieving average reductions of 14.33%, 12.15% and 7.6% FCT compared to DCTCP, TIMELY and EAR respectively.




(a) Overall (b) Small flow (<100KB) Avg (c) Small flow $99^{th}$ percentile (d) Large flow (>10MB) Avg

Figure 6: Web search workload: FCT across different flow sizes




(a) Overall (b) Small flow (<100KB) Avg (c) Small flow $99^{th}$ percentile (d) Large flow (>10MB) Avg

Figure 7: Data mining workload: FCT across different flow sizes

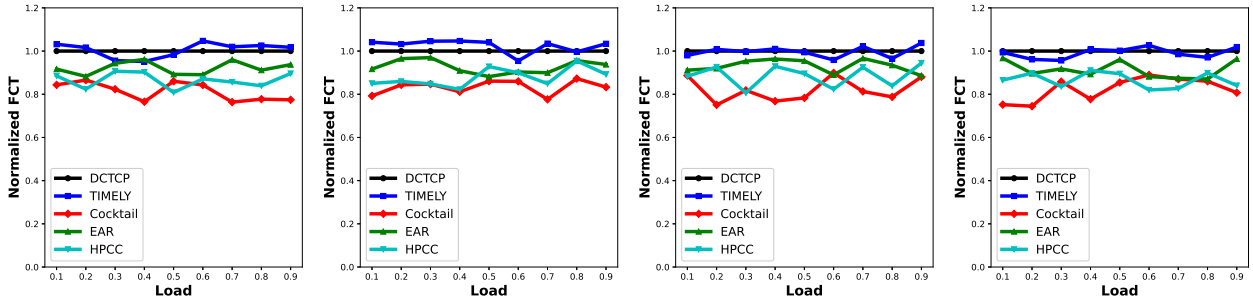
In a larger flow size group, *Cocktail* greatly outperforms other schemes. It achieves 20.24% lower average FCT compared to DCTCP and 19.34% lower than TIMELY. Furthermore, *Cocktail* obtains up to 16% improvement in FCT when compared to EAR.

Across all flow size, *Cocktail* can achieve achieves performance comparable to HPCC. Especially for small flow, *Cocktail* demonstrates its effectiveness by reducing FCT by up to 8.3%, and in most cases, *Cocktail* achieves lower FCTs.

**Under the data mining workload:** The data mining workload comprises both tiny and large-sized flows, ranging from hundreds of bytes to 1GB. The flow sizes are highly skewed: 80% of the flows are smaller than 10KB, while 95% of the total bytes come from flows larger than 30MB [15] The flow completion times of data mining workload are presented in Figure 5.2.

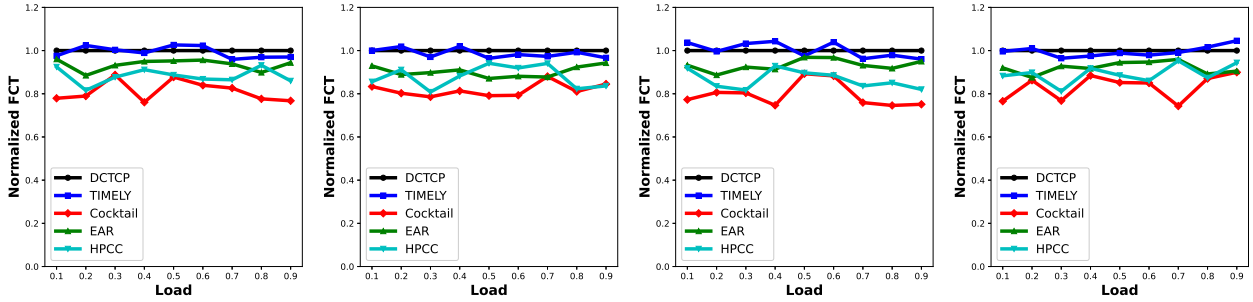
The performance improvement of *Cocktail* is more prominent for small flows in the data mining workload than in the search workload. From Figure 7(a), *Cocktail* generally achieve the best overall average FCT. Compared to DCTCP, *Cocktail* achieves up to 21% lower FCT. When compared to TIMELY, *Cocktail* outperforms it across all loads for the data mining workload. Moreover, *Cocktail* reduces FCT by up to 16.6% compared to EAR's FCT.

