He et al. Journal of Cloud Computing (2024) 13:72
https://doi.org/10.1186/s13677-024-00642-8

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Accurate and fast congestion feedback in MEC-enabled RDMA datacenters

Xin He[1], Feifan Liang[1], Weibei Fan[1], Junchang Wang[1], Lei Han[1], Fu Xiao[1*] and Wanchun Dou[2]

Abstract

Mobile edge computing (MEC) is a novel computing paradigm that pushes computation and storage resources to the edge of the network. The interconnection of edge servers forms small-scale data centers, enabling MEC to provide low-latency network services for mobile users. Nowadays, Remote Direct Memory Access (RDMA) has been widely deployed in such data centers to reduce CPU overhead and network latency. Plenty of congestion control mechanisms have been proposed for RDMA data centers, aiming to provide low-latency data delivery and high throughput network services. However, our fine-grained experimental analysis reveals that existing congestion control mechanisms still have performance limitations due to inappropriate congestion notifications and the long congestion feedback cycle. In this paper, we propose Mercury, which is an accurate and fast congestion feedback mechanism. Mercury comprises two key components: (1) the state-driven congestion detection and (2) the window-based congestion notification. Specifically, the state-driven congestion detection monitors the queue length and the number of packets received at the switch egress port when the PFC is triggered. It determines the states of egress ports and identifies flows that really contribute to congestion. Then, window-based congestion notification calculates the window sizes for congested flows and rapidly returns Congestion Notification Packets (CNPs) with the window information to the sender. It facilitates the rate adjustment of congested flows. Mercury is compatible with existing RDMA CC mechanisms and can be easily implemented in switches. We employ real-world data sets and conduct both micro-benchmark and large-scale simulations to evaluate the performance of Mercury. The results indicate that, thanks to the accurate and fast congestion feedback, Mercury achieves a reduction in the 99th tail flow completion time by up to 45.1%, 41.8%, 38.7%, 30.9%, and 37.9% compared with Timely, DCQCN, DCQCN+TCD, PACC, and HPCC, respectively.

Keywords Congestion feedback, Congestion detection, MEC-enabled RDMA datacenters

Introduction

Mobile edge computing (MEC) is a novel computing paradigm that pushes computation and storage resources to the edge of the network [1–3]. The interconnection of edge servers forms small-scale data centers. These data centers are required to support increasingly diverse applications, such as federated learning, data analysis, and parallel computing [4–6]. These applications have stringent performance requirements (e.g., high throughput and low latency [7, 8]), which places enormous pressure on the data center. The traditional TCP/IP stacks are no longer suitable since their high CPU overhead and long processing latency [9]. As a consequence, Remote Direct Memory Access (RDMA) which enables kernel-bypass and zero-copy data transport, has become an attractive option for MEC-enabled data centers [10, 11].

RDMA over Converged Ethernet v2 (RoCEv2) is the de-facto standard to deploy RDMA in the data center [10]. RoCEv2 is compatible with IP/Ethernet and

*Correspondence:
Fu Xiao
xiaof@njupt.edu.cn
[1] School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, Jiangsu, China
[2] Department of Computer Science and Technology, Nanjing University, Nanjing 210023, Jiangsu, China

adopts Priority Flow Control (PFC) to achieve lossless data transmission in RMDA data centers [12]. However, PFC is a coarse-grained flow control mechanism. To avoid packet drops, PFC roughly pauses/resumes the egress port of the switch, leading to several potential problems, e.g., Head-of-Line (HOL) blocking, PFC deadlock, and PAUSE storm [13, 14], which damage the performance of the network. Therefore, in recent years, the congestion control (CC) algorithms for RDMA data centers that can reduce the activation of PFC have attracted much attention.

The CC mechanisms for RDMA data centers can be broadly classified into two categories: (1) end-to-end CC mechanisms and (2) switch-driven CC mechanisms. The end-to-end CC mechanisms adopt different signals such as Explicit Congestion Notification (ECN) [10, 15], Round-Trip Time (RTT) [11, 16] and In-Network Telemetry (INT) [17] to mark in-network congestion. However, the control loop of end-to-end CC mechanisms is long. Sluggish congestion reaction aggravates the queue accumulation of congested switches, thus prolonging the flow completion time (FCT). Besides, as network bandwidth continues to increase (from 1Gbps to 40Gbps/100Gbps/400Gbps) [10], an increasing number of flows can complete their transmission within one RTT [18]. It is hard for the end-to-end CC mechanisms to control such flows effectively. The switch-driven CC mechanisms [19, 20] leverage the in-network switch to measure congestion closely and feedback congestion control information to the sender directly. As a consequence, the switch-driven CC provides fast congestion feedback by reducing the congestion loop. However, most of the switch-driven CCs rely on the Proportional Integral (PI) controller to calculate the congestion control information [19–21]. The PI controller needs well-tuned control parameters, thus bringing implementation challenges in modern programmable switches [22, 23].

The congestion detection is a cornerstone of the CC mechanisms, which determines when and where the CC mechanisms take effect [24, 25]. The existing RDMA CC mechanisms still have limitations in congestion detection. For example, the typical CC mechanism DCQCN [10] marks flows as congested flows when the switch queue accumulates and exceeds the specified threshold.However, such a congestion detection mechanism is inaccurate since the PFC also incurs queue accumulation.As shown in Fig. 1a, the PFC enables the downstream switch to pause the data transmission of the upstream switch when its queue length exceeds the PFC pause threshold X_{off}, and then the queue of the upstream switch builds up. Therefore, the existing RDMA CC mechanisms may mislabel flows as congested flows if the real congested flow and uncongested flows (i.e., victim flows) share a paused queue. Namely, the existing RDMA CC mechanisms cannot accurately determine which flow is the culprit of congestion. Although the newly proposed TCD [26] has attempted to distinguish the congested flows and victim flows, it determines the state of each flow after a preconfigured parameter $max(T_{on})$ expires, which cannot rapidly take effect in high-speed data center networks.

To solve the problem mentioned above, we present Mercury, which aims to provide accurate and fast congestion feedback to the sender. The sender receives the congestion feedback and rapidly adjusts its sending rate to eliminate the congestion. Mercury designs two key components to achieve our goals:

(1) State-driven congestion detection: Mercury leverages the switch, which is directly relative to in-network congestion, to detect the congestion closely. To identify which flows are the real culprits of congestion, Mercury first defines the states of the switch egress ports. Then, Mercury monitors both the queue length and the number of packets received at the egress port when the PFC is triggered. According to the above statistical information, Mercury determines whether the port is a congested port. Flows passing through the congested port are the ones that actually cause the congestion. With the state-driven congestion detection, Mercury identifies congested flows accurately and only feedbacks Congestion Notification Packets (CNPs) of the congested flows.

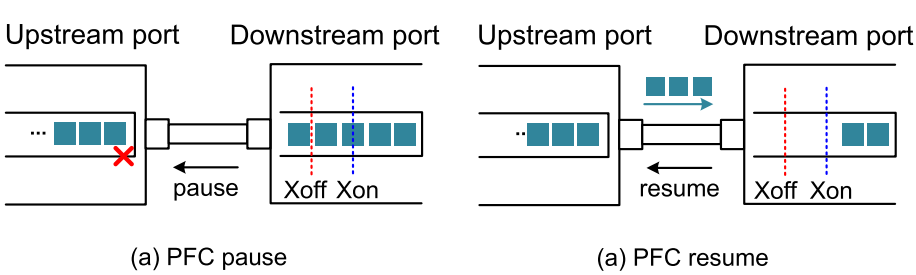


Fig. 1 PFC mechanism

(2) Window-based congestion notification: To rapidly adjust the sending rate of congested flows, the window-based congestion notification calculates the window sizes of each congested flow to limit the number of inflight packets. The window sizes can be carried in CNPs and sent back to the sender to limit the number of inflight packets. At the sender, the window information can be integrated with existing congestion control mechanisms to efficiently detect and feedback network congestion without any modifications in network hardware.

In summary, this paper has the following contributions:

- We reveal that existing RDMA CC mechanisms still have performance limitations due to inappropriate congestion notifications and the long congestion feedback through fine-grained experiments.
- We present Mercury, which develops the state-driven congestion detection to identify flows that really contribute to congestion. For the real congested flows, Mercury leverages the window-based congestion notification to adjust their sending rates to eliminate congestion rapidly.
- We conduct comprehensive experiments to evaluate Mercury. The results show that Mercury reduces the 99th tail FCT by up to 45.1%, 41.8%, 38.7%, 30.9%, and 37.9% compared with Timely, DCQCN, DCQCN+TCD, PACC, and HPCC, respectively.

The rest of the paper is organized as follows. "Background and motivating" section illustrates the relative background and the motivation to design Mercury. "Mercury design" section introduces the design of Mercury in detail. "Performance evaluation" section evaluates the performance of Mercury. "Related work" section introduces the related work. "Conclusion" section concludes this paper.

