R-AQM: Reverse ACK Active Queue Management in Multi-tenant Data Centers

Xinle Du[1], Tong Li[2]*, Lei Xu[2], Kai Zheng[2], Meng Shen[3], Bo Wu[1], Ke Xu[145]
Tsinghua University[1], Huawei[2], Beijing Institute of Technology[3], BNRist[4], PCL[5]
Email: dxl18@mails.tsinghua.edu.cn, li.tong@huawei.com, thuxl07@gmail.com, kai.zheng@huawei.com
shenmeng@bit.edu.cn, wub14@tsinghua.org.cn, xuke@tsinghua.edu.cn

***Abstract*—TCP incast has become a practical problem for high-bandwidth, low-latency transmissions, resulting in throughput degradation of up to 90% and delays of hundreds of milliseconds, severely impacting application performance. However, in virtualized multi-tenant data centers, host-based advancements in the TCP stack are hard to deploy from the operators perspective. Operators only provide infrastructure in the form of virtual machines, in which only tenants can directly modify the end-host TCP stack. In this paper, we present R-AQM, a switch-powered reverse ACK active queue management (R-AQM) mechanism for enhancing ACK-clocking effects through assisting legacy TCP. Specifically, R-AQM proactively intercepts ACKs and paces the ACK-clocked in-flight data packets, preventing TCP from suffering incast collapse. We implement and evaluate R-AQM in NS-3 simulation and NetFPGA-based hardware switch. Both simulation and testbed results show that R-AQM greatly improves TCP performance under heavy incast workloads by significantly lowering packet loss rate, reducing retransmission timeouts, and supporting 16 times (i.e., 60→1000) more senders. Meanwhile, the forward queuing delays are also reduced by 4.6 times.**

I. Introduction

Data centers have evolved rapidly over the last few years, providing a wide variety of cloud services [1], [2] using TCP as the dominant transport layer protocol. However, the TCP incast problem causes drastic performance degradation when multiple senders synchronously send data to one receiver (i.e., many-to-one communication) with high-bandwidth and low-latency links [3], [4]. As the number of senders increases, bottleneck switches can quickly become overfilled. Inevitable packet drops would impose TCP retransmission timeout (RTO) for hundreds of milliseconds, resulting in goodput (the application-level throughput [5]) reduction of up to 90% [6], which affects the performance of applications.

Recently, a large number of improvements of TCP have been proposed [1], [6]–[10]. Some work identifies the cause of performance degradation and suggests adjusting existing congestion control (CC) parameters to match the data center network. For instance, Reducing-RTO [6] reduces the minimum retransmission timeout (RTO_{min}) value and reduces unnecessary waiting after packet drops. Others have suggested redesigning CC, using a new lossless RDMA (Remote Direct Memory Access) based network stack, or even designing entirely new data center transmission protocols. For example, DCTCP [1] accurately controls the total throughput through the explicit congestion notification (ECN) identifier provided by the switch to avoid overloading the switch buffer and packet loss. DCQCN [8] is a CC for the lossless network protocol RoCEv2 (RDMA over Converged Ethernet version 2) [11], which uses Priority-based Flow Control (PFC) [12] to avoid buffer overflow by forcing the immediate upstream entity to pause data transmission. NDP [9] redesigns the entire data center transport protocol, including routing and CC, to provide low latency and high throughput.

Although many of the above proposals have proven to be commercially available, they face a great challenge on real-world deployment in public and multi-tenant data centers [13]. This is because that the common physical infrastructures such as switches and network interface cards (NIC) are shared by multiple tenants in the form of virtual machines (VMs). It is the tenants who are able to deploy applications in the VMs, select the corresponding transport layer protocols, and decide end-system protocol stack parameters such as ECN support and RTO_{min} value. Consequently, from the perspective of operators of multi-tenant data centers, when the tenants have already run the VMs, it requires more effort to modify the network protocol stack than to modify the common physical infrastructures (see Section II-B). In this case, simply changing the controllable physical infrastructure without any modifications to the legacy transport protocol stack in order to improve transmission performance transparently would be a contribution for data center operators.

A basic idea of transparently enhancing the transport protocol stack is an intrusive modification to the headers of packets forwarded by switches. For example, HSCC [14] rewrites the value of the receive window (denoted by $rwnd$) to one MSS (Maximum Segment Size) in the ACK headers for all congested flows. These approaches, however, are limited by an artifact of the current window-based transport protocol design (e.g., NewReno [15], CUBIC [16], and DCTCP [1]), in which the window indicates the number of full-sized packets. In other words, $rwnd$ can not be rewritten to a proper fraction (i.e., $rwnd \notin (0,\ 1)$). This coarse granularity significantly limits the scale of concurrency.

In this paper, we present a new mechanism called R-AQM

*Tong Li is the corresponding author (email: li.tong@huawei.com).

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

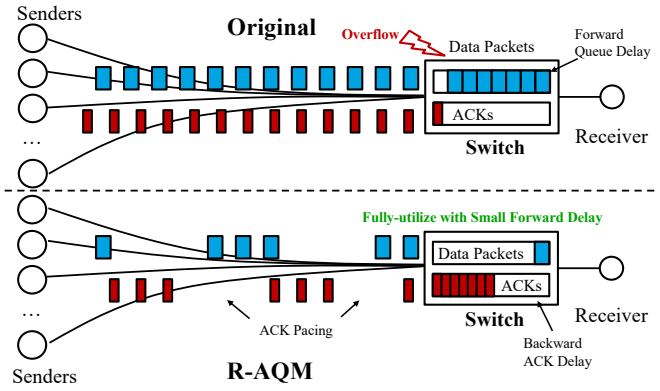


Figure 1. The general idea of R-AQM an illustrative example.

(Reverse Active Queue Management), which is transparent to end-systems and fine-grained. Figure 1 illustrates the general idea of R-AQM. The fundamental premise of R-AQM is ACK-clocking [17], i.e., ACKs not only acknowledge receipts of data packets but also trigger new packet sending. Unlike existing AQM schemes [18], [19] that intrusively modify the content of packets, R-AQM proactively intercepts ACKs to prevent the source from sending the next packet too fast, which also slows down the increase of the sending window. In this way, R-AQM is able to deploy active queue management for ACKs in the reverse path to adjust the in-flight traffic without overwhelming the switch in the case of incast congestion.

The rest of the paper is organized as follows. We introduce the background of the TCP incast problem, the deployment challenges for multi-tenant data centers, and the degradation of goodput by RTO in Section 2. Section 3 illustrates the design rationale of our solution. The detailed design of R-AQM is demonstrated in Section 4. In Section 5, we address the implementation of R-AQM on NetFPGA. In Section 6 and Section 7, we evaluate R-AQM in NS-3 and a small-scale testbed. Section 8 surveys the related work. Finally, Section 9 concludes this paper.

II. Background and Motivation

A. TCP Incast Problem Hurts Transmission Performance

TCP incast is a catastrophic goodput collapse that occurs as the number of servers sending data to a client increases beyond the ability of an Ethernet switch to buffer packets. This scenario often happens intra data center communication when requesting data for file systems [20], during the shuffle phase of cloud computing systems [21], and in the partition/aggregate pattern of large-scale web applications [1]. The synchronous request workload causes packets to exceed the buffer on the bottleneck link, resulting in severe packet losses. Packet loss further causes costly timeout, which lasts for hundreds of milliseconds (varies in different scenarios). As a result, the goodput of the link drops due to wasting opportunities for sending data during the retransmission timeouts. We also give a quantitative analysis on the TCP incast problem below.