**Under the Hadoop workload:** The Hadoop workload mostly contains tiny flows. We demonstrate that *Cocktail* also achieves good performance under this workload. *Cocktail* attains comparable overall performance to HPCC while reducing FCT by up to 24.3% and 23.7% compared to DCTCP and TIMELY, respectively. Furthermore, compared to DCTCP, *Cocktail* reduces the average FCT for small flows by up to 20.8% and achieves about 25.9% improvement on FCT compared to TIMELY.




(a) Overall (b) Small flow (<100KB) Avg (c) Small flow $99^{th}$ percentile (d) Large flow (>10MB) Avg

Figure 8: Hadoop workload: FCT across different flow sizes

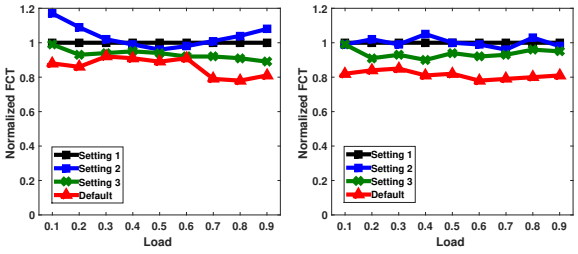



(a) Overall (b) Small flow (<100KB) Avg (c) Small flow $99^{th}$ percentile (d) Large flow (>10MB) Avg

Figure 9: Cache follower workload: FCT across different flow sizes

When it comes to large flows, *Cocktail* slightly outperforms other schemes.

**Under the cache follower workload:** In general, *Cocktail* achieves excellent overall performance among all schemes. In particular, *Cocktail* demonstrates comparable performance to HPCC and reduces the average Flow Completion Time (FCT) by about 18.5% compared to TIMELY. The most noticeable performance improvement for small flows is observed in the 99th percentile FCT at 40% load in Figure 5.2, where *Cocktail* outperforms DCTCP by up to 24.7% and reduces FCT by approximately 26.9% compared to TIMELY. For large flows, *Cocktail* generally outperforms other three mechanisms and achieves slightly better FCT than HPCC. Further, compared to DCTCP, *Cocktail* reduces the average FCT for large flows by up to 26.4%. *Cocktail* obtains about 24.5% better FCT as TIMELY's.




(a) Overall (b) Small flow (<100KB) Avg

Figure 10: FCT with different *Fastbyte*. FCT is normalized to the value achieved by the default setting.

### 5.5. Impact of Fastbyte

In this section, we evaluate *Cocktail*'s robustness to parameter settings through a series of targeted simulations.

We analyzed all workloads and average results are shown in Figure 10. Our objective is to highlight factors influencing threshold values, rather than to fine-tune parameters. The *Fastbyte* parameter is essential to network performance. Generally, the sender transmits $Fastbyte$ bytes blindly to cover the round trip time to the receiver and the receiver can return explicit congestion information (e.g., RTT and ECN) to help the sender control future transmissions. Four settings are configured to explore *Cocktail*'s sensitivity to $Fastbyte$, i.e., 5 packets (setting 1), 30 packets (setting 2), 20 packets (setting 3) and 10 packets (default setting). As Figure 10 illustrates, larger $Fastbyte$ can degrade the FCT performance. Since more packets get into the switches, leading to congestion at a single hop. The response from receivers may not arrive at the sender immediately, making congestion inevitable. Conversely, a smaller *Fastbyte* can also hinder performance, as it may underutilize network capacity. Therefore, in our simulations, setting *Fastbyte* to 10 packets achieves the best balance and performance.

## 6. Related Works

To provide acceptable network performance, many schemes combining two signals have been proposed, like EAR [25], ECN# [26], RTT_DCQCN [22] and FFC [27]. While these schemes reveal the ineffectiveness of either signal, their mechanism may have limitations on improving performance in multi-bottleneck scenarios.

**EAR & ECN#:** EAR and ECN# are twins which share similar design principles. So we only discuss EAR below and ECN# encounters similar problems. EAR is a congestion control for datacenter networks, which combines ECN and RTT signals. Specifically, the EAR senders treat ACKs satisfying one of the following conditions as congestion indicators:

$$queue > ecn\_threshold(K) \tag{17}$$

$$RTT > base\_RTT + delay\_threshold(T) \tag{18}$$

where *base_RTT* is the minimal RTT sample, $K$ and $T$ are the threshold for switch queue and RTT. The congestion window is computed in alignment with the original DCTCP, where senders aggregate congestion indicators over a set window to produce a congestion fraction as the measurement output. The subsequent congestion control intelligence module adjusts the congestion window based on the well-studied control laws of DCTCP , while standard TCP features, including slow start, fast retransmission on triple duplicate ACKs, and fast recovery from packet loss, remain unchanged. A recent method [22] combines delay and CNP feedback for data management, using delay solely to assess congestion upon receiving CNP feedback. However, this hybrid approach does not fully exploit the potential of delay signals.