Background and motivating

RDMA data centers

Nowadays, the link bandwidth of data centers is growing rapidly, and applications are imposing more stringent requirements on network performance. Traditional TCP protocol has become a bottleneck of modern data centers. More and more data centers are adopting the RDMA to replace TCP [10]. With the kernel-bypass and zero-copy data transport, RDMA achieves high throughput and low latency network performance. RoCEv2 is the standardized protocol for deploying RDMA over Ethernet. It relies on PFC to achieve the lossless data transmission.

Figure 1 shows the details of the PFC mechanism. PFC is a hop-by-hop flow control mechanism that enables the downstream port to send a pause/resume frame to control the traffic transmission of the upstream port. Specifically, PFC defines two thresholds X_{off} and X_{on} for a queue of the ingress port. As shown in Fig. 1a, the downstream port sends a PFC pause frame to stop data transmission of the upstream port when the queue length exceeds X_{off}. Triggering PFC causes the link utilization between the upstream port and the downstream port to drop to zero, and packets are accumulated in the upstream queue. As shown in Fig. 1b, the downstream port sends a PFC resume frame when the queue length decreases to X_{on}, and the upstream port resumes data transmission. In this way, PFC can rapidly react to congestion and ensure lossless data delivery.

Existing congestion control schemes are insufficient

In the RDMA data centers, the existing CC schemes [10, 15, 17, 20] are insufficient in congestion feedback accuracy and speed, which affects the network performance.

Firstly, the congestion notifications of the existing CC schemes are inaccurate. They determine the congested flows based on whether the switch queue length exceeds the specified congestion threshold. When a flow is marked as congested, the switch sends a congestion notification to the receiver or the sender to guide the data transmission behavior in different ways. However, as we mentioned before, the PFC mechanism also leads to queue buildup, and the queue length may also exceed the congestion threshold. Therefore, detecting congestion only based on queue length cannot precisely identify which flow is a criminal of congestion, leading to inaccurate congestion notification and incorrect rate increment.

We conduct an experiment in the NS3 simulator [27] to illustrate this problem. We adopt a widely used fat-tree topology shown in Fig. 2. We set the bandwidth of H1-S1 and H2-S1 to 20 Gbps. The remaining link bandwidth is set to 40 Gbps. The propagation delay between switches is 4 μs. We adopt DCQCN [10] as the default congestion control scheme and set the X_{on}/X_{off} of PFC to 318/320 KB. We first generate two long-lived flows F1 and F2 at time T0. At time T1 (i.e., 3ms after T0), we send 8 concurrent burst flows to R2, lasting for about 500 μs. Ideally, port P3 of switch S3 experiences congestion due to the bursty traffic. F2 passing through the congested port will be marked with the congested flow, and the sender will slow down the rate of F2. Since F1 does not pass through the congested port P3, the rate of F1 will not be affected.

Figure 3 shows the experimental result. As shown in Fig. 3, the rates of F2 and F1 decrease consecutively. The fundamental reason is that the congestion on port P3 has spread to ports P2 and P1. Specifically, when switch S3 triggers PFC, the data transmission of port P2 is paused, resulting in the queue accumulation on port P2. Similarly,

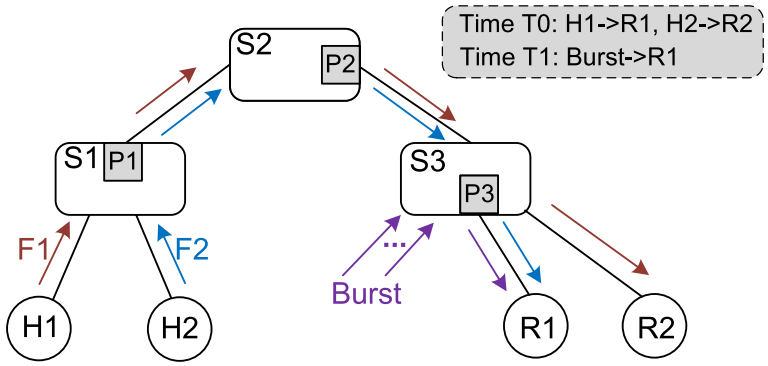


Fig. 2 Motivation topology

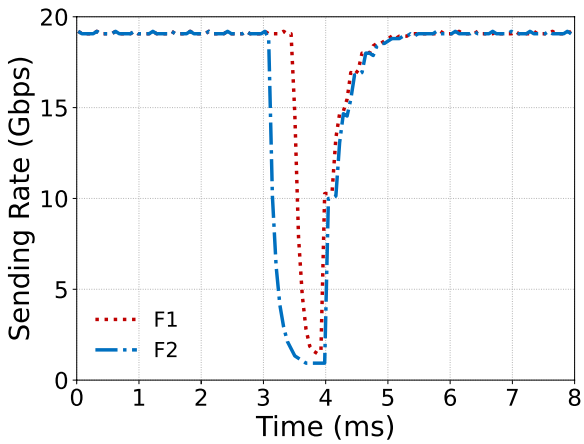


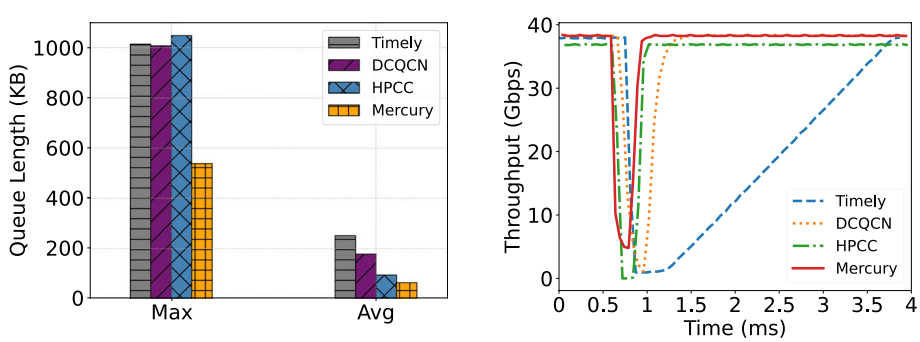
Fig. 3 Performance of congested flow and victim flow

the PFC pause frame spreads to port P1 of S1, and the queue of P1 builds up. When the queue length of P2 and P1 exceeds the congestion threshold, both F2 and F1 are marked with Explicit Congestion Notification (ECN) and treated as congested flows. Then, senders receive the congestion notifications and decrease the rates of F2 and F1. Although P1 and P2 are not congested ports and F1 is not a flow that really contributes to congestion, F1 is still mistakenly marked with ECN and experienced the rate reduction. F1 becomes a victim flow. Therefore, the existing RDMA congestion control mechanisms cannot effectively identify the congested flow and the victim flow, thereby failing to generate accurate congestion notifications.

Secondly, the congestion feedback of the existing CC schemes are slow. Most of the RDMA CC mechanisms leverage different signals (e.g., ECN [10], RTT [11] and INT [17]) to detect congestion, and then adjust the data transmission rate based on the end-to-end control signal. However, the long control loop fails to provide fast congestion feedback when the congestion occurs, resulting in queue accumulation and increasing the FCT.

We also conduct an NS3 simulation to verify this problem. We adopt a topology similar to Fig. 2 and configure 9 hosts (i.e., H1~H9) to connect to switch S1. The link bandwidth of the network is set to 40 Gbps, and the link propagation delay remains 4 μs. Initially, H1 generates a long-lived flow and transmits it to R1. After 600 μs, H2-H9 concurrent send flows with 120 KB to R1.

Figure 4 shows the queue length of S1 and the throughput of the long-lived flow. As shown in Fig. 4a, since the



(a) Queue Length of S1 (b) Throughput of the long-lived flow

Fig. 4 Performance of different congestion control mechanisms

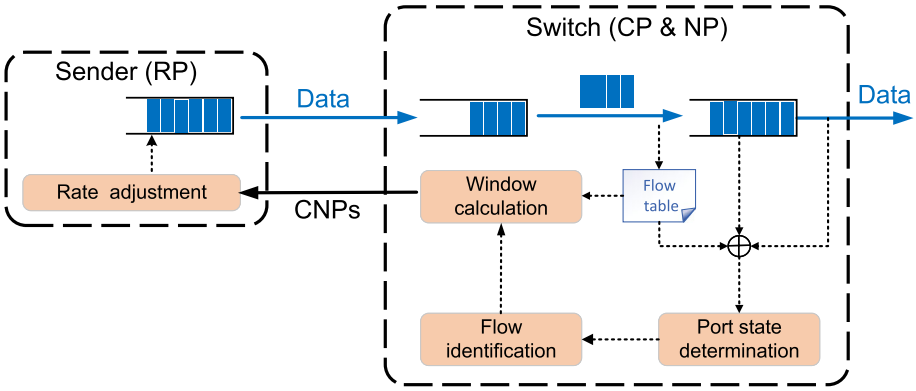


Fig. 5 Overview of Mercury

end-to-end congestion control signals cannot feedback congestion rapidly, the maximum queue accumulation produced by DCQCN [10], Timely [11], HPCC [17] exceeds 1000 KB. It incurs the long queueing delay, which greatly impacts short latency-sensitive flow. Moreover, the existing CC algorithms continuously reduce the sending rate or decrease the number of inflight packets until the queue length drops to a specified congestion threshold.The more the queue length exceeds the specified congestion threshold, the more congestion feedback signals are produced. As shown in Fig. 4b, in order to handle burst traffic, DCQCN [10], Timely [11], HPCC [17] almost decrease the throughput of the long throughput-sensitive flow to zero during a period of time. In contrast, a fast congestion feedback mechanism, i.e., Mercury, ensures the minimum throughput exceeding 6 Gbps. When the accumulated packet is below the congestion threshold, the throughput recoveries. Figure 4b reveals