Assume that N incast flows with the same RTT are sharing a bottleneck link with the capacity of C. Each flow needs to send X bits, let n be the number of RTTs it takes to complete the transfer of one flow and m be the average RTO times. As illustrated in [14], the average goodput of the link is given by:

$$Goodput = \frac{X}{n \cdot RTT + m \cdot RTO + \frac{X \cdot N}{C}}$$

In the case that RTT = 100μs and RTO = 100ms (the lower bound in the Linux implementations), since RTO is usually three orders of magnitude larger than RTT, it is easy to see that a single time of RTO can lead to a sharp drop in goodput.

B. Deployment Challenges for Multi-tenant Data Centers

To control the TCP incast, data center operators need to upgrade their hardware and (or) software. In private data centers, administrators can not only change the physical infrastructure such as switches and network interface cards (NIC), but also modify the transport protocol stack at end-systems. Therefore, improvements (e.g., Reducing-RTO [6], DCTCP [1], DCQCN [8], NDP [9]) are possible to be deployed by systematically upgrading infrastructures and systems.

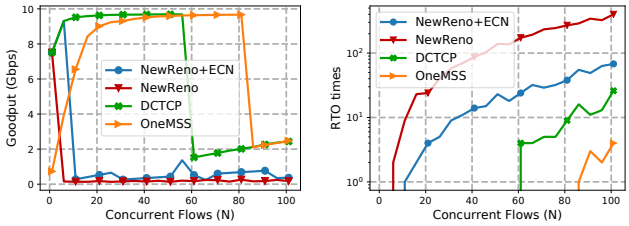
However, in public and multi-tenant data centers, it requires more effort for operators to modify the network protocol stack than to modify the common physical infrastructures. In virtualized multi-tenant data centers [13], [22], the common physical infrastructures are shared by multiple tenants in the form of virtual machines (VMs). Generally, the data center operators deploy a default transport protocol stack in the system image of each VM. It is the tenants who are able to deploy applications or systems in the VMs, select the corresponding transport layer protocols, and decide end-system protocol stacks (e.g., Use BBR [23] between the user and the data center, use DCTCP [1] or NewReno [15] with ECN within the data center) and parameters (e.g., ECN support and RTO_{min} value).

Consequently, from the operators' perspective, it requires extra effort to modify the network protocol stack after tenants have already run the VMs. For example, enabling virtual CC in the hypervisor as specified in prior works such as AC/DC TCP [24] and vCC [25]. Both of which provide congestion agents in the hypervisor that transparently place efficient CCs for tenant VMs. However, these methods require full TCP state tracking and full TCP finite-state machines in the hypervisor, which may overload the hypervisor and slow it down considerably. In addition, since incast usually happens on the last-hop switch, the end-to-end hypervisor-based way may still suffer from incast problems. In other words, simply applying the hypervisor-based solution only solves part of the problems [9].

Based on above observations, we seek a solution that not only works on the incast problems, but also requires no change to the TCP stack at end hosts.

C. Fine Granularity Requirement of Window Control

HSCC is a switch-based congestion controller [14] that rewrites the value of receive window (denoted by $rwnd$) to one MSS (Maximum Segment Size) in the ACK headers for



(a) Goodput in incast (b) RTO times in incast

Figure 2. The experiment to show the incast problem.

all congested flows without modifying TCP itself. However, HSCC cannot cooperate with legacy TCP CCs very well in data centers. Legacy TCP CCs in the Linux Kernel are almost window-based (NewReno [15], CUBIC [16], BBR [23], and DCTCP [1]). These window-based CCs have fundamental flaws in small RTT networks, because they cannot reduce the sending window infinitely (i.e., not less than 1 MSS). Since the bandwidth-delay product (BDP) in a data center netowrk is usually small due to the small RTT, it is very easy for the in-flight packets to become less than the BDP+buffer. In this case, the extra packets can only be dropped or be resent by retransmission. However, when incast occurs, flows may fail to build large enough in-flight packets to recover via fast retransmission (e.g., 3-duplicate ACKs). As a result, this coarse granularity significantly limits the scale of concurrency and we need fine-grained window control.

To motivate the requirement of fine-grained window control, we give a modeling analysis as below. Assume that the window size of flow i at time t is $w_i(t)$, the switch buffer and the link capacity are B and C, respectively. The queue size $q(t)$ in the switch at time t in the case of N incast flows is given by:

$$q(t) = \sum_{i=1}^{N} w_i(t) - C \cdot RTT$$

In the case of a large number of concurrent flows when $N \cdot MSS - C \cdot RTT \geq B$, a considerable proportion of flows may fall back to the stop-and-wait paradigm to avoid packet loss and RTO. That is, there must be some flows stopping sending data and setting the window to zero. This coarse granularity significantly limits the scale of concurrency. Particularly, the number of concurrent flows is limited to 4060 in most modern switches [1], [26], [27]. However, this is not nearly enough to sustain real data center communications. For example, a cluster running data mining tasks have more than 80 concurrent flows per node [28], [29]; In Facebook's Memcached cluster [30], a single Web server may access over 100 Memcached servers. Worse, a production data center with 6000 servers supporting Web search applications has over 1000 concurrent traffic on work nodes [1]. It is obvious that even 1 MSS of sending window per flow is enough to overwhelm the switch buffer on a synchronous burst.

To better understand how does the granularity of window control impacts the scale of concurrency, we further conduct a simulation. Each sender sends a 320KB message to a fixed receiver. Three common CCs (NewReno with ECN, NewReno without ECN, DCTCP) and one particular CC that always sets the congestion window to 1 MSS (similar to HSCC [14]) are investigated. Figure 2 shows the goodput and RTO times with different scales of concurrency. Some insights are listed below:

(1) Even one RTO occurs, the loss of goodput is enormous.

(2) NewReno does not work well. Even with ECN, the number of senders cannot exceed 10.

(3) DCTCP can alleviate the occurrence of incast collapse, but the concurrency can only be maintained around 60.

(4) Even if the sending window is always 1 MSS, only about 80 concurrent flows can be maintained.

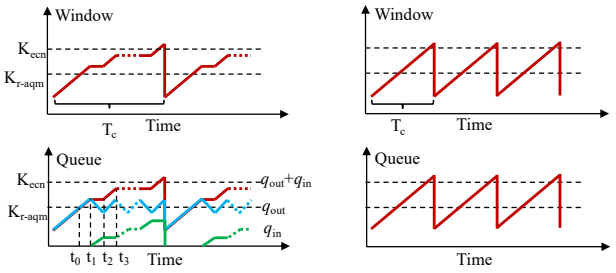
In summary, a more fine-grained window control is needed to solve the incast problem. Meanwhile, switch-based mechanisms have the potential to overcome the deployment challenges for multi-tenant data centers. These greatly motivate the design ideas and principles of R-AQM.

III. DESIGN RATIONALE

Our goal is to design an incast control mechanism in the multi-tenant data center to handle as many concurrent connections as possible effectively. We came up with a new active ACK control approach called R-AQM. The critical factor that inspires the new ACK control approach is that if a sender does not receive an ACK, the sender cannot send the next data packet. If the ACK can be intentionally delayed, the senders' following sending action will also be delayed accordingly. The protocol that relies on the arrival of ACK packets to infer that the network can accept more packets is called window-based ACK-clocking protocol [17], [31].