**FFC:** FFC is a transport protocol that independently adjusts its congestion window based on two-dimensional congestion notifications from RTT and ECN signals. Specifically, the FFC sender processes each ACK to capture real-time RTT and ECN marks, reducing the congestion window only upon receiving ECN-marked ACKs. Additionally, FFC calculates the window reduction based on the measured RTTs of marked ACKs within the current measurement period. The congestion window is then computed as follows:

$$W_{j+1} = \begin{cases} W_j + 1, \text{ if no ACK is marked} \\ W_j \times \frac{R(E-\hat{q}))}{E^2} + \frac{\hat{q}R}{E^2}, \text{ otherwise} \end{cases} \tag{19}$$

where $E$ is the RTT when the queue length is just equal to the marking threshold, $\hat{q}$ is the queuing delay attributed to the part of queue length that exceeds the marking threshold and $R$ represents the minimal RTT.

EAR, in particular, employs two thresholds within a single hops queue to control queue length at each switch, akin to the mechanism used by DCTCP. Thus, EAR may encounter similarly accumulative large end-to-end latency in multi-bottleneck scenarios, as seen in DCTCP or other ECN-based schemes. Besides, though FFC continually estimates the RTT and ECN fraction, it actually recognizes the ECN as the congestion signal and only uses RTT for shrinking the congestion window when the network is congested. Similarly, RTT_DCQCN only uses delay as rate computation input when the sender receives CNP signals, which confines its sending rate adhering to ECN-based mechanism. These methods fall short of fully leveraging ECN and RTT signals in multi-bottleneck scenarios. On the contrary, *CorkTail* integrates both metrics to optimize network performance, with experiments confirming its effectiveness.

In recent year, Bolt [6], EQCIN [11], SWIFT [14],

HPCC [16], Poseidon [21] and ACC [28] introduced new congestion control mechanisms aimed at enhancing transmission performance. However, these schemes rely on in-network telemetry (INT) to accurately obtain congestion information, which poses challenges for large-scale deployment.

Recent research has made significant strides in enhancing TCP performance over high bandwidth-delay product paths, including TCP Wave [2] and BBRv2+ [23]. However, these schemes do not focus on maintaining small queue lengths, which is a critical requirement in datacenter environments [1].

## 7. Conclusion

In this paper, we present *Cocktail*, a simple yet effective congestion control that uses both ECN and RTT as congestion signals to simultaneously achieve high throughput and low latency simultaneously. Our evaluation results show that *Cocktail* significantly reduces flow completion times in multi-bottleneck scenarios, demonstrating that it is a practical solution that meets all our design objectives.

## References


[1] Abdelsalam, A., Luglio, M., Quadrini, M., Roseti, C., Zampognaro, F., 2019. Quic-proxy based architecture for satellite communication to enhance a 5g scenario. In: 2019 International Symposium on Networks, Computers and Communications (ISNCC). IEEE, pp. 1–6.

[2] Abdelsalam, A., Luglio, M., Roseti, C., Zampognaro, F., 2017. Tcp wave: a new reliable transport approach for future internet. Computer Networks 112, 122–143.

[3] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., Sridharan, M., 2010. Data center tcp (DCTCP). In: ACM SIGCOMM computer communication review. Vol. 40. ACM, pp. 63–74.

[4] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., Yasuda, M., 2012. Less is More: trading a little bandwidth for ultra-low latency in the data center. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, pp. 19–19.

[5] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., Shenker, S., 2013. pfabric: Minimal near-optimal datacenter transport. In: ACM SIGCOMM Computer Communication Review. Vol. 43. ACM, pp. 435–446.

[6] Arslan, S., Li, Y., Kumar, G., Dukkipati, N., Apr. 2023. Bolt: Sub-RTT congestion control for Ultra-Low latency. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, pp. 219–236.
URL `https://www.usenix.org/conference/nsdi23/presentation/arslan`

[7] Bai, W., Chen, K., Chen, L., Kim, C., Wu, H., 2016. Enabling ECN over generic packet scheduling. In: Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies. ACM, pp. 191–204.