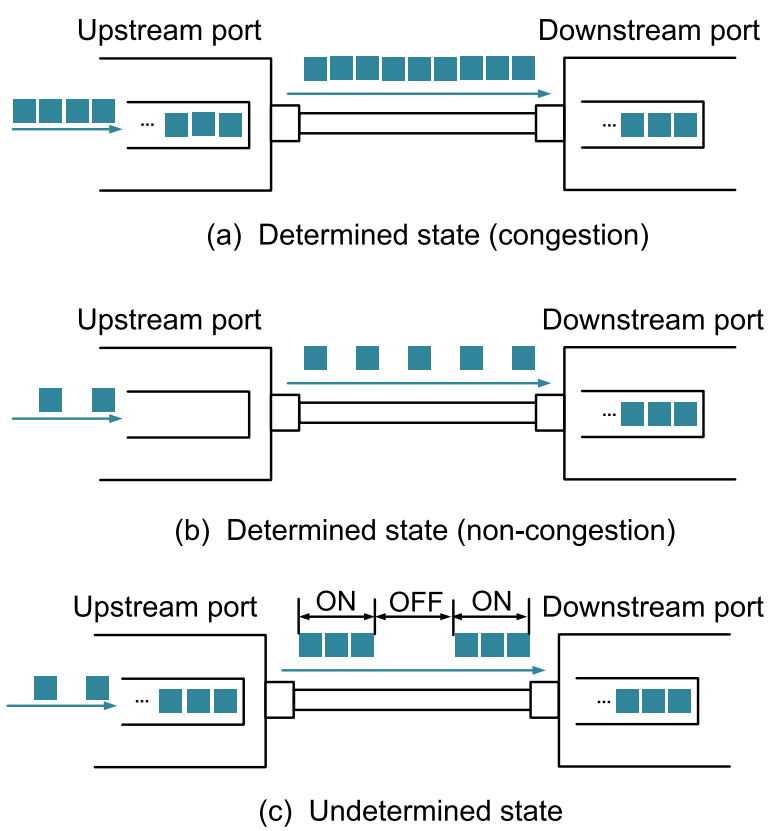


Fig. 6 States of egress ports

that the fast congestion feedback not only reduces the decline in throughput but also offers rapid throughput recovers.

Mercury design

In this section, we introduce Mercury. Specifically, we first exhibit the overview of Mercury. Then, we define different states of the switch egress ports to assist in identifying the flows that actually causing congestion. Finally, we introduce the design details of Mercury.

Mercury overview

The key insight of Mercury is to achieve accurate congestion detection and fast congestion feedback. Figure 5 shows the overview of Mercury. The traditional end-to-end congestion control mechanisms DCQCN [10] specify the behavior of three entities: Reaction Point (RP) at the sender to adjust the sending rate, Congestion Point (CP) at the switch to detect congestion, and Notification Point (NP) at the receiver to notify congestion to the sender. To get the fast congestion feedback, Mercury integrates CP and NP into the switch, i.e., the switch detects congestion and feeds back CNPs immediately when congestion occurs. It is widely used in data centers [20]. To accurately identify the real congestion flows and rapidly adjust its sending rate, Mercury presents port state determination, flow identification, and window calculation and adds them to the existing CP and NP modules. Specifically, triggering PFC causes queue accumulation, which affects the congestion detection. To accurately identify which ports are genuinely congested and which ones are affected by PFC, Mercury monitors the queue length and maintains a flow table to record the number of packets received at the egress port of the switch when the PFC is triggered. With the queue length and packet information,

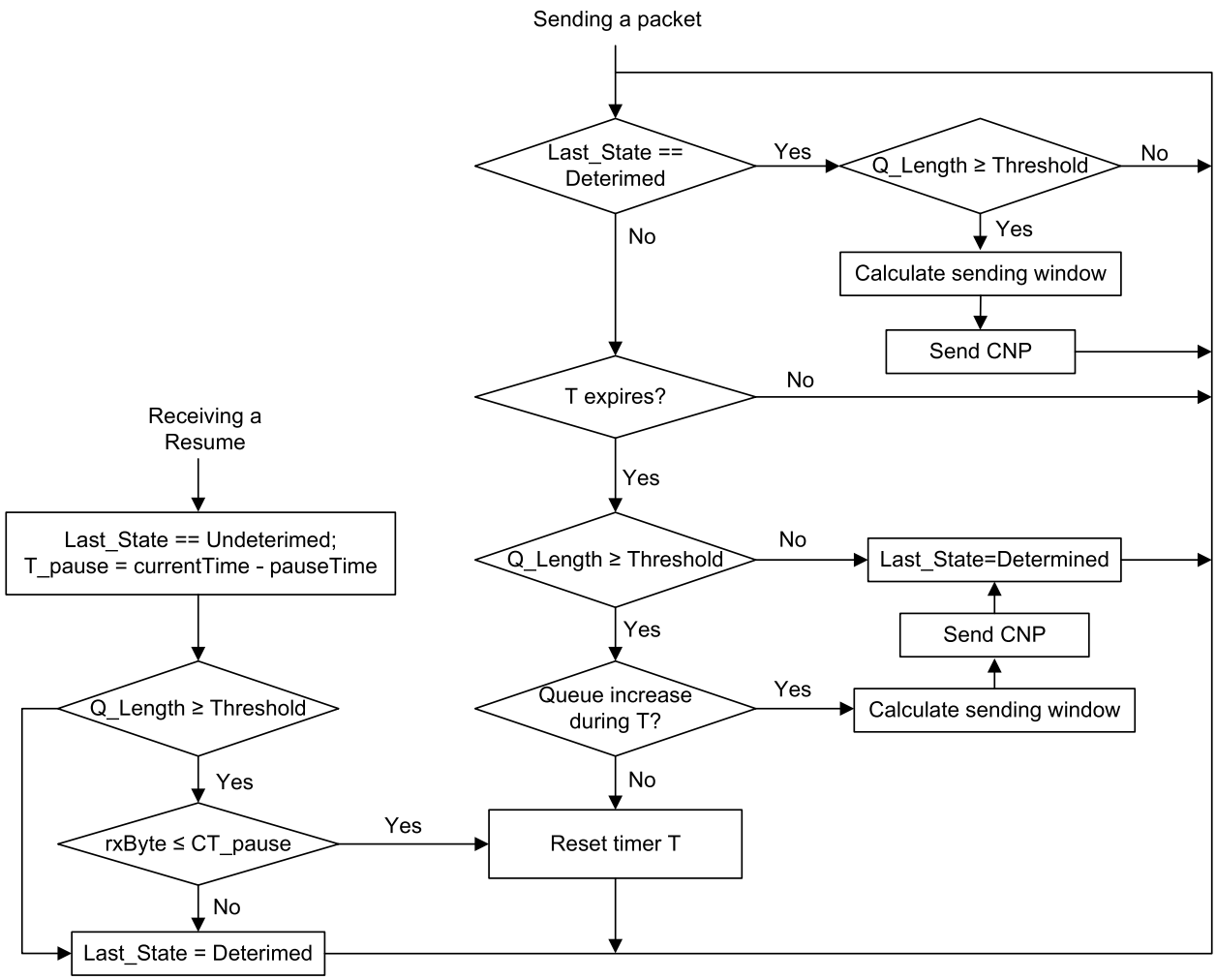


Fig. 7 Algorithm flowchart of Mercury

the port state determination module determines the state of each egress port, and then the flow identification module recognizes flows that really contribute to congestion. For the congestion flows, the window calculation module calculates their sending windows and explicitly assigns a window size to each congestion flow. The window information is carried back to the sender by a CNP. In the sender, Mercury does not require modifications to the RP algorithm of DCQCN [10], it only needs to add a sending window to the rate adjustment module to limit the number of inflight packets, which speeds up congestion elimination. Mercury is easy to deploy and compatible with the existing CC mechanisms.

