ABQ: Active Buffer Queueing in Datacenters

Lei Xu, Ke Xu, Tong Li, Kai Zheng, Meng Shen, Xiaojiang Du, and Xinle Du

Abstract

Congestion control is always an active research field for guaranteeing the Quality of Services (QoS) of datacenter networks (DCNs). However, the current end-to-end TCP design enables both senders and receivers not to directly and precisely obtain the congestion information, incurring inaccurate or non-real-time adjustments. Moreover, switches that act as fundamental primitives in DCN are usually used for reactive congestion feedbacks (e.g., Ack and ECN), leading to underused for proactive congestion control. In this article, we propose a novel active queue management (AQM) scheme called Active Buffer Queueing (ABQ) which leverages active buffer queueing of DCN switches to achieve both high-throughput and loss-free transmissions. When the traffic pattern in DCN is changed intensely in flow size, number and interval, the feedback information to TCP end hosts are imprecise. Unlike other AQMs, the key idea of ABQ is to adjust the flow rate (or packet pace) directly in the switch according to the real-time congestion state. We explain the design rationale behind ABQ and present simulation results of its performance. Finally, we discuss the implementation on the NetFPGA.

Introduction

In this article, we design a novel Active Queue Management (AQM) scheme called Active Buffer Queueing (ABQ) that employs active buffer queueing for congestion control with the following key properties:

- Keep buffer small: It attempts to match sender rates to network capacity while keeping buffers small, regardless of the number of senders.
- Proactive queueing: Switches can proactively buffer packets according to congestion state in order to adjust flow rate (or packet pace).

The first feature implies that, like all AQM schemes, high utilization is not achieved by keeping large backlogs in the network, but by feeding back the right information for users to set their rates. We present simulation results which demonstrate that ABQ can maintain high utilization with negligible loss or queuing delay as the load increases.

The second feature is essential for ABQ in DCNs where the traffic pattern is changed intensely in flow size, number and interval. The second feature is the fundamental difference between ABQ and other AQMs. Most schemes simply emphasize how to mark explicit congestion notification (ECN) better, but ABQ tries to enable the switch buffer packets to change the flow rate (or packet pace).

In the following, we describe ABQ and explain why it works effectively in DCNs. It will become clear that these features are really useful in DCNs when we present its mechanisms. ABQ can be implemented in real switches. We then compare the performance of ABQ with that of DropTail and ECN in DCNs through simulations. DropTail is with the TCP sender, ECN and ABQ are with the DCTCP sender. From our experiment, we find TCP and DCTCP still drop packets, making retransmission timeout (RTO) frequently. We explain how ABQ can help address this problem and present simulation results of its performance.

The primary intention of ABQ is that congestion should be mitigated where it happens. Current switches do not support proactive queueing. ABQ makes a small change in switches and can relieve congestion a lot. To the best of our knowledge, the proposed ABQ in this article is the first scheme that uses switches to adjust flow rate (or packet pace). We implement an ABQ prototype using NetFPGA, demonstrating novel insights for congestion control in both industry and academia.

Our contributions include three aspects listed as follows:

- We have developed Active Buffer Queueing (ABQ) in congestion control for DCNs, which enables switches to adjust the flow rate directly for congestion control.
- We have used NS3 (Network Simulation 3) to simulate an ABQ prototype and NetFPGA 1G to implement an ABQ prototype. The NS3 C++ codes of an ABQ prototype are available online at https://github.com/inituser/scc.
- We have evaluated the effectiveness of ABQ for relieving congestions. In our congestion evaluations compared with Droptail (with TCP) and ECN (with DCTCP), ABQ shows a 50 percent less mean size of output queue, four times fewer packet drops, and approximately 10 percent more goodput.

Background

A modern datacenter network has been an important infrastructure around the globe. Driven by novel cloud computing and data storage applications [1], a datacenter network (DCN) has

Lei Xu is with Tsinghua University and ArxanChain; Ke Xu and Xinle Du (corresponding authors) are with Tsinghua University and Beijing National Research Center for Information Science and Technology; Tong Li and Kai Zheng are with Huawei; Meng Shen is with Beijing Institute of Technology; Xiaojiang Du is with Temple University.