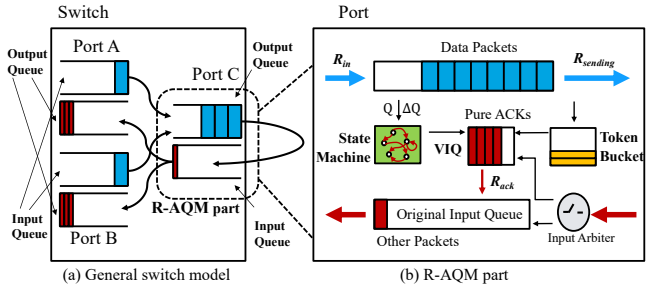
When incast occurs, the switch can proactively intercept the ACK packet in the backward direction and send ACKs at a rate that does not make the ACK-triggered data packet overwhelming the switch. In this way, we can leverage active ACK control to adjust in-flight traffic without being constrained by the minimum window size shared by window-based solutions. We only need to adjust the ACK rhythm appropriately in the switch. Consequently, it is ideal for multi-tenant cloud data center networks. Moreover, because the bottleneck switch can capture the instantaneous queue length, it can sense incast more quickly and thus make decisions more quickly to prevent further congestion.

The rationale of R-AQM is to lower the nontrivial forward data queuing delay by introducing a trivial backward ACK queuing delay. In the incast traffic pattern applications, the forward packet is usually the service request packet (e.g., Reduce in MapReduce [21]), while the reverse packet is usually only the ACK without piggybacked data. Compared with the ACK's backward queuing delay, applications pay more attention to the forward queuing delay of the data packets. Forward delay refers to the time taken by a packet departing from the sender to the receiver, which is very important for the application's QoE. Backward delay in the reverse direction only delays the confirmation of a data packet [32], [33], which does not greatly impact application QoE.



(a) DCTCP with R-AQM (b) DCTCP with ECN

Figure 3. Window Size and Queue Size Process.



(a) General switch model (b) R-AQM part

Figure 4. R-AQM design.

Figure 3 shows the window size and queue size with R-AQM and ECN. We now analyze the steady-state behavior of the R-AQM control loop in a simplified setting to understand how to convert the forward queuing delay to the backward queuing delay. We assume that the N flows are synchronized for convenience of understanding; i.e., their sawtooth window dynamics are in-phase. At time t_0, output queue length (Q_{out}) exceeds K_{r-aqm}, and the switch starts to buffer ACKs actively. From time t_0 to t_1, the senders need time to react to the ACK buffer action on the switch. From time t_1 to t_2, the senders do not receive ACKs, so the window remains unchanged. Meanwhile, the sending window does not change, the total number of packets in the network also does not change, so Q_{in}+Q_{out} remains unchanged. But the input queue length (Q_{in}) begins to grow, and the output queue length (Q_{out}) begins to decline. At time t_2, the source begins to receive ACKs again. R-AQM will repeat the same action to keep Q_{out} at a low level while the extra inflight ACK packets are stored in Q_{in}. Through the above steps, the data queue transforms into the ACK queue, and the forward queuing delay transforms into the backward queuing delay. R-AQM alleviates the excessive window growth rate of DCTCP (Figure 3(b)) without loss of throughput (Q_{out}>0).

With these benefits, the next questions are how to properly hold and send back ACKs in the switch, what problems active ACK interception can cause, how to fix it, and so on. In the next section, we introduce how we solve these problems by proposing R-AQM.

IV. R-AQM

R-AQM is an incast control mechanism that aims to mitigate buffer overflow problems by shaping ACKs in the switch through assisting legacy TCP. Figure 4(b) presents our design framework, which contains three main functional components: the Virtual Input Queue, the Token Bucket, and the State Machine. As shown in Figure 4, packets sent by the sender are queued on the bottleneck port as usual, and each packet a sender sends will be acknowledged by the receiver. (1) When the returned ACK enters the bottleneck port, the VIQ (virtual input queue) located in the switch input port recognizes ACKs, intercepts them and stores them; (2) The Token Bucket monitors the immediate egress sending packets and generates tokens to the bucket to trigger the VIQ dequeue action; (3) The State Machine parses queue length information, calculates the draining rate according to the congestion state. After the sender receives the ACK, the sender adjusts the sending rate and sends the next packet.

In this section, we propose our design by answering the following four questions:

- How to intercept and buffer ACKs?
- What is the ACK dequeue policy in the switch?
- How to determine the draining rate of ACKs?
- What are the side effects, and how to compensate?

A. How to intercept and buffer ACKs?

The first step is to distinguished between different ACKs, based on which the piggybacked ACKs (which can affect the application QoE) and ACKs with the FIN flag (which can not trigger a new data packet either) are excluded. In this paper, we define pure ACKs as ACKs without FIN and ACKs that are not piggybacked, which are determined by the combination of the packet size and the header tag. With the input arbiter, pure ACKs are queued in VIQ, and others are queueing in the original input queue. VIQ is located between the switch input port and the forward core. To better control the pure ACK draining rate, we need to separate the ACKs from other packets, setting up a virtual input queue for pure ACKs. VIQ sets the highest priority of each ACK to prevent delay and packet loss due to reverse-path congestion. Even if the ACK packet size is small, VIQ still needs some memory to store ACKs, so the design needs to consider how to drop packets. When the ACK is dropped, the sender will assume that the data was not received, which wastes forward throughput and might cause RTO.

There are two reasons why we choose to set up a VIQ instead of an ACK output queue. First, ACKs of congested flows should be proactively intercepted. As shown in Figure 4(a), port C's output queue is the congestion point. If the output queues of A and B are proactively intercepted, then the wrong ACKs from other ports may be buffered, affecting non-congested traffic. Second, deploying on an input queue is relatively easier. The input queue can directly obtain the output queue length within the same port, and the changes in operation logic are minimized, which does not affect the top design of the switch.

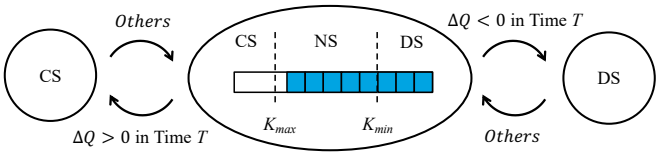


Figure 5. R-AQM state transition diagram

B. What is the ACK dequeue policy in the switch?

A proper switch implementation requires a hardware input queue, and its dequeue action needs to be controlled by a data plane. R-AQM uses the token bucket to trigger the dequeue action. The token bucket is an algorithm for traffic shaping in packet-switched networks. It can be used to check whether data transmission at the packet granularity conforms to the defined limits of bandwidth and burst, which measures the unevenness or variability of traffic. As shown in Figure 4(b), the token bucket controls the token input rate by monitoring the average value of $R_{sending}$. Each increment of a token triggers an enqueue action (not necessarily sending, see Section IV-C). Using the token bucket, on the one hand, we can regulate the sending rhythm of ACKs. On the other hand, the draining rate R_{ack} can be adjusted by controlling the proportion of the ACK consumption token.

C. How to determine the draining rate of ACKs?

Having figured out how to buffer ACKs proactively, we need to figure out when to drain ACKs. We use the State Machine to judge the congestion state and adjust the ACK draining rate. A simple idea is mapping the output queue length directly to the ACK draining rate. The longer the queue, the more severe the congestion and the slower the ACK should be sent. The shorter the queue, the less severe the congestion, and the faster the ACK should be sent. However, such a naive idea suffers from some issues.

First, mapping functions are costly for hardware [5]. The mapping of a linear function is difficult to implement in ASIC or FPGA hardware due to the requirement of division operation. In general, the linear function is approximated by a step function [34], [35]. Second, the feedback latency causes oscillation. Since the network is a pipelined model, when R-AQM buffers ACKs, it does not immediately reduce congestion. Frequent changes in the ACK rate can cause oscillations. Third, queue length does not identify a burst. This is because a low threshold is too sensitive to identify the burst, and a high threshold makes it difficult to identify a burst accurately.