State definition of egress ports

In order to distinguish the real congested flows and victim flows, Mercury divides the state of the egress port into the following two categories:

Determined state: The port being in the determined state indicates that the relationship between the incoming rate of the port and the link capacity is unequivocal. As shown in Fig. 6a, the link capacity is fully utilized and packets are accumulated in the egress queue. It indicates that the incoming rate of the upstream port exceeds the link capacity persistently. At this point, the port is in a determined congestion state. In contrast, when the link capacity is underutilized and the egress queue is empty as shown in Fig. 6b, it indicates that the incoming rate of the upstream port is under the link capacity. At this point, the port is in a determined non-congestion state. For flows passing through ports with determined congestion states, the sender can reduce their sending rates to alleviate the queue buildup. In contrast, for flows passing through ports with determined non-congestion states, the sender can increase their sending rates to improve link utilization. In summary, for the flow passing through the port with the determined state, the rate regulation mode is also determined.

Undetermined state: The port being in the undetermined state indicates that the relationship between the incoming rate of the port and the link capacity is ambiguous. Figure 6c shows the port in the undetermined state. For the port with the undetermined state, there is queue accumulation despite the link being underutilized. The reason is that when receiving PFC pause and resume frames, the data transmission of the port switches between ON and OFF. Therefore, it is unknown whether the incoming rate of packets exceeds the link capacity or not. The corresponding rate regulation mode is also undetermined. To avoid the victim flows, we do not adjust the rate of flow passing through the undetermined state port until the corresponding port changes to the determined state.

Design details

Mercury comprises two key components: (1) the state-driven congestion detection and (2) the window-based congestion notification. Figure 7 shows the process of Mercury. Parameter Last_State denotes the current port state. Parameter rxByte denotes bytes that arrived at the egress queue when the port is paused by PFC. T_{pause} is the duration for which PFC pauses the port. *Q_Length* and C is the link bandwidth. Now, we illustrate the state-driven congestion detection and the window-based congestion notification in detail.

The state-driven congestion detection. Since the network is highly dynamic, the state of the port may shift frequently. The key insight of state-driven congestion detection is to determine the state of each port rapidly and identify which flow is the congestion flow accurately.

Algorithm 1 Mercury

```
   Input: The arrived packets, the queue length Q_Length.
   Output: CNPs.
 1 procedure Receive_pause:
 2     pauseTime = currentTime ;
 3     rxByte = 0 ;
 4     if Receive a packet then
 5         rxByte+ = packet.Size ;
 6 procedure Receive_resume:
 7     T_pause = currentTime − pauseTime;
 8     if Q_Length ≥ Threshold then
 9         if rxByte ≤ C × T_pause then
10             Last_State = Undetermined ;
11         else
12             Last_State = Determined ;
13     else
14         Last_State = Determined ;
15 procedure Send_packet:
16     if Last_State == Undetermined and T expires then
17         if Q_Length ≥ Threshold and the queue length decreases during T then
18             Reset timer ;
19         else
20             Last_State = Determined;
21 function Flow_identification and Window_calculation:
22     if Flow passes through Determined port and Q_Length ≥ Threshold then
23         Win ← Algorithm 2 ;
24         Send CNP with Win[sip,dip] ;
```

At the beginning, all ports are in the *Determined* states and there is no victim flow. When PFC is triggered, the port enters the *Undetermined* state. We need to determine whether the port affected by PFC is a real congested port or not. In this way, we can identify congestion flows and victim flows. Algorithm 1 shows the details of port state determination and flow identification.

When receiving a PFC pause frame, the port pauses data transmission. Mercury records the time when data transmission is paused and initialize rxByte to 0 (lines 2-3). During the pause period, the switch updates rxByte when the egress queue of the port receives a packet (lines 4-5).When receiving a PFC resume frame, Mercury calculates the pause duration T_{pause} (line 7),

and checks the current queue length *Q_Length*. If *Q_Length* exceeds the congestion threshold, Mercury further judge the state of the port based on the relationship between the receiving rate and sending rate of the port (lines 8,9). We adopt $rxRate = \frac{rxByte}{T_{pause}}$ to represent the average receiving rate during the PFC pause period. When there is a queue accumulation, the port sends data at line speed, i.e., the link capacity C. For ease of implementation, we convert the relationship between the receiving and sending rates into whether the received packets *rxByte* can be drained out within T_{pause} at the line rate C.If $rxRate \leq C$, it indicates that the queue buildup will be alleviated when the port resumes the data transmission. Therefore, the port is not a real congestion port and the temporary data accumulation is caused by PFC. Mercury configures the port as an *Undetermined* port (line 10). Conversely, if $rxByte > C \times T_{pause}$, it demonstrates that although the port resumes data transfer, the queue length will continue to increase. Therefore, the port is a congestion port and the state of the port is *Determined* (line 12). For the condition where *Q_Length* below the congestion threshold, the port is the non-congestion port and the state of the port is also configured as *Determined* (lines 13-14).

Since the *Undetermined* state is temporary. Mercury set a timer *T* and periodically checks *Q_Length* to determine whether to change the port from the *Undetermined* state to the *Determined* state (lines 15-20). Specifically, when a packet dequeues from the *Undetermined* port and *T* expires, Mercury checks the current queue length and the evolution of the queue length during *T*. If the current queue length exceeds the congestion threshold and the queue length decreases during T, it indicates that the port is still affected by PFC. Therefore, the port remains in the *Undetermined* state. Otherwise, the port swifts to the *Determined* state.

The different port states enable Mercury to distinguish ports affected by PFC and ports that actually experience congestion. The flow_identification function identifies congested flows, i.e., flows passing through the congestion port. Then, the window_calculation function calculates the sending window of the congestion flow. Mercury only sends a CNP with window information to enable the sender to adjust the rates of congestion flows, thus avoiding mistakenly slowing down the rate of victim flows.

The window-based congestion notification. The key insight of window-based congestion notification is calculating the sending window for congestion flow and carrying it back to the sender by the CNP. Similar to the sending window adopted in the TCP, the function of the sending window is to limit the number of packets that are already sent but have not yet been received by the receiver. Algorithm 2 illustrates the window-based congestion notification.

Algorithm 2 Compute Window

Input: The link bandwidth C, the base propagation RTT $baseRTT$.
Output: CNP with Win[sip,dip].

```
1  /*Switch*/
2  procedure Receive_packet:
3      port.flowTable[sip, dip].data+ = Packet.Size ;
4      Sum(port.flowTable.data)+ = Packet.Size ;
5  procedure Send_packet:
6      port.flowTable[sip, dip].data− = Packet.Size ;
7      Sum(port.flowTable.data)− = Packet.Size ;
8      if port.flowTable[sip, dip].data == 0 then
9          Clear port.flowTable[sip, dip] ;
10 function Window_calculation:
11     Rate[sip, dip] = C × port.flowTable[sip,dip].data / Sum(port.flowTable.data) ;
12     Win[sip, dip] = Rate[sip, dip] × baseRTT;
13     Return Win[sip, dip]
14 /*Sender*/
15 procedure Receive_CNP:
16     Decreases its sending rate by DCQCN; ;
17     cwnd = min(cwnd, Win[sip, dip]) ;
18 procedure Rate_recovery:
19     Increases its sending rate by DCQCN; ;
20     cwnd = C × baseRTT;
```

Initially, the window sizes are uniformly set to $C \times baseRTT$ to fully utilize the link bandwidth, where *baseRTT* is the base propagation RTT. For the specified network topology, *baseRTT* can be known in advance [17]. Mercury maintains a flow table to record the packets of each flow and the total packets on the congested port. Mercury uses the source address and destination address as flow identifier (FID), and uses the combination of the flow's egress port and the hash of FID to index the table entry. The flow table is updated when the switch receives a packet or sends a packet (lines 1-6). When the flow size of the flow table is zero, the switch deletes the corresponding entry to save memory space (lines 7-8). When congestion occurs, Mercury calls the Window_calculation function to calculate the window size of the congestion flows (lines 10-13). Specifically, $Rate[sip, dip] = C \times \frac{port.flowTable[sip,dip].data}{Sum(port.flowTable.data)}$ obtains the upper bound of the flow rate that will not cause congestion in the switch. Since the topology in data centers is regular and the baseRTT is usually known in advance [17], we can obtain the sending window of each congestion flow according to the equation $Win[sip, dip] = Rate[sip, dip] \times baseRTT$. Mercury uses the 32-bit reserved segment in CNP to carry the window information, which is compatible with the CNP packet in RoCEv2 [24]. Then, Mercury sends CNP to the sender.

When the sender receives a CNP, it reduces its sending rate based on the existing congestion control scheme (we

adopt DCQCN by default). Meanwhile, the sender parses the CNP and updates its congestion window cwnd (lines 15-17). When the number of inflight packets exceeds the window size, the sender stops packet transmission immediately. It allows the congestion switch to drain out its egress queue quickly. When congestion disappears, the sender no longer receives CNP and gradually recovers the sending rate. At this point, the window size will be set to the initial value (i.e., $C \times baseRTT$) to enable the sender to recover its sending rate rapidly (lines 18-20).

Discussions

In this section, we discuss the implementation of Mercury in the real environment.