Digital Object Identifier: 10.1109/MNET.001.1900266

0890-8044/20/$25.00 © 2020 IEEE

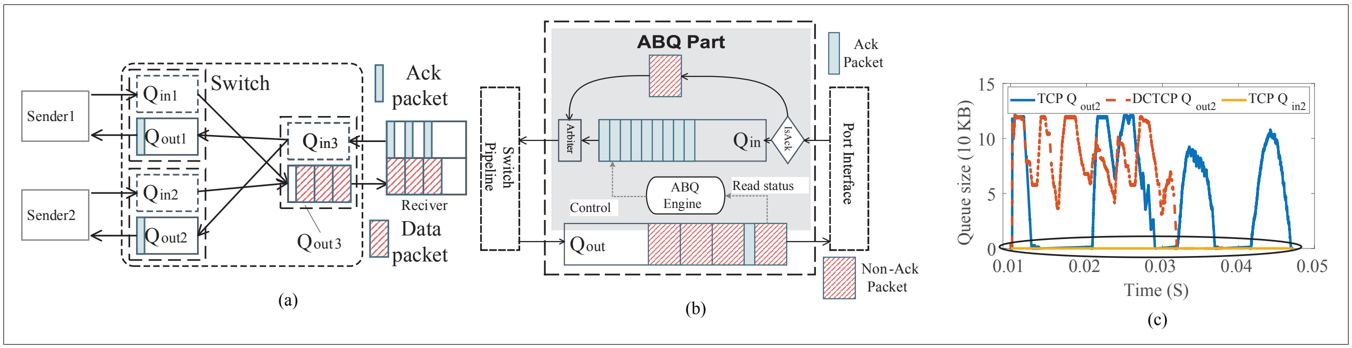

FIGURE 1. The switch model and ABQ framework: a) the switch model; b) the ABQ framework in switch; c) the oscillation of output queue and idleness of input queue.

a more explicit tree topology with higher throughput and lower delay [2], compared with the local area network (LAN).

Due to the topology and delay difference in DCNs, traditional congestion control cannot handle the congestion issues. Especially when many flows are competing in the same bottleneck link, the lower delay makes datacenter round trip time (RTT) small, and the congestion control mechanism will rapidly reduce the congestion window. Afterward, while the switch queue overflows, multiple flows will lose packets. Even worse, a certain flow may lose packets in the entire window with the small congestion window, which cannot trigger TCP fast retransmission but trigger retransmission time out (RTO), resulting in throughput decline.

Nowadays, the existing AQMs (e.g., RED [3], REM [4]) try to use packet loss or ECN label as congestion information. In a high-speed network, packet loss cannot be used, because of easy-to-lose packets and nonnegligible retransmission delay. In contrast, ECN can well mitigate the above mentioned delay, and it is widely used in DCNs as the congestion feedback. Priority-based Flow Control (PFC) [5] allows a switch to avoid buffer overflow. PFC forces the upstream entity (either another switch or a host NIC) to suspend data transmission. However, PFC operates at port level and does not distinguish between flows. This can cause congestion-spreading, which leads to poor performance. Cutting Payload (CP) drops packet payload at an overloaded switch to inform the sender about the state more quickly [6]. MQ-ECN [7] designs an effective solution to mark the ECN flag for multi-service multi-queue production DCNs.

Meanwhile, many other transport protocols were also proposed with capabilities for congestion mitigation. We classify the related literature into two categories.