In R-AQM, we use queue length and its gradient to judge the congestion state comprehensively, and only three corresponding states are set, greatly simplifying the design. Figure 5 shows the state transition diagram, which is the most critical part of R-AQM. Compared to general AQMs, it uses variation in queue length to determine the state of congestion. First, it calculates whether the queue length continues to grow or decline in time T. If the queue continues to grow, there will be a burst, so no matter how long the current queue is, it should trigger an active buffer to accommodate burst (left part of Figure 5). If the queue length continues to decrease, it means that the burst has ended. The ACK can be returned at this point, rather than waiting for the queue to decrease to a certain value (right part of Figure 5). When the network state is stable, just like the traditional AQM, it can be determined by the threshold (middle part of Figure 5).

Algorithm 1 VIQ send algorithm. $\boldsymbol{state}$ is the R-AQM state of one port. α and n are the number of tokens consumed and the number of ACKs emitted at each time. α and n control the ACK draining rate.

```
1: function viq_send( )
2:     if state is NS and token ≥ α1 then
3:         token -= α1; VIQ.pop(n1) // Normal State
4:     else if state is DS and token ≥ α2 then
5:         token -= α2; VIQ.pop(n2) // Draining State
6:     else if state is CS and token ≥ α3 then
7:         token -= α3; VIQ.pop(n3) // Congest State
8:     end if
9: end function
```

Algorithm 1 illustrates the process of the ACK send action in the switch. Generating a token in the token bucket triggers the procedure $viq\_send()$ at Line 1. There are three states to represent the different actions, namely Congest State (CS), Draining State (DS), and Normal State (NS). We use α to represent the number of tokens consumed and n to represent the number of ACKs emitted at each time. NS is the steady-state of the switch and requires only a uniform ACK response. In NS, $\frac{\alpha_1}{n_1} = 1$ (Line 2-3). DS indicates that the forward queue is about to empty, so we need to speed up emptying the reverse ACK queue. In DS, $\frac{\alpha_2}{n_2} < 1$ (Line 4-5). CS means extreme congestion. In CS, $\frac{\alpha_3}{n_3} >> 1$ (Line 6-7). R-AQM needs to ensure that the ACK is sent at a low rate, but not stopped. First, it avoids RTO caused by senders that do not receive any ACKs for a long time. Second, it prevents some of the flows from starvation in the case of burst congestion.

D. What are the side effects on TCP, and how to compensate?

Interaction with TCP RTO:

One concern of R-AQM is its interaction with TCP RTO. R-AQM limits the rate of ACK in order to prevent RTO caused by packet loss, so it is inevitable to increase RTT. It is not sure whether this will cause RTT to be prolonged beyond RTO_{min}, leading to TCP timeouts and spurious retransmissions. For this reason, we specifically measure the RTTs in our experiments. We find that our ACK control does not adversely prolong the RTTs (for example, with 200 connections, the 99th percentile RTT is less than 0.3ms). And we do not observe any spurious retransmission.

Even though this phenomenon is rare, we still take into account the possibility and design counter-measures. As each pure ACK enters the switch, R-AQM records the time stamp in the auxiliary packet header. When each pure ACK exits the switch, R-AQM makes a judgment that if there is more than

5ms (which is the recommended as the smallest RTO_{min} in [6]), it is considered as an old ACK (which may be caused by TCP timeout and retransmission), and will be dropped. The reason is that by dropping the out-of-order ACKs, R-AQM avoids disturbing the TCP at the sender for subsequent unnecessary retransmissions.

Interaction with TCP CC:

Another concern of R-AQM is its interaction with TCP CC. R-AQM takes effect before packet loss and ECN trigger. Therefore, when R-AQM senses congestion, it not only needs to delay the ACK transmission, but also needs to prevent the sender from increasing the sending window.

Two mechanisms are recommended to compensate. The first is a BECN-like mechanism [36] that directly marks the ECN-Echo in the TCP packet header of the ACK in the switch when there exists congestion. The senders therefore can use ECN to reduce the sending window, avoiding congestion quickly. Second, by considering packet loss as the congestion signal [37], it is recommended to adopt a mechanism similar to HSCC [14] that directly sets the ACK headers rwnd to 1 in the switch when incast congestion occurs.

With these two mechanisms, R-AQM works well with existing CCs in the Linux kernel (NewReno [15], CUBIC [16], and DCTCP [1]). It can also coexist with different CCs and TCP settings, while R-AQM limits the sending window (rwnd by HSCC) to 1 MSS, therefore treating each sender fairly.

E. Discussions

Parameters Guidance: According to Algorithm 1, nine parameters of R-AQM need to be set. K_{max} and K_{min} represent the maximum and minimum values in the steady state, respectively, and T stands for burst duration. $\alpha_1, \alpha_2, \alpha_3$ and n_1, n_2, n_3 control the ACK draining rate in different state. The setting of these parameters is a trade-off. A small value of K indicates that R-AQM will be triggered when the forward queue is small, which will lead to low end-to-end latency but also low throughput. In order to keep high throughput, we need to make sure there are always packets in the forward queue. Therefore, K_{min} can be set to 0.5-1.0 times BDP, K_{max} can be set to 2 times BDP, and T can be set to 0.2-0.6 times RTT, which depends on the CC algorithm and the traffic workload in datacenters. As discussed in section IV-C, $\frac{\alpha_1}{n_1} = 1$, $\frac{\alpha_2}{n_2} < 1$ and $\frac{\alpha_3}{n_3} >> 1$, this paper suggests $\frac{\alpha_2}{n_2} = 0.5$ and $\frac{\alpha_3}{n_3} = 10$.

Different ACK Mechanism: R-AQM assumes that the tenants are using per-packet ACK (one incoming packet triggers one ACK), which provides more precise control [35], [38]. However, tenants may need to modify their ACK mechanism, such as enabling Delayed ACK [39] so that an ACK can trigger more than one data packet at a time. Hence data center operators should encourage tenants to use the default TCP stack to get better performance. In the worst case, this phenomenon can still be alleviated by adjusting K_{max}, K_{min} and α.

Symmetric Routing Dependency: R-AQM by design requires the ACKs to return on the same backward path as the data flows through. This requirement can be easily met given the common deployment of ECMP routing in data centers [14], [28], [38].

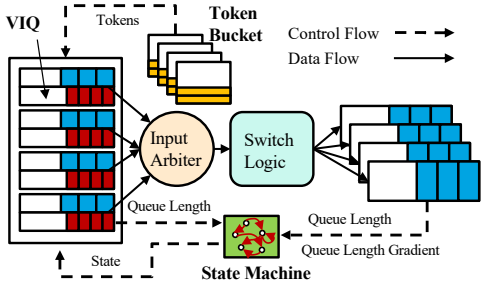


Figure 6. R-AQM switch architecture on NetFPGA-SUME

Non-ACK-clocking Cases: R-AQM cannot work with UDP and non-ACK-clocking CCs (like BBR). Although 99% of flows in data centers use TCP [1], R-AQM has no effect on UDP flows. In this case, the administrator can reserve a small portion of the switch bandwidth exclusively for UDP, while TCP with R-AQM monopolizes most bandwidth. BBR [23] is usually used for transmission in the wide area network. R-AQM, designed for intra data center, has little effect on BBR.

Elephant Flow and Mice Flow: R-AQM's motive is to address the TCP incast problem. Thus, like CC, R-AQM only controls elephant flows and has little control over mice flows, with only a few ACKs. However, R-AQM can reduce the FCT of mice flows. Most mice flows can end up in a single sending window, and R-AQM makes the forward queue length very small, so mice flows can pass quickly.