Mercury is mainly implemented in programmable switches and is compatible with the existing CC mechanisms. Mercury can inherit the rate control scheme of the existing CC mechanisms at the sender without any modification. It only requires an additional register to store the window information received from CNP to limit the number of flight packets. In order to achieve accurate and fast congestion feedback, Mercury needs a register at the switch to record the port state (i.e., Last_State). Besides, Mercury also needs registers to track packets that arrive at the egress queue when the port is paused by PFC and record the PFC duration. In commercial switches, registers are abundant to achieve the above operations. Mercury also maintains a flow table to record the number of packets of each flow received at the egress port when the PFC is triggered. With the above information, Mercury can obtain the sending rate of each flow. To get the window size, we need to further determine baseRTT. Due to the regularity of the topology, the RTT between server pairs in the data center is very close which makes it possible for all flows to use the same baseRTT [17]. Therefore, if the network topology is determined, baseRTT can be pre-configured as a known parameter in switches. The switch sends a CNP with the window information to the sender if the switch is congested. The CNPs can be generated on the control plane of the programmable switches, which has already been achieved in [28].

Performance evaluation

In this section, we conduct both small-scale experiments and large-scale experiments based on the NS3 simulator to evaluate the performance of Mercury. We adopt the widely-used congestion control algorithm DCQCN [10] as the default congestion control algorithm of Mercury. In fact, Mercury can also be compatible with other RDMA congestion control schemes. We adopt the open-source code [29] to implement Mercury.

Experiment settings

Topology: In the large-scale simulation, we adopt a fat-tree topology [30], which includes 20 ToR switches, 20 aggregation switches, and 16 core switches. Each ToR switch is connected to 16 servers, and the link bandwidth between the ToR switch and the server is 100 Gbps. The rest of the link bandwidth is set to 400 Gbps. We set the propagation delay of each link to 1 us, thus the maximum base RTT is 12 us. The whole network is a single RDMA domain.

Benchmarks: We compare Mercury with several RDMA CC mechanisms, i.e., Timely [11], DCQCN [10], PACC [20], and HPCC [17], where Timely [11], DCQCN [10], and HPCC [17] are common used end-to-end CC mechanisms and PACC [20] is the state-of-the-art switch-driven CC mechanism. Besides, we combine the state-of-the-art congestion detection algorithm TCD [26] with DCQCN [10] and treat DCQCN+TCD as one of the comparison algorithms. All of the benchmarks are implemented based on the open-source code [29].

Parameter settings: We set all experiments to enable PFC. In the small-scale experiments, we set the PFC thresholds X_{off} to 320 KB and X_{on} to 318 KB. In the large-scale experiments, we set X_{off} to 620 KB and X_{on} to 618 KB. Mercury needs to check the queue length and update the port status periodically. We set the corresponding period T to $10us$. For the benchmarks, unless otherwise specified, we employ the parameter settings recommended in their papers [10, 11, 17, 20, 26].

Workloads: We adopt four widely-used realistic workloads, i.e., Hadoop [31], WebServer [31], CacheFollower [32], WebSearch [33] to analyze the performance of Mercury. The flow distributions of the four workloads are shown in Fig. 8. In the Hadoop cluster, about 60% of flows are smaller than 1 KB, and in the web server cluster, about 80% of flows are less than 10 KB. Compared with Hadoop and Websearch, WebSearch and CacheFollower have more long flows. In each workload, we generate flows following a Poisson arrival process.

Performance Metrics: We verify the performance of Mercury from the following four aspects: (1) throughput, (2) buffer usage, (3) average FCT, (4) 99th tail FCT.

Small-scale simulations

We first conduct small-scale experiments. We still adopt the topology shown in Fig. 2. We maintain the parameter settings and traffic generation model in "Existing congestion control schemes are insufficient" section, i.e., there are two long-lived flows F1 and F2, and 8 concurrent burst flows in the network. The size of each burst flow is 50KB, and the duration is 500 μs. Since the size of the

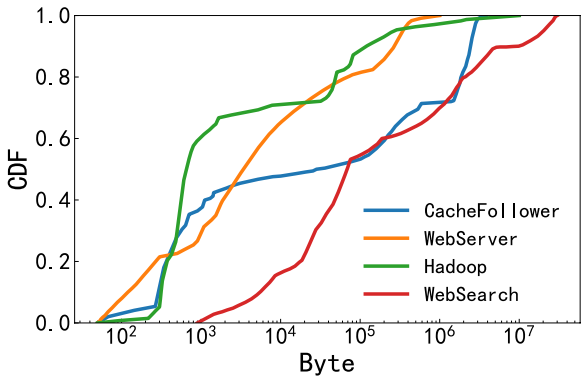


Fig. 8 Flow distribution of typical workloads

burst flow is smaller than the Bandwidth Delay Product (BDP), it is impossible for the congestion control mechanisms to adjust the rate of the burst flow. In this case, the queue is building up on port P3 of S3. Then, S3 triggers PFC to pause the data transmission and the pause may spread to port P1 of S1. As we illustrate in "Existing congestion control schemes are insufficient" section, F2 is a congestion flow and F1 is a victim flow.

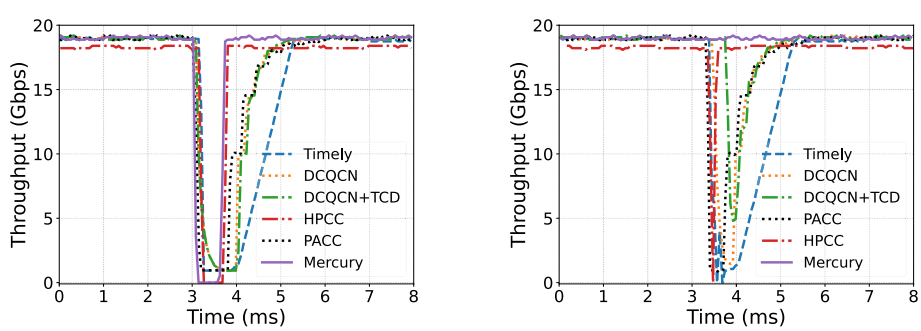
Figure 9 shows the throughput of the congestion flow F2 and the victim flow F1. Figure 9a shows that all of the algorithms slow down the congestion flow. Compared with several benchmarks, Mercury can quickly adjust the sending rate of the congestion flow when the congestion occurs or disappears. It benefits from the fact that Mercury can respond to congestion at the nearby switch and the window carried in CNP further controls the number of packets sent by the source. Figure 9b shows that both Timely, DCQCN, DCQCN+TCD, HPCC, and PACC reduce the rate of the victim flow, while Mercury keeps the rate of victim flow unchanged. Timely, DCQCN, HPCC, and PACC do not have the ability to identify the congested flow and victim flow when PFC is triggered, so they regard the victim flow as the congested flow and thus sharply slow it down. It increases the FCT of the victim flow and incurs throughput loss. Although DCQCN+TCD can identify victim flows, the rate of the victim flow still decreases to 5 Gbps in our simulation. It indicates that DCQCN+TCD may misjudge the victim flow when congestion is severe. As a comparison, Mercury identifies the victim flow accurately and only adjusts the rate of the congestion flow.

Afterward, we still use the topology shown in Fig. 1 and set all of the link rates to 40Gpbs. We set H1 and H2 as senders, and R1 and R2 as receivers. We generate 80% Hadoop workload and observe the performance of Mercury. The results are shown in Fig. 10b. Compared to other algorithms, Mercury reduces the 99th tail FCT by about 16.1%~25.5%, and reduces the maximum queue occupancy of the switch by about 20.9%~39.2%. The reason is that congestion mainly occurs in the first hop switch. Mercury can detect the congestion at the switch rapidly and return CNPs with the sending window directly, which eliminates congestion quickly.

Large-scale simulations

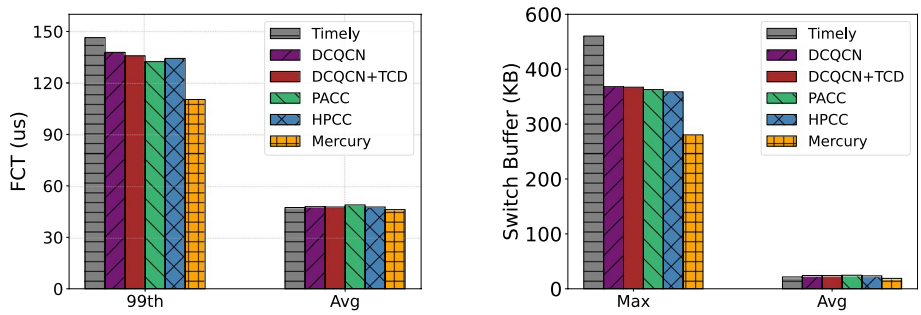
We also conduct large-scale simulations to comprehensively evaluate the performance of Mercury. As mentioned in "Experiment settings" section, we adopt a fat-tree topology with 320 servers. We analyze Mercury under different workloads and traffic loads.