First, the traditional sender-driven protocols, like DCTCP [8], DCQCN [9] and TIMELY [10]. Datacenter TCP (DCTCP) uses the ECN mechanism to detect congestion degree. It suppresses congestion by adjusting the congestion window with the ECN mark. DCQCN is an end-to-end congestion control scheme based on the Remote Direct Memory Access (RDMA) protocol RoCEv2. It uses QCN, PFC and SmartNIC to achieve high throughput and ultra-low latency with low CPU overhead. TIMELY adjusts transmission rates by using RTT gradients to keep packet latency low while delivering high throughput. All these works adjust the rate on the sender. However, it is hard for a sender to sense congestion accurately. DCTCP's one-bit ECN flag does not reflect congestion in real-time. Although DCQCN is more timely, it requires PFC and QCN to offer multi-bit information. TIMELY can offer more information by RTT, but it needs accurate measurement and is easily affected by network instability.

Second, some protocols try to adjust the rate at the receiver side, for example, ExpressPass [11], NDP [12] and Homa [13]. ExpressPass is an end-to-end credit-scheduled, delay-bounded congestion control framework. It uses credit packets to control congestion even before sending data packets, which achieves bounded delay and fast convergence. NDP and Homa use receiver-driven scheduling and priorities to schedule packets properly. They decouple the functions of adjusting rate from the sender to solve last-hop switch congestions.

Although these schemes can handle DCN congestion issues to some extent, they do not take full care of switches. Switches in a traditional wide area network (WAN) are hard to replace, because of the widely used TCP/IP. Instead, in DCN, it is feasible to change the design of the switch to enhance transmission performance [12]. Later, we will demonstrate our new AQM designed for DCNs.

Motivation

In DCNs, a host can generate more than 1,000 concurrent connections [8]. On such a workload, state of the art congestion control algorithms or mechanisms such as TCP will experience incast congestion. As shown in Fig. 1c, when congestion occurs, the output queue of the bottleneck switch, often the last hop switch, begins to build up. Meanwhile, both the senders and the receivers are not aware of the congestion accurately and promptly.

We use NS3 to conduct an 80-to-1 traffic pattern to observe a congestion problem in the output queue. The traffic pattern is run with DropTail (with TCP) and ECN (with DCTCP) respectively. Figure 1c illustrates the oscillation of output queue sizes over the period from a flow's start to its end.

From Fig. 1c, we find that the output queue sizes sharply go up to its maximum and go down quickly. The oscillation in output queue is unstable. The output queue always overflows. ECN shows better than DropTail because its mechanism informs the sender to decrease the congestion window rather than drop packets, but ECN

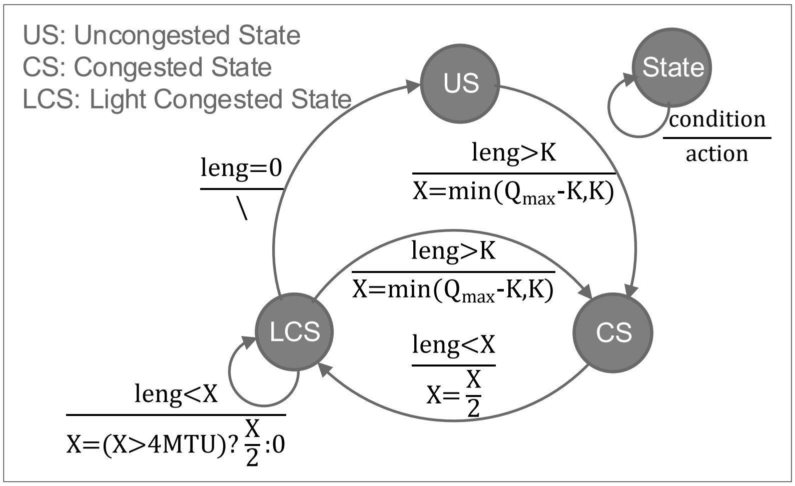

FIGURE 2. ABQ state machine.

still cannot make the queue stable. Meanwhile, Q_{in} always maintains one Ack (Acknowledgment) packet, as shown in the black ellipse in Fig. 1c, because the input queue is used for time synchronization rather than buffering. The input queues are underused.