Maximum support senders and Memory usage: There is an upper limit to the number of R-AQM concurrency, which depends on the use of two pieces of memory (VIQ and the regular output queue). R-AQM controls the senders sending action by shaping the ACKs, so it cannot reduce packet loss during the first control loop only by increasing the output queue memory. When traffic is stable, the number of senders depends on the VIQ size. Because the ACK is smaller than the data packet (60B v.s. 1460B), R-AQM can support more senders than traditional methods.

V. Implementation

Ideally, the R-AQM's Token Bucket and the State Machine would be implemented in switch ASICs. We build a prototype of such a solution using the NetFPGA-SUME platform [40], a programmable hardware platform. It has four 10Gb/s Ethernet interfaces and a Xilinx Virtex-7 FPGA with QDRII+ and DDR3 memory resources.

Figure 6 shows the top design of the R-AQM switch in NetPFGA. Packets enter one of the 10Gb/s interfaces and are stored in a regular input queue or VIQ. VIQ is allocated 36KB of SRAM, which separates from the regular input queue. ACKs are recognized while entering the input port. Pure ACKs enter VIQ, and others enter the regular input queue. The input arbiter takes packets from the input queues using a round-robin

(RR) scheduling policy and feeds them to the L2 switching logic via a 256bit-wide 200MHz bus, which is fast enough to support more than 40Gb/s. The token bucket is used to trigger the VIQ sending action. The state machine determines the network congestion and adjusts the ACK draining rate through the queue length and its gradient. Pure ACK requires tokens and enters the input arbiter. Other packets can directly go to the input arbiter. After a conventional L2 forwarding decision is made, the packet reaches output queues.

Clearly, R-AQM implementation is quite simple, and the processing delay at the switch is very small. R-AQM does not operate normally for every packet, because it is only triggered to avoid packet dropping. Therefore, the additional processing delay at the switch is not introduced frequently. In addition, R-AQM introduces only a little resource consumption on switches. The R-AQM switch uses 58,610 LUTs (14% of the Virtex7s capacity), 29,370 FlipFlops (10%), and 1,470 blocks of RAM (45%). In comparison, the ECN-version FPGA switch uses 12%, 9%, and 40% respectively, so the complexity added by R-AQM is quite small.

We conclude that the implementation complexity, processing delay, and resource consumption of R-AQM are acceptable; thus, R-AQM can be built into commercial switches.

VI. Simulations

In this section, we conduct a simulation analysis of R-AQM performance using NS-3 [41]. Specifically, we evaluate three critical aspects of R-AQM as follows: (1) The flow scalability of R-AQM in terms of goodput, drop times, RTO times, latency and queuing. (2) The incast reaction details of R-AQM, concurrency, sensitivity and fairness analysis of R-AQM. (3) The effectiveness of R-AQM under all to all traffic.

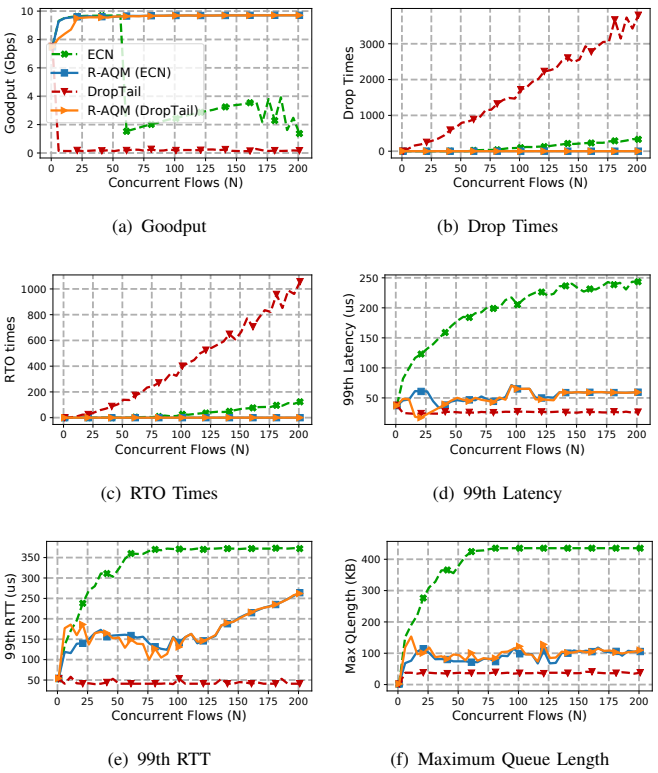
A. Settings

The topology in the NS-3 simulations is a FatTree [42]. There are 16 Core switches, 20 Aggregation switches, 20 ToRs (Top-of-Rack switches) and 320 servers (16 in each rack), and each server has a single 10Gbps NIC connected to a single ToR. The capacity of each link between Core and Aggregation switches, Aggregation switches and ToRs are all 40Gbps. All links have a 1μs propagation delay, which gives a 12μs maximum base RTT. The switch per port's buffer is 300 packets (or about 400KB) derived from real device configurations.

We use two standard AQMs as baselines, ECN and droptail. We list the terminologies below:

- **ECN**: The corresponding senders' CC for ECN uses DCTCP.
- **R-AQM (ECN)**: The corresponding senders' CC for ECN uses DCTCP and the switch deploys R-AQM.
- **DropTail**: The corresponding senders' CC uses TCP NewReno.
- **R-AQM (DropTail/DropT)**: The corresponding senders' CC uses TCP NewReno and the switch deploys R-AQM.

The relevant parameters are set as follows: For R-AQM, we set K_{min} to 20 packets, K_{max} to 40 packets, T to 3μs,



(a) Goodput (b) Drop Times

(c) RTO Times (d) 99th Latency

(e) 99th RTT (f) Maximum Queue Length

Figure 7. Goodput, Drop times, RTO times, Latency, RTT and Queue Length with many concurrent flows

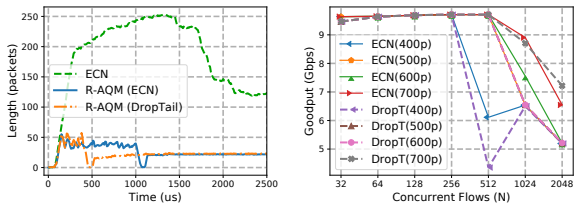
$\frac{\alpha_1}{n_1} = 1$, $\frac{\alpha_2}{n_2} = 0.5$ and $\frac{\alpha_3}{n_3} = 10$ in Algorithm 1. The switch VIQ is 500 packets, about 23KB. TCP is set to the default TCP RTO_{min} of 100 ms. For the ECN threshold, we scale the ECN threshold proportional to the link bandwidth. We set $ECNK_{min}$ = $ECNK_{max}$ = 65 packets according to [1].

B. Incast

In incast, N hosts send a 3.2MB flow to a host. We vary the number of flows from 1 to 200. Figure 7 shows the results.

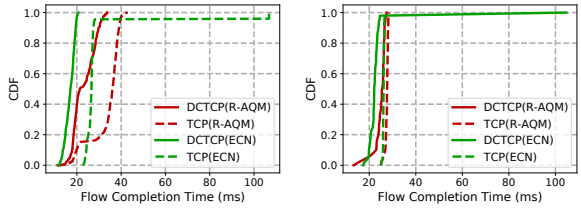
Goodput: First, we measured the goodput. In general, R-AQM can easily handle 200 concurrent connections without seeing any trend in performance degradation, while Droptail and ECN begin to downgrade when the numbers of connections exceed 5 and 60, respectively. When the number of connections is small, DCTCP with R-AQM shows little advantage over TCP with R-AQM in goodput (a few Gbps). For TCP, R-AQM only limits the sending window by limiting rwnd, so utilization decreases when the number of senders is small. However, R-AQM can continue to achieve near 9.8 Gbps goodput with an increasing number of connections.