Figures 11 and 12 show the average and 99th tail FCT under the four workloads. The results show that the average and 99th tail FCT increase as the traffic load increases. We note that even Mercury can always provide the lowest average FCT and 99th tail FCT compared with other algorithms under different traffic patterns. Specifically, Mercury reduces the overall average FCT of by 21.1%~38.5% compared with other algorithms in CacheFollwer workload. In Hadoop workload, the overall average FCT reduction is 11.2%~36.1%.



(a) Congestion flow (b) Victim flow

Fig. 9 Throughput in small-scale simulation



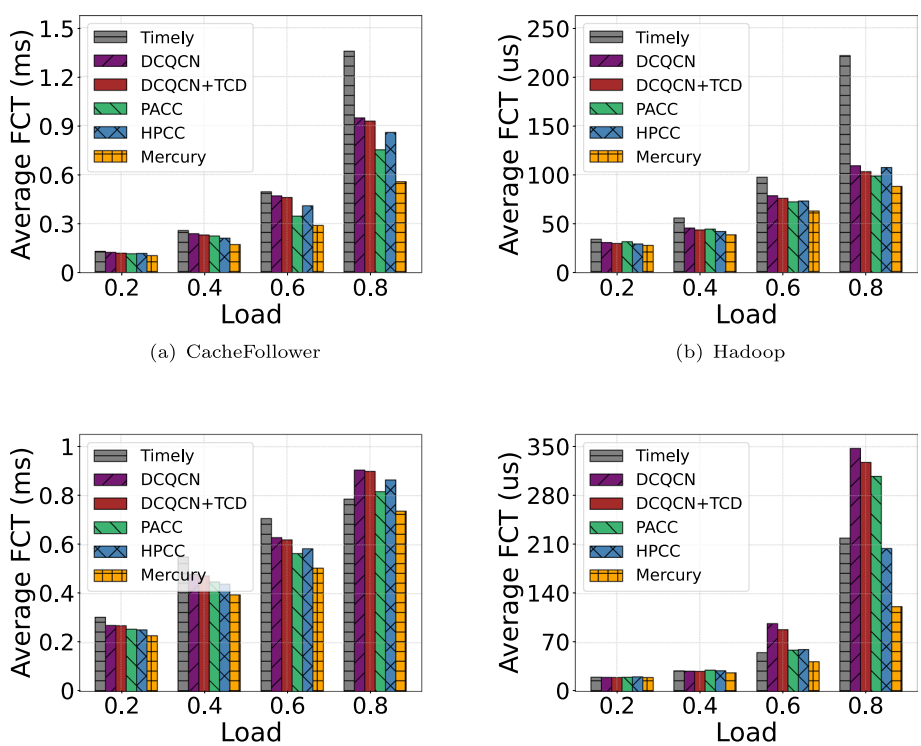
(a) FCT (b) Switch buffer usage

Fig. 10 Performance on Hadoop workload

Similarly, Mercury reduces the overall average FCT by up to 21.9% in WebSearch workload and 32.6% in WebServer workload. The results of the 99th tail FCT maintain similar trends. The reason is that Mercury can identify which flow is the culprit of congestion and only reduce the rate of the congestion flows. The throughput of the victim flow is not affected, thus reducing the FCT of the victim flow. Besides, Mercury enables the switch to send the sending window of each congestion flow rapidly, the FCT of the congestion flows will be reduced accordingly.

To further verify the performance of Mercury for different sizes of flows, we decomposed the FCT in the workloads by flow size. Since the performance trends of different workloads are similar, we only present the results on the CacheFollower and WebSearch workloads as shown in Figs. 13 and 14.

Figures 13a~b and 14a~b show the average FCT and 99th tai FCT for short flows. The FCT of Mercury for short flows is lower than that of Timely, DCQCN, DCQCN+TCD, and PACC, and the average FCT is only slightly higher than that of HPCC. Specifically, in the



(a) CacheFollower (b) Hadoop

(c) WebSearch (d) WebServer

Fig. 11 Average FCT

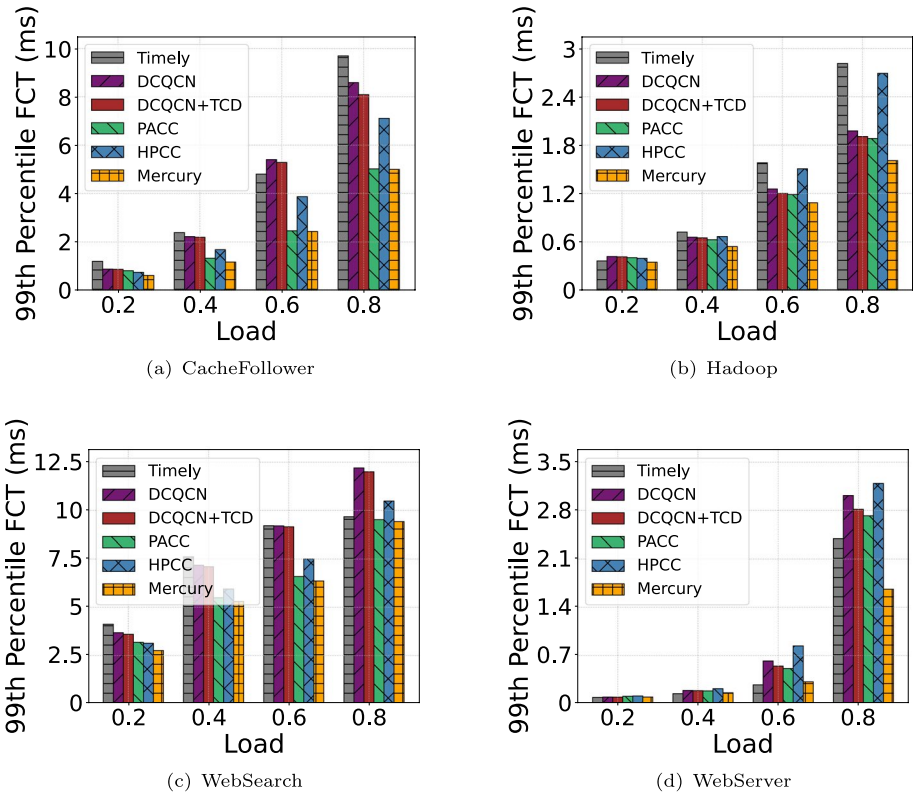
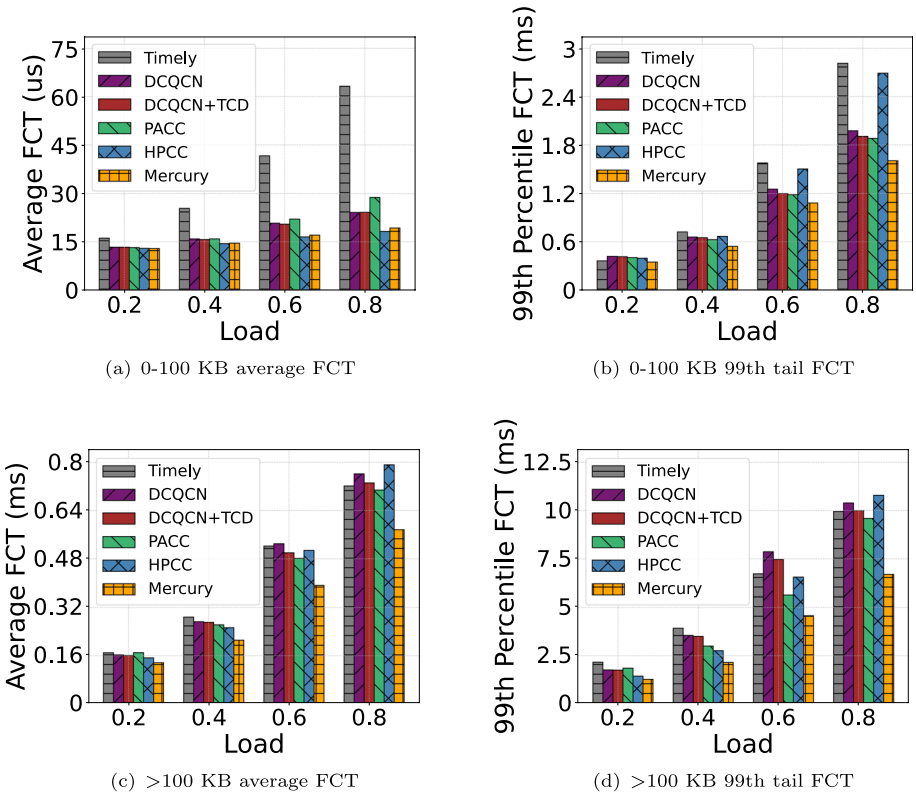


Fig. 12 99th tail FCT

Hadoop workload, Mercury decreases at most 64.7% average FCT and 58.7% tail FCT for short flows compared with Timely, DCQCN, DCQCN+TCD, and PACC. In the Webserver workload, the average and 99th tail FCT reduction are at most 81.1% and 80.6%, respectively. Figures 13c~d and 14c~d show the average FCT and 99th tai FCT for long flows. The results show that Mercury outperforms other algorithms in both the average FCT and 99th tail FCT. Specifically, under the Hadoop workload, Mercury decreases the average FCT by up to 26.5%, 26.3%, 21.8%, 19.7% and 27.1% and tail FCT by up to 45.1%, 41.8%, 38.7%, 30.9% and 37.9% compared with Timely, DCQCN, DCQCN+TCD, PACC, and HPCC, respectively. For the WebServer workload, we observed a similar trend. Mercury reduces the average FCT and tail FCT by up to 56.2% and 68.6% compared to other schemes.