To explain the design, we use a simple switch model as shown in Fig. 1a. The input queue is a buffer queue locating in the entrance of packets, for example, $Q_{in1,2,3}$ in Fig. 1a. As mentioned earlier, the input queues are underused. It can be changed to use arbiters to buffer ACK packets and are easy to obtain congestion state of the switches. Accordingly, ABQ can be deployed in the input queues to slow down the reverse flow.

The output queue is a buffer queue locating in the exit of packets, for example, $Q_{out1,2,3}$ in Fig. 1a. ABQ cannot be deployed in the output queues, because congestion always happens in the output queues and if the output queues are active buffering, the congestion will be more severe. ABQ needs to be deployed where there is not that much congestion, and active buffer will make the flow slow down.

Hence we use the switches' input queues to build a mechanism that can react to congestion proactively and accurately. It should keep the output queue stable rather than overflow. We propose ABQ, which proactively buffers packets according to congestion state in order to adjust the flow rate (or packets pace) to stable the rate rather than only mark the congestion label.

In a word, the switch has two factors contributing to ABQ for relieving congestion. First, the switch is where congestion occurs, and it is straightforward to access and utilize the congestion state. Second, ABQ is deployed in the switch input queue. It does not require modification on output queue mechanisms and end host protocols, which means ABQ can cooperate with other AQMs at the same time.

Active Buffer Queueing (ABQ)

In this section, we will explain how ABQ works in congestion and how to design ABQ. When there is no congestion, senders send data packets to receivers and receivers send Acknowledgment (Ack) packets to senders. According to the Acks' label or sequence number, senders adjust their sending rate, like Fig 1a. When a switch output queue is congested, the ABQ in the corresponding input queue starts buffering incoming Ack packets. Since the sender does not receive any Acks, it will stop sending data packets, which alleviates congestion to some extent.

As shown in Fig. 1b, the ABQ framework consists of several modules. *IsAck* is used to check whether a packet is an Ack packet. *Arbiter* is used to choose a packet and deliver it to switch pipeline, preventing collisions between two packets from Q_{in} and *IsAck*, respectively. The core module *ABQ-Engine* includes two parts. The first is a queuing discipline for input queue Q_{in}. The second is a congestion window corrector for setting ECE (ECN-Echo) to the Ack header. Q_{out} is the same as the output queue of normal switches, and it provides its real-time length to the ABQ-Engine.

Queuing Discipline

ABQ defines three congestion states: uncongested state (US), congested state (CS), and light congested state (LCS). In different states, Q_{in} has different dequeuing disciplines.

Figure 2 illustrates the ABQ state machine. There are two parameters used to determine the current congestion state. The first is *K*, representing the threshold for Q_{out}. The second is *X*, representing the degree of congestion. The larger the *X*, the more congested the network. Once in congestion state (CS), $X = min(Q_{max} - K, K)$. In light congested state (LCS), when the length of Q_{out} decreases to *X*, *X* is halved, that is, $X = X/2$. If *X* is less than the size of four maximum transmission units (MTUs), *X* is set as zero, that is, $X = (X > 4MTU)?X$: 0 and ABQ is in uncongested state (US).

Enqueue Discipline: As described above, ABQ input queue Q_{in} always enqueues an Ack packet if an Ack packet arrives at Q_{in}. Hence, the enqueuing discipline is the same as the enqueuing discipline of the FIFO queue.

Dequeue Discipline:

- In US, Q_{in} dequeues Ack packets as long as Q_{in} is not empty.
- In CS, Q_{in} does not dequeue Ack packets.
- In LCS, including CS changing to LCS and LCS changing to US, Q_{in} dequeues Ack packets once only when the length of Q_{out} is *X*, then *X* is halved which is explained in Fig. 2.

In LCS, Q_{in} can choose different numbers of Ack packets to dequeue. There is a trade-off between congestion and link utilization. In this article, we suggest that once dequeuing, Q_{in} dequeues *all* the Ack packets in it.