[8] Bai, W., Chen, L., Chen, K., Wu, H., 2016. Enabling ECN in Multi-Service Multi-Queue Data Centers. In: NSDI. pp. 537–549.

[9] Chiu, D.-M., Jain, R., 1989. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. Computer Networks and ISDN systems 17 (1), 1–14.

[10] Cho, I., Jang, K., Han, D., 2017. Credit-scheduled delay-bounded congestion control for datacenters. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. ACM, pp. 239–252.

[11] Dong, P., Lu, X., Huang, T., Chen, L., Yang, Y., Zhang, L., 2024. Predictive queue-based rate control for low latency in lossless data center networks. IEEE Transactions on Network and Service Management.

[12] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., Sengupta, S., 2009. VL2: a scalable and flexible

data center network. In: ACM SIGCOMM computer communication review. Vol. 39. ACM, pp. 51–62.

[13] Grigorik, I., 2013. Optimizing the Critical Rendering Path. O'Reilly Velocity Conference, Santa Clara., CA.

[14] Kumar, G., Dukkipati, N., Jang, K., Wassel, H. M., Wu, X., Montazeri, B., Wang, Y., Springborn, K., Alfeld, C., Ryan, M., et al., 2020. 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. pp. 514–528.

[15] Lee, C., Park, C., Jang, K., Moon, S. B., Han, D., 2015. Accurate Latency-based Congestion Feedback for Datacenters. In: USENIX Annual Technical Conference. pp. 403–415.

[16] Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., et al., 2019. HPCC: High precision congestion control. In: Proceedings of the ACM Special Interest Group on Data Communication. pp. 44–58.

[17] Mittal, R., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., Zats, D., et al., 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In: ACM SIGCOMM Computer Communication Review. Vol. 45. ACM, pp. 537–550.

[18] Montazeri, B., Li, Y., Alizadeh, M., Ousterhout, J., 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. arXiv preprint arXiv:1803.09615.

[19] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al., 2013. Scaling memcache at facebook. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). pp. 385–398.

[20] Roy, A., Zeng, H., Bagga, J., Porter, G., Snoeren, A. C., 2015. Inside the social network's (datacenter) network. In: ACM SIGCOMM Computer Communication Review. Vol. 45. ACM, pp. 123–137.

[21] Wang, W., Moshref, M., Li, Y., Kumar, G., Ng, T. S. E., Cardwell, N., Dukkipati, N., Apr. 2023. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, pp. 255–274.
URL https://www.usenix.org/conference/nsdi23/presentation/wang-weitao

[22] Wang, Y., Lan, M., Zhao, T., Gao, Z., Xie, M., 2020. Combining rtt and ecn for rocev2 protocol. HPCCT & BDAI '20. Association for Computing Machinery, New York, NY, USA, p. 158164.
URL https://doi.org/10.1145/3409501.3409509

[23] Yang, F., Wu, Q., Li, Z., Liu, Y., Pau, G., Xie, G., 2022. Bbrv2+: Towards balancing aggressiveness and fairness with delay-based bandwidth probing. Computer Networks 206, 108789.

[24] Ye, J., Leung, K.-C., Li, V. O., Low, S. H., 2018. Combating bufferbloat in multi-bottleneck networks: Equilibrium, stability, and algorithms. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, pp. 648–656.

[25] Zeng, G., Bai, W., Chen, G., Chen, K., Han, D., Zhu, Y., 2017. Combining ECN and RTT for Datacenter Transport. In: Proceedings of the First Asia-Pacific Workshop on Networking. ACM, pp. 36–42.

[26] Zhang, J., Bai, W., Chen, K., 2019. Enabling ecn for datacenter networks with rtt variations. In: Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies. CoNEXT '19. Association for Computing Machinery, New York, NY, USA, pp. 233–245.

[27] Zhang, T., Huang, J., Wang, J., Chen, J., Pan, Y., Min, G., 2018. Designing Fast and Friendly TCP to Fit High Speed Data Center Networks. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). IEEE, pp. 43–53.

[28] Zhang, Y., Meng, Q., Hu, C., Ren, F., 2024. Revisiting congestion control for lossless ethernet. In: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). pp. 131–148.

[29] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., Zhang, M., 2015. Congestion control for large-scale RDMA deployments. In: ACM SIGCOMM Computer Communication Review. Vol. 45. ACM, pp. 523–536.

[30] Zhu, Y., Ghobadi, M., Misra, V., Padhye, J., 2016. 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. ACM, pp. 313–327.