The reasons for the above results are as follows. Timely and DCQCN rely on end-to-end congestion control signals. They cannot control the rate of short flows that are less than 1 RTT. Therefore, packets accumulate in the network when congestion occurs, prolonging the FCT of short flows and long flows. PACC inherits the rate control of DCQCN. It detects congestion at the switch and assigns CNPs through the PI controller. Although PACC reduces FCT in most scenarios compared to DCQCN, it needs to wait for a period of time (80 μs by default) to update the flow table and send CNPs, which is not timely for 40/100 Gbps data centers. Therefore, PACC also suffers from long FCTs. HPCC adopts the sending window to control the number of inflight packets. With the sending window, HPCC avoids data accumulation in the switch and maintains near-zero in-network queues. As shown in Figs. 13a~b and 14a~b, HPCC provides ultra-low latency for short flows. However, HPCC needs to adjust the window size so that the inflight bytes passing through the bottleneck link are slightly less than the product of bandwidth and base RTT. It may cause the bandwidth underutilization. As shown in Figs. 13c~d and 14c~d, HPCC increases the FCT of long flows. Besides, both Timely, DCQCN, PACC, and HPCC do not identify the victim flows, which may improperly slow down the rate of the victim flow, thus affecting the FCT of the victim flow. Although DCQCN+TCD provides a method to detect victim flows and reduce the FCT compared with DCQCN, TCD is still possible to misjudge victim flows as congestion flows when congestion is severe. Besides, TCD determines the state of each flow after a pre-configured parameter $max(T_{on})$ expires, which increases the congestion notification



(a) 0-100 KB average FCT (b) 0-100 KB 99th tail FCT

(c) >100 KB average FCT (d) >100 KB 99th tail FCT

Fig. 13 Overall Performance in Hadoop workload

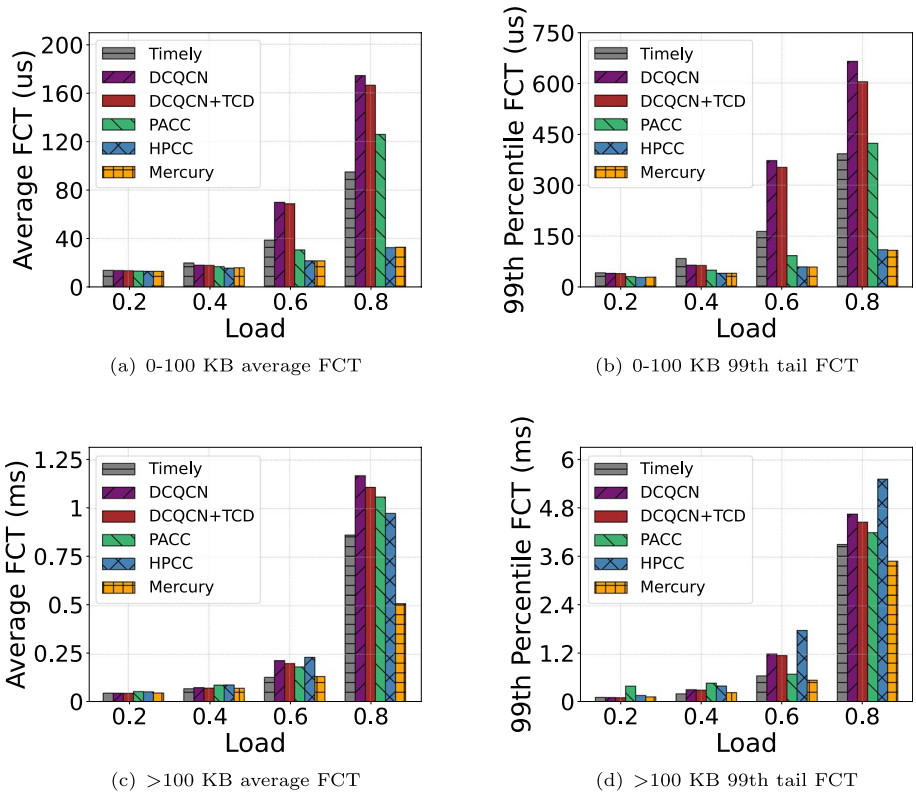
delay. As a comparison, Mercury improves the network performance under different workloads, and the effect is more obvious in the WebServer workload. It benefits from the fact that Mercury can detect and identify the congestion rapidly at the in-network switch. The sender receives the congestion notification timely and adjusts the rate of congestion flow to avoid queue accumulation. The window calculated by Mercury limits the number of inflight packets, which further speeds up the queue emptying. Besides, Mercury keeps the rate of victim flow unchanged, which ensures the throughput and reduces the FCT of victim flows.

Related work

Congestion control is an enduring topic in data centers. In the last few years, several novel congestion control methods have been proposed to improve network performance [34, 35]. In this section, we will briefly introduce some closely related work from the following aspects.

End-to-end congestion control: DCTCP [33] is the first congestion control mechanism that leverages Explicit Congestion Notification (ECN) to detect and respond to network congestion. In RDMA data center, QCN [36] provides end-to-end congestion control based on the network feedback (e.g., ECN) at Layer 2. However, QCN cannot be implemented in IP routing networks, which is not suitable for large-scale data centers. DCQCN [10] is designed based on DCTCP and QCN. It uses the ECN and PFC to achieve rate-based congestion control and lossless data transmission. DCQCN is the most widely used congestion control algorithm in RDMA data centers, and has been integrated into RDMA NIC(RNIC) as the default mechanism. Timely [11] and Swift [16] are RTT-based RDMA congestion control solutions that monitor network congestion by measuring RTT and then converting RTT signals into target transmission rates. HPCC [17] adopts In Network Telemetry (INT) to monitor the traffic load on each link and control the number of inflight bytes passing through bottleneck links. Since HPCC cannot avoid triggering PFC [37], it also faces the issue of incorrectly identifying flows that really contribute to congestion. Besides, as RDMA link bandwidth and burst traffic continue to increase, end-to-end congestion control mechanisms struggle to respond to network congestion rapidly due to their inherent long control cycles. As a supplement to existing end-to-end congestion control algorithms, Mercury can accurately identify congested flows and provide rapid congestion feedback.

Switch-driven congestion control: XCP [38] and RCP [39] require the switch to calculate the window size and a fair-shared rate per link, and then feedback congestion on the switch. However, the control signals of XCP [38]



(a) 0-100 KB average FCT (b) 0-100 KB 99th tail FCT

(c) >100 KB average FCT (d) >100 KB 99th tail FCT

Fig. 14 Overall Performance in WebServer workload

and RCP [39] still experience end-to-end propagation delay. TFC [40] counts the number of active flows within a fixed time interval and adopts a token-based bandwidth allocation scheme to alleviate congestion. Instead of generating tokens, RoCC [28] detects the queue length of the switch as the input of the PI controller to compute the fair flow rate. Similar to RoCC [28], PACC [20] also proposes a PI controller-based method to generate CNPs in proportion to the number of congested packets. However, the above congestion control mechanisms overlook the queue accumulation caused by PFC, making it difficult to detect the real congested flows. Therefore, they are unable to provide accurate congestion feedback.

Congestion feedback: Congestion feedback is a critical component of congestion control. It determines when and where the congestion control takes effect. Recently, several research works [24, 26] have begun to investigate congestion feedback in RDMA data centers. They aim to design effective mechanisms to distinguish congestion flows and victim flows accurately, and then provide the congestion feedback for congested flows. PCN [24] presents the RDMA congestion management method to detect and identify congested flows according to the link utilization. Then, PCN [24] adopts the end-to-end congestion control signal to only regulate the rate of congested flows. On this basis, TCD [26] detects the congested port based on the ternary states, thereby accurately identifying the congestion. However, TCD determines the egress port states only after a pre-configured parameter $max(T_{on})$ expires. It may prolong the congestion feedback time and mislabel flows when the PFC is triggered frequently. Therefore, the existing RDMA congestion feedback mechanisms still have limitations in terms of detection accuracy and feedback speed.

Conclusion

This paper presents Mercury, an accurate and fast congestion feedback mechanism for MEC-enabled RDMA data centers. By leveraging the switch queue length and port sending rate, Mercury can accurately identify flows that really contribute to the congestion. For the real congested flows, Mercury calculates the sending windows and enables the switch to send CNPs with the window information, thereby achieving the fast rate adjustment to eliminate congestion. The micro-benchmark and large-scale simulation show that Mercury can significantly improve the throughput and reduce the FCT under realistic workloads.