There are two reasons for dequeuing all Ack packets. First, dequeuing fewer Ack packets would lead to lower link utilization, especially for light congested situations, where the delayed Ack would increase flow completion time. Second, to further relieve the congestion in Q_{out}, ABQ adopts a congestion window corrector to suppress the increment of the congestion window for each connection.

Congestion Window Corrector

The queuing discipline in the above subsection relieves Q_{out} congestion initially. However, if the sender receives Ack packets, it still increases its congestion window dramatically according to

TCP additive increase and multiplicative decrease algorithm, leading to congestion afterward. Hence ABQ provides a congestion window corrector to suppress the increment of the congestion window when Q_{out} is congested.

The corrector uses ECN-related mechanisms. When ABQ is in LCS, the corrector would mark congestion flag ECE to each Ack packet every time Q_{in} dequeues them. Once senders receive the Ack packets marked with ECE, they will suppress window increment according to the ECN mechanism.

It changes only one bit in the header and causes one bit cost in the TCP header checksum. This is because the checksum is the one's complement of the sum of 16 bit chunks. If one bit of ECE flag is changed, the checksum can be directly calculated by subtracting the fix value from old checksum.

Implementation and Evaluation

We implemented ABQ on NetFPGA 1G, as shown in Fig. 3. On NetFPGA, ABQ is added as *Input Queue* pipeline in front of *User Data Path* pipelining. We plot the data path and the control path which are important to the ABQ framework.

The *Input Queues* module has four data paths corresponding to four switch interfaces. Meanwhile, there are feedbacks (*Length*) from the *Output Queues* module to indicate how long the output queue is. The following discusses the details of each logic module in *Input Queues*.

The first logic module is Is*Ack* as shown in Fig. 3. This module is used to check whether a packet is an Ack packet, as described above. The second and last logic module is *Input Arbiter*. This module is to solve collision among four concurrent incoming packets and choose one packet to send to the following path.

The enqueuing discipline of ABQ is deployed in *Store Pkt*. The *Remove Pkt* module is used to get a packet from SRAM and send it out. If the current input queue has Ack packets in SRAM, it would send *Rd Req* to SRAM. Then SRAM returns a packet to the corresponding FIFO.

We implement dequeuing discipline in the Dequeue module shown in Algorithm 1, The algorithm is implemented in an *always* statement of Verilog Hardware Description Languages (HDL).

We use packet-level simulator NS3 to evaluate the performance of ABQ. In this section, ABQ works with senders deployed with DCTCP. In NS3, the link delay is 40 μs and the bandwidth is 1 Gb/s. The size of the switch output queue is 128 KB (i.e., 80 1500-B packets). The Retransmission TimeOut (i.e., RTO_s^{min} for TCP sender) is set to 10 ms, which is similar to the configuration in [14]. The ECN marking threshold K_{ECN} is set to 40 packets for DCTCP. We set the parameters K of ABQ to 20 packets. The K_{ECN} and K have different limitations because they have different models for steady-state behaviors.

Micro-Benchmark

We focus on the incast problem, in which many concurrent senders start to transmit 32-KB flows to one receiver at the 0.01th second. All the hosts are under the same switch whose configurations are described above. We increase the number of concurrent senders from 10 to 100 to observe the performance.