Packet Drops and RTO: Second, we measured the drop times and RTO times. As shown in Figure 7(b) and 7(c), when the concurrency value is less than 200, R-AQM does not lose packets and trigger RTO. This also explains why R-AQM has no goodput loss. However, in the case of ultra-high concurrency, it still causes packet loss and RTO, which will be analyzed later.



(a) Reaction Details Analysis of R-AQM (b) Sensitivity Analysis of R-AQM

Figure 8. Reaction Details and Sensitivity Analysis of R-AQM



(a) 10-to-1 incast (b) 100-to-1 incast

Figure 9. Fairness of different CCs

Latency and RTT: Third, we measured forward one way latency and RTT. It can be seen that the latency can be significantly reduced with R-AQM. R-AQM achieves from 4.6× to 7.5× lower 99th latency compared to ECN. Droptails latency is very low because its goodput is low, and the switch cannot be utilized effectively. As the number of concurrent connections increases, the 99th RTT increases but the 99th latency almost remains unchanged, indicating that R-AQM can keep the forward delay at a low value regardless of the number of concurrent flows. We also find that R-AQM delivers little impact on RTT. For example, when there are 200 concurrent connections, 99% of them are less than 0.3ms. Currently, many production data centers have reduced RTO_{min} to a low value (e.g., 10ms [1]). In Linux, the lowest possible RTO value is 5 jiffies (5ms) [6]. This suggests that R-AQM can work smoothly and will not result in issues like spurious timeouts and retransmissions in production datacenters with low RTO_{min}.

Queue Length: Fourth, we measure the switch buffer use of R-AQM. Figure 7(f) shows that the buffer occupation of R-AQM is much lower than that of ECN. Moreover, as the number of senders increases, the buffer grows less significantly, while the ECN fills up when the sender is 70. In combination with Figure 7(d) and 7(e), with the sender number increasing, latency is unchanged while RTT is increasing, indicating that the reverse latency is increasing. This proves that the forward queue length is unchanged while the reverse VIQ is increasing all the time. The increase of ACK numbers makes little use of the buffer, so the utilization of the buffer can be reduced.

Response Details of R-AQM: Fifth, we analyzed R-AQM response details and found why it could alleviate incast and reduce latency. Figure 8(a) shows the change in the queue length of the bottleneck switch overtime where N=36. R-AQM's queue starts to drop as it grows to 50 packets, while the ECN needs to grow to 250 before it can be adjusted. Fast adjustment of queue length can avoid more packet loss and reduce more RTO times. Moreover, R-AQM converges fast, which converges before 1000μs while ECN converges after 2000μs. Also, R-AQM negative feedback regulates queue length, maintaining a short queue length.

Concurrency and Sensitivity Analysis of R-AQM: To explore the maximum number of connections that R-AQM can handle, we fix the total traffic volume and gradually increase the number of senders. We also repeat the simulation experiment using different parameters to assess the sensitivity of R-AQM to the setting of parameters. The results show that goodput is affected by the choice of the parameter VIQ length. Other parameters (omitted due to space) are not very sensitive to goodput. From Figure 8(b), we find that R-AQM can easily support more than 500 concurrent connections and sustain near 9Gbps goodput when facing 1000 senders. The goodput loss of 400p in R-AQM with 256 senders was due to VIQ dropping the ACK. As VIQ's Buffer grows, it can support up to 512 nodes, but not more. As discussed earlier, the reason is that excessive concurrency causes forward data packet loss in the first control loop.

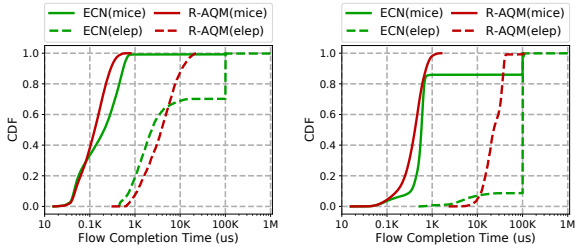
Fairness: Multi-tenant data centers might provide different CCs in use at the same time, so we also explore the fairness of hosts using different CCs. We observed the FCT distribution of different CCs' flow in the incast scenario. We discuss two scenarios, 10-to-1 incast and 100-to-1 incast, in which 50% of hosts use DCTCP, and the other 50% of hosts use TCP. In the 100-to-1 incast, the flow size is 3.2MB. In the 10-to-1 incast, to keep the time scale the same, we use 32MB flows. Figure 9 shows the results. In the 100-to-1 incast, R-AQM can reduce the gap between the two CCs. In the 10-to-1 incast, R-AQM can reduced the gap in 99th FCT. We believe this is because that R-AQM avoids RTO (abnormal state) effectively and different CCs behave similarly in normal state.

C. All to All

The all-to-all traffic patterns commonly happen in the shuffle step of MapReduce [21], which generates incast towards each host running a task. We simulate an all-to-all workload using NS-3. We select three machines on each rack, a total of 3×20=60 machines. Each machine sends a 500KB elephant flow and 50 5KB mice flows to the other 59 machines. So each machine sends and receives 3,540 (60×59) elephant flows and 177,000 (60×59×50) mice flows.

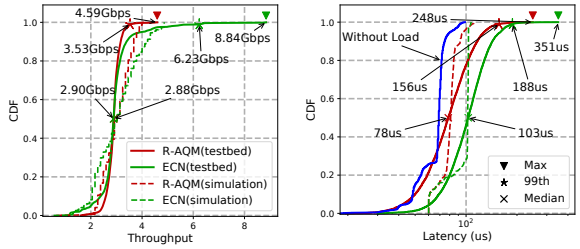
Figure 10 shows the CDF of the flow completion time under different loads. Because Droptail results are unsatisfactory, which causes many RTO, it is omitted here.

At the load of 40% scenarios, with R-AQM, the completion time of the mice flows is reduced. The 50th percentile FCTs are reduced from 208μs to 128μs and the 99th percentile FCTs are reduced from 768μs to 466μs in R-AQM. In ECN, 20% of the flow timeout, while in R-AQM there is no flow timeout. ECNs require queues long enough to trigger, so there is a lot



(a) 40% avg. load (b) 80% avg.load

Figure 10. Shuffle workload



(a) Throughput of CDF (b) Latecny of CDF

Figure 11. Latency and Throughput of CDF in the incast scenario.

of traffic resulting in RTOs due to packet loss, resulting in very long tail completion times. At 40% load, the R-AQM's mice-flow completion time was lower than that of the ECN, with no timeouts. This can prove that R-AQM can maintain a short forward queue, significantly improving the mice-flow application experience.

At the load of 80% scenarios, more than 15% of mice flows and 80% of elephant flows timeout in ECN, and only a small number of elephant flow timeouts in R-AQM. Even though the network load is very high, making ECN almost unusable, R-AQM guarantees that mice flows will not be RTO.

Through the above experiments, on the premise of avoiding RTO as much as possible, R-AQM can also provide a low latency for mice flows, which is enough to show that R-AQM can effectively alleviate the incast problem.

VII. Testbed Experiments

In this section, we conduct a testbed experiment to validate the performance of R-AQM on the small-scale network. We verify R-AQM's ability to mitigate packet loss and low latency in a small-scale scenario through two typical workloads.