Authors' contributions

Xin He wrote the main manuscript and performed the experimental design. Feifan Liang conducted experiments. Weibei Fan and Junchang Wang prepared figures. Lei Han, Fu Xiao, and Wanchun Dou supervised the experimental design. All authors reviewed the manuscript.

Funding
This work was supported in part by the Natural Science Foundation of China under Grant No. 62202237, the Natural Science Foundation of Jiangsu Province under Grant No. BK20220389, the Natural Science Foundation of the Higher Education Institutions of Jiangsu Province under Grant No. 22KJB520005, the State Key Lab. for Novel Software Technology under Grant No. KFKT2022B04, the Natural Science Research Start-up Foundation of Recruiting Talents of Nanjing University of Posts and Telecommunications under Grant No. NY222016.

Availability of data and materials
No datasets were generated or analysed during the current study.

Declarations

Ethics approval and consent to participate
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 26 January 2024 Accepted: 19 March 2024
Published online: 25 March 2024

References

1. Xu Z, Zhang Y, Li H, Yang W, Qi Q (2020) Dynamic resource provisioning for cyber-physical systems in cloud-fog-edge computing. J Cloud Comput 9(1):1–16
2. Zhang M, Wang S, Gao Q (2020) A joint optimization scheme of content caching and resource allocation for internet of vehicles in mobile edge computing. J Cloud Comput 9:1–12
3. Chen H, Qin W, Wang L (2022) Task partitioning and offloading in IoT cloud-edge collaborative computing framework: a survey. J Cloud Comput 11(1):86
4. (2015) Amazon Web Services. https://aws.amazon.com/s3/
5. (2018) Google Cloud Platform. https://cloud.google.com
6. (2015) Microsoft Azure. https://azure.microsoft.com
7. Song CH, Khooi XZ, Joshi R, Choi I, Li J, Chan MC (2023) Network load balancing with in-network reordering support for rdma. In: Proceedings of the ACM SIGCOMM 2023, ACM, New York, p 816–831
8. Wang W, Moshref M, Li Y, Kumar G, Ng TSE, Cardwell N, Dukkipati N (2023) Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In: Proceedings of the USENIX NSDI 2023, USENIX Association, Boston, p 255–274
9. Marinos I, Watson RNM, Handley M (2014) Network stack specialization for performance. In: Proceedings of the ACM SIGCOMM 2014, ACM, Chicago, p 175–186
10. Zhu Y, Eran H, Firestone D, Guo C, Lipshteyn M, Liron Y, Padhye J, Raindel S, Yahia MH, Zhang M (2015) Congestion control for large-scale rdma deployments. ACM SIGCOMM Comput Commun Rev 45(4):523–536
11. Mittal R, Lam VT, Dukkipati N, Blem E, Wassel H, Ghobadi M, Vahdat A, Wang Y, Wetherall D, Zats D (2015) Timely: Rtt-based congestion control for the datacenter. ACM SIGCOMM Comput Commun Rev 45(4):537–550
12. (2021) IEEE 802.1 Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html
13. Guo C, Wu H, Deng Z, Soni G, Ye J, Padhye J, Lipshteyn M (2016) Rdma over commodity ethernet at scale. In: Proceedings of the ACM SIGCOMM 2016, ACM, Florianópolis p 202–215
14. Zhu Y, Ghobadi M, Misra V, Padhye J (2016) Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In: Proceedings of the ACM CoNEXT 2016, ACM, California p 313–327
15. Gao Y, Yang Y, Chen T, Zheng J, Mao B, Chen G (2018) Dcqcn+: Tam ing large-scale incast congestion in rdma over ethernet networks. In: Proceedings of the IEEE ICNP 2018, IEEE, Cambridge p 110–120
16. Kumar G, Dukkipati N, Jang K, Wassel HMG, Wu X, Montazeri B, Wang Y, Springborn K, Alfeld C, Ryan M, Wetherall D, Vahdat A (2020) Swift: Delay is simple and effective for congestion control in the datacenter. In: Proceedings of the ACM SIGCOMM 2020, ACM, p 514–528. Online Conference
17. Li Y, Miao R, Liu HH, Zhuang Y, Feng F, Tang L, Cao Z, Zhang M, Kelly F, Alizadeh M, Yu M (2019) Hpcc: high precision congestion control. In: Proceedings of the ACM SIGCOMM 2019, ACM, Beijing p 44–58
18. Shan D, Liu Y, Zhang T, Liu Y, Tang Y, Li H, Zhang P (2023) Less is more: Dynamic and shared headroom allocation in pfc-enabled datacenter networks. In: IEEE International Conference on Distributed Computing Systems (ICDCS), IEEE, p 591–602
19. Taheri P, Menikkumbura D, Vanini E, Fahmy S, Eugster P, Edsall T (2020) Rocc: robust congestion control for rdma. In: Proceedings of the ACM CoNEXT 2020, ACM, Barcelona p 17–30
20. Zhong X, Zhang J, Zhang Y, Guan Z, Wan Z (2022) Pacc: Proactive and accurate congestion feedback for rdma congestion control. In: Proceedings of the IEEE INFOCOM 2022, IEEE, London, p 2228–2237
21. Menikkumbura D, Taheri P, Vanini E, Fahmy S, Eugster P, Edsall T (2023) Congestion control for datacenter networks: A control-theoretic approach. IEEE Trans Parallel Distrib Syst 34(5):1682–1696
22. Bosshart P, Daly D, Gibb G, Izzard M, McKeown N, Rexford J, Schlesinger C, Talayco D, Vahdat A, Varghese G et al (2014) P4: Programming protocol-independent packet processors. ACM SIGCOMM Comput Commun Rev 44(3):87–95
23. (2020) Barefoot tofino. https://barefootnetworks.com/products/brieftofino/
24. Cheng W, Qian K, Jiang W, Zhang T, Ren F (2020) Re-architecting congestion management in lossless ethernet. In: Proceedings of the USENIX NSDI 2020, USENIX Association, Santa Clara, p 19–36
25. Zhang Y, Meng Q, Liu Y, Ren F (2003) Revisiting congestion detection in lossless networks. IEEE/ACM Transactions on Networking 31(5):2361-75
26. Zhang Y, Liu Y, Meng Q, Ren F (2021) Congestion detection in lossless networks. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. SIGCOMM '21, ACM, p 370–383. Online Conference
27. (2023) Network simulator 3. https://www.nsnam.org/
28. Menikkumbura D, Taheri P, Vanini E, Fahmy S, Eugster P, Edsall T (2023) Congestion control for datacenter networks: A control-theoretic approach. IEEE Trans Parallel Distrib Syst 34(5):1682–1696
29. (2019) Alibaba. 2019. HPCC simulator. https://github.com/alibaba-edu/High-Precision-Congestion-Control
30. Al-Fares M, Loukissas A, Vahdat A (2008) A scalable, commodity data center network architecture. ACM SIGCOMM Comput Commun Rev 38(4):63–74
31. Roy A, Zeng H, Bagga J, Porter G, Snoeren AC (2015) Inside the social network's (datacenter) network. In: Proceedings of the ACM SIGCOMM 2015, ACM, London, p 123–137
32. Montazeri B, Li Y, Alizadeh M, Ousterhout J (2018) Homa: a receiver-driven lowlatency transport protocol using network priorities. In: Proceedings of the ACM SIGCOMM 2018, ACM, Budapest, p 221–235
33. Alizadeh M, Greenberg A, Maltz DA, Padhye J, Patel P, Prabhakar B, Sengupta S, Sridharan M (2010) Data center tcp (dctcp). In: Proceedings of the ACM SIGCOMM 2010, ACM, New Delhi, p 63–74
34. Huang J, Li W, Li Q, Zhang T, Dong P, Wang J (2020) Tuning high flow concurrency for mptcp in data center networks. J Cloud Comput 9:1–15
35. Sun X, Wang Z, Wu Y, Che H, Jiang H (2021) A price-aware congestion control protocol for cloud services. J Cloud Comput 10:1–15
36. Pan R, Prabhakar B, Laxmikantha A (2007) QCN: Quantized congestion notification. IEEE802 1:52–83
37. Li W, Zeng C, Hu J, Chen K (2023) Towards fine-grained and practical flow control for datacenter networks. In: Proceedings of the IEEE ICNP 2023, IEEE, Reykjavik, p 1–11
38. Katabi D, Handley M, Rohrs C (2002) Congestion control for high bandwidth delay product networks. In: Proceedings of the ACM SIGCOMM 2002, ACM, Pittsburgh Pennsylvania, p 89–102
39. Dukkipati N, McKeown N (2006) Why flow-completion time is the right metric for congestion control. ACM SIGCOMM Comput Commun Rev 36(1):59–62
40. Zhang J, Ren F, Shu R, Cheng P (2016) Tfc: token flow control in data center networks. In: Proceedings of the ACM EuroSys 2016, ACM, London

Publisher's Note