A. Settings

The topology of the testbed experiment mimics a small rack of the datacenter. The testbed includes one ToR NetFPGA switch and four servers connected via four 10Gbps links. Each server is equipped with a single Intel Xeno CPU and two dual-port Intel 82599 10G NICs. The CC algorithm at the hosts is DCTCP. TCP's results are similar and are therefore omitted later. On the NetFPGA switch, we also implemented the ECN version for comparison. The relevant parameters are set as follows: For R-AQM, we set K_{min} to 20 packets, K_{max} to 40 packets, T to 3μs, $\frac{\alpha_1}{n_1} = 1$, $\frac{\alpha_2}{n_2} = 0.5$ and $\frac{\alpha_3}{n_3} = 10$ in Algorithm 1. For the ECN threshold, R-AQM and pure ECN both set the 65 packets according to [1], and the switch output drop threshold is 300 packets.

B. Incast

We run a 3-to-1 incast: the frontend application on one host sends requests to another three servers. Upon receiving the request, each server replies with continuous elephant flow immediately. Meanwhile, a 1KB test mice flow is sent every second, which is used to measure the end to end latency. We calculate throughput and latency per second for each computer NIC port. At the same time, we apply the same scenario in the simulation to validate the testbed.

Figure 11 shows the throughput and latency of ECN and R-AQM in both simulation and testbed. We can see that the simulation experiment results are similar to the testbed experiment. For the 50th percentile throughput, R-AQM performs almost equally well compared to ECN, with the throughput of 2.88Gbps for ECN and 2.90Gbps for R-AQM. ECN has a long tail of 8.84Gpbs, which means an extremely unfair distribution of throughput could occur, with one port higher and the other two lower. The reason is that when a flow suffers from RTO caused by packet loss, other traffic takes up the bandwidth. In contrast, R-AQM shows a more even distribution of throughput and provides better fairness.

For latency, the 50th percentile and the worst case of R-AQM are 25μs and 103μs, respectively, both of which are lower than ECN. The 50th percentile of one-way delay for R-AQM is approximately 78μs, which is just slightly bigger (10μs) than the optimum transfer time in an idle network (67.7μs, labeled as without load in the figure). The improvement in latency by R-AQM is not outstanding due to the small scale of the testbed and the fact that the system kernel latency occupies a certain proportion of the overall latency. Despite that, it has been proved that R-AQM is effective in reducing latency.

C. All to All

To further demonstrate the functionality of R-AQM, we performed an all to all traffic pattern. Each host sends requests to the other three at the same time. The reply traffic's load varies from 10% to 90%. Meanwhile, a 1KB test mice flow is sent every second, which is used to measure the end to end latency. As shown in Figure 12, with the increase of load before 40%, the latency of ECN increases linearly, while R-AQM does not change. At 40%, R-AQM achieves 1.06× faster average latency than ECN, and the gap is more significant at the 99th percentile. This shows that R-AQM can effectively reduce end-to-end latency. Over 40%, the switch cannot handle more data, and packet loss begins, so the latency for both ECN and R-AQM starts to decrease. However, it is evident from the figure that R-AQM can keep the loss rate lower. Compared with ECN, R-AQM achieves up to 3.22 lower packet loss rates.

To conclude from our experiment, R-AQM effectively tames incast problems, decreases packet loss rate, reduces latency,

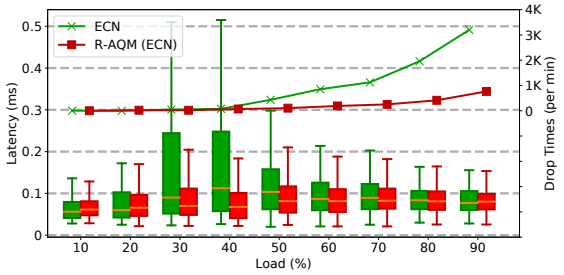


Figure 12. Latency and Drop Times in the all to all scenario (testbed).

and provides better fairness.

VIII. Related Work

Many proposals address the incast congestion problem in data centers. Nevertheless, there has been relatively little effort to address incast in multi-tenant data centers. AC/DC TCP [24], vCC [25], DCTCP [1] and HSCC [14] can be used in multi-tenant data centers, but as discussed earlier, when incast arrived, even all senders' send windows are 1 MSS, the concurrency would not support much.

The most related works to R-AQM is PAC [27], which controls the sending rate of ACKs on the receiver to prevent incast congestion. We clarify the differences between R-AQM and PAC in three aspects. (1) PAC is an end-hosts mechanism, while R-AQM is a switch mechanism. (2) PAC cannot accurately predict the incast because the incast often occurs on the last-hop switch. R-AQM can directly obtain the queue length of the last-hop switch and take effect in advance. (3) It is difficult to deploy PAC in multi-tenant data centers because it requires each tenant to modify or update its kernel.

In addition to the above works, a number of other data center transport designs have emerged, although their primary design space is not suitable for multi-tenant data centers.

Congestion control in private datacenter:

ICTCP [26] handles incast by adaptively adjusting the receiver sides receive window to throttle aggregate throughput. Tuning ECN [43] accelerates the delivery of congestion notifications using dequeue marking instead of traditional enqueue marking. D2TCP [44] adds deadline-awareness at the top of the DCTCP. It adjusts the congestion window to meet the deadline based on congestion conditions and deadline information. ExpressPass [38] uses credit packets for the preallocation of bandwidth to avoid congestion and to guarantee bounded queues.

DCQCN [8] and TIMELY [45] are proposed as the new end-to-end CC scheme designed for RDMA over Converged Ethernet v2 (RoCEv2) [11]. RoCEv2 enables lossless networks through Priority-based Flow Control (PFC), so there is no problem with large-scale RTO and goodput degradation. HPCC [35] also is a RoCEv2 CC that uses switch INT (in-network telemetry) to obtain the precise switch congestion state and calculates the remaining bandwidth. However, incast can also cause other congestion problems, such as PFC storm [35] and PFC deadlock [46], [47], resulting in high latency and unusable network.

All of the above approaches may face deployment issues in multi-tenant data centers and is out of the design scope of R-AQM. However, there is a theoretical possibility that R-AQM can be incrementally deployable with these approaches.

Switch-assisted mechanisms and CCs:

QCN [48] sends the quantized value of the congestion metric as feedback to senders, requiring fine parameters adjustment. PFC [12] allows switches to avoid buffer overflows by forcing the direct upstream switch or NIC to suspend data transfers. XCP [49] and RCP [50] use explicit feedback to measure the extent of congestion. D3 [51] achieves explicit rate control based on deadline information to guarantee deadlines. HULL [52] uses phantom queues to simulate a network at less than 100% utilization and relies on ECN to deliver congestion information. CP [53] and NDP [54] realizes fast packet loss notification by cutting packet payload in the switch and sending packet header back to the sender quickly.

These approaches share the same idea that switches cooperate with congestion control through congestion signals (packet loss, ECN, RTT, INT, etc.). However, most of them require an intrusive modification to the protocol stack. In contrast, R-AQM is an incremental and transparent design for end-systems, helping to alleviate the incast problem and gaining marginal benefit in terms of concurrency.

IX. Conclusion and Future Work

In this paper, we present R-AQM, a transparent reverse ACK active queue management design for multi-tenant data centers to tame the TCP incast problem through active ACK control. The basic principle of R-AQM is to reduce the nontrivial forward queue delay by introducing a trivial backward ACK delay. The critical design idea behind R-AQM is to proactively intercept the ACK in the switch and release it at a moderate rate to prevent too fast new packets from overwhelming the switch. R-AQM set up VIQs to buffer the ACK and use the Token Bucket to shape the flow. R-AQM also uses queue length and its gradient to judge burst and congestion. Our extensive simulations and experiments have shown that R-AQM can enhance existing CC solutions by supporting 16 times more senders and reducing forward queue delay by 4.6 times. One of the limitations of R-AQM is that it can only cooperate with window-based ACK-clocking protocols. As a part of our future work, we are modifying the rate-based DCQCN and TIMELY to some extent so that R-AQM can help alleviate the incast problem and provide a low-latency data center network.

X. Acknowledgement

This work is not possible without the efforts of Xinping Chen and Shengjun Chen. We are grateful for conversations with and feedback from Binzhang Fu and Jincheng Bao. We also thank the reviewers and the shepherd for their valuable comments. This work is supported by China National Funds for Distinguished Young Scientists with No. 61825204, NSFC Project with No. 61932016, and Beijing Outstanding Young Scientist Program with No. BJJWZYJH012019100003011.

REFERENCES

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[2] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 123–137.

[3] J. Zhang, F. Ren, and C. Lin, "Modeling and understanding tcp incast in data center networks," in *Proceedings of IEEE INFOCOM*, 2011, pp. 1377–1385.

[4] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive understanding of tcp incast problem," in *Proceedings of IEEE INFOCOM*, 2015, pp. 1688–1696.

[5] T. Li, K. Wang, K. Xu, K. Yang, C. S. Magurawalage, and H. Wang, "Communication and computation cooperation in cloud radio access network with mobile edge computing," *CCF Transactions on Networking*, vol. 2, no. 1, pp. 43–56, 2019.

[6] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller, "Safe and effective fine-grained tcp retransmissions for datacenter communication," *ACM SIGCOMM computer communication review*, vol. 39, no. 4, pp. 303–314, 2009.

[7] X. Du, K. Xu, T. Li, K. Zheng, S. Fu, and M. Shen, "Traffic control for data center network: State of the art and future research (in chinese)," *Chinese Journal of Computers*, vol. 43, no. 17, pp. 1–23, 2020.

[8] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, 2015.

[9] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for rdma," in *Proceedings of ACM SIGCOMM*, 2018, pp. 313–326.

[10] L. Xu, K. Xu, T. Li, K. Zheng, M. Shen, X. Du, and X. Du, "Abq: Active buffer queueing in datacenters," *IEEE Network*, vol. 34, no. 2, pp. 232–237, 2020.

[11] I. T. Association, "Rocev2. https://cw.infinibandta.org/document/dl/7781, september 2014."

[12] IEEE, "802.11qbb. priority based flow control," 2011.

[13] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson *et al.*, "Network virtualization in multi-tenant datacenters," in *Proceedings of USENIX NSDI*, 2014, pp. 203–216.

[14] A. M. Abdelmoniem and B. Bensaou, "Hysteresis-based active queue management for tcp traffic in data centers," in *Proceedings of IEEE INFOCOM*, 2019, pp. 1621–1629.

[15] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 6582, Apr. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6582.txt

[16] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks," RFC 8312, Feb. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8312.txt

[17] K. Jacobsson, L. L. Andrew, A. Tang, K. H. Johansson, H. Hjalmarsson, and S. H. Low, "Ack-clocking dynamics: Modelling the interaction between windows and the network," in *Proceedings of IEEE INFOCOM*, 2008, pp. 2146–2152.

[18] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on networking*, vol. 1, no. 4, pp. 397–413, 1993.

[19] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin, "Rem: Active queue management," *IEEE network*, vol. 15, no. 3, pp. 48–53, 2001.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of IEEE MSST*, 2010, pp. 1–10.

[21] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[22] K. Xu, T. Li, H. Wang, H. Li, W. Zhu, J. Liu, and S. Lin, "Modeling, analysis, and implementation of universal acceleration platform across online video sharing sites," *IEEE TSC*, vol. 11, no. 3, pp. 534–548, 2018.

[23] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *ACM Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[24] K. He, E. Rozner, K. Agarwal, Y. Gu, W. Felter, J. Carter, and A. Akella, "Ac/dc tcp: Virtual congestion control enforcement for datacenter networks," in *Proceedings of ACM SIGCOMM*, 2016, pp. 244–257.

[25] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, "Virtualized congestion control," in *Proceedings of ACM SIGCOMM*, 2016, pp. 230–243.

[26] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "Ictcp: Incast congestion control for tcp in data-center networks," *IEEE/ACM transactions on networking*, vol. 21, no. 2, pp. 345–358, 2012.

[27] W. Bai, K. Chen, H. Wu, W. Lan, and Y. Zhao, "Pac: Taming tcp incast congestion using proactive ack control," in *Proceedings of IEEE ICNP*. IEEE, 2014, pp. 385–396.

[28] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4, 2009, pp. 51–62.

[29] K. Xu, L. Lv, T. Li, M. Shen, H. Wang, and K. Yang, "Minimizing tardiness for data-intensive applications in heterogeneous systems: A matching theory perspective," *IEEE TPDS*, vol. 31, no. 1, pp. 144–158, 2019.

[30] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Proceedings of USENIX NSDI*, 2013, pp. 385–398.

[31] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4, 1988, pp. 314–329.

[32] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan, "Tack: Improving wireless transport performance by taming acknowledgments," in *Proceedings of ACM SIGCOMM*, 2020, pp. 15–30.

[33] T. Li, K. Zheng, and K. Xu, "Acknowledgment on demand for transport control," *IEEE Internet Computing*, vol. 25, no. 2, pp. 109–115, 2021.

[34] K. Qian, W. Cheng, T. Zhang, and F. Ren, "Gentle flow control: avoiding deadlock in lossless networks," in *Proceedings of ACM SIGCOMM*, 2019, pp. 75–89.

[35] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: high precision congestion control," in *Proceedings of ACM SIGCOMM*, 2019, pp. 44–58.

[36] S. Floyd, "Tcp and explicit congestion notification," *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 5, pp. 8–23, 1994.

[37] L. Li, K. Xu, T. Li, K. Zheng, C. Peng, D. Wang, X. Wang, M. Shen, and R. Mijumbi, "A measurement study on multi-path tcp with multiple cellular carriers on high speed rails," in *ACM SIGCOMM*, 2018, pp. 161–175.

[38] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proceedings of ACM SIGCOMM*, 2017, p. 239252.

[39] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan, "Revisiting acknowledgment mechanism for transport control: Modeling, analysis, and implementation," *IEEE/ACM Transactions on Networking*, pp. 1–15, 2021.

[40] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[41] "Network simulator 3. (2019). https://www.nsnam.org/." 2019.

[42] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, 2008, pp. 63–74.

[43] H. Wu, J. Ju, G. Lu, C. Guo, Y. Xiong, and Y. Zhang, "Tuning ecn for data center networks," in *Proceedings of ACM CoNEXT*, 2012, pp. 25–36.

[44] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.

[45] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "Timely: Rtt-based congestion control for the datacenter," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 537–550, 2015.

[46] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *Proceedings of ACM SIGCOMM*, 2016, pp. 202–215.

[47] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen, "Tagger: Practical pfc deadlock prevention in data center networks," in *Proceedings of ACM CoNEXT*, 2017, pp. 451–463.
[48] IEEE, "802.11qau. congestion notification," 2010.
[49] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *ACM SIGCOMM computer communication review*, vol. 32, no. 4, 2002, pp. 89–102.
[50] N. Dukkipati, *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.
[51] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011, pp. 50–61.
[52] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of USENIX NSDI*, 2012, pp. 253–266.
[53] P. Cheng, F. Ren, R. Shu, and C. Lin, "Catch the whole lot in an action: Rapid precise packet loss notification in data center," in *Proceedings of USENIX NSDI*, 2014, pp. 17–28.
[54] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of ACM SIGCOMM*, 2017, pp. 29–42.