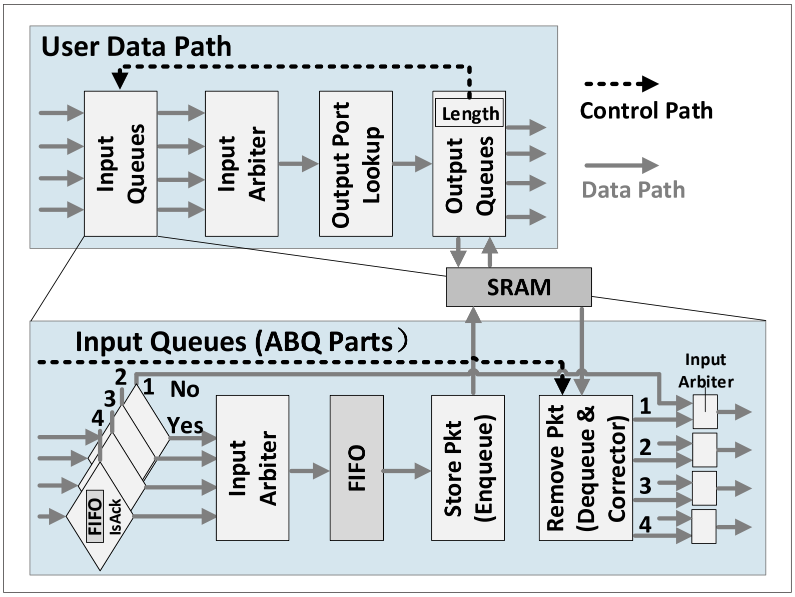

FIGURE 3. The NetFPGA implementation of ABQ.

```
Require: Num_Words_in_Q
If $Num\_Words\_in\_Q > K$ then
    $State = CS$; Check next queue;
else If $Num\_Words\_in\_Q < X$ and ($State == CS|LCS$) then
    $State = LCS$; $X = X/2$;
    If $X < 4$ then $State = US$;
    end If
    Dequeue all Ack packets;
end If
```

ALGORITHM 1. Dequeue Logic.

Queue Buildup and Input Queue Cost: We measure the size of Q_{out} and ABQ's Q_{in} in the last-hop switch. In Fig. 4a, the red bar is the length of the output queue and the purple bar is the total length of the input queue and the output queue. From Fig. 4a, the mean sizes of DropTail (with TCP) and ECN (with DCTCP) are 1.5 and 1.6 times of that of ABQ, and the total length of the input queue and the output queue is close to the length of the output queue. ABQ can make queues significantly shorter, with very little buffer overhead. These observations reveal that ABQ is effective to suppress switch congestion.

Goodput and Packet Drops: In Fig. 4b, ABQ maintains better goodput than DropTail and ECN, while DropTail and ECN have goodput decreases, revealing that they are impacted by incast. ABQ has effective mechanisms to suppress congestion, leading to better goodput. Sometimes ABQ suppressed congestion radically so that the goodput is also suppressed, for example, in 10, 20 and 70 senders scenarios. Note that ECN has lower goodput than DropTail at 20 senders. This is because ECN defers congestion to subsequent rounds instead of eliminating them, resulting in more packet-drops.

In Fig. 4c, ABQ drops a few packets in all the incast scenarios. ECN drops fewer packets than TCP due to the ECN mechanism. ABQ drops the fewest packets in all the incast scenarios. The reason is that ABQ has the smallest output queue size among all the scenarios.

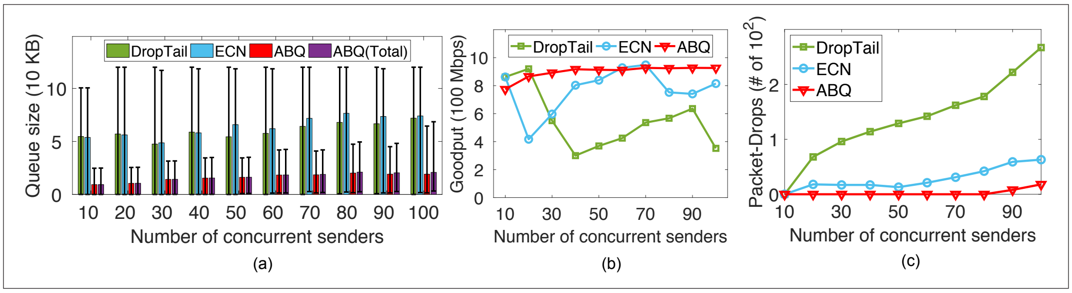

FIGURE 4. Queue size and Goodput: a) the mean with the 5th and 95th percentile of queue size; b) goodput; c) packet-drop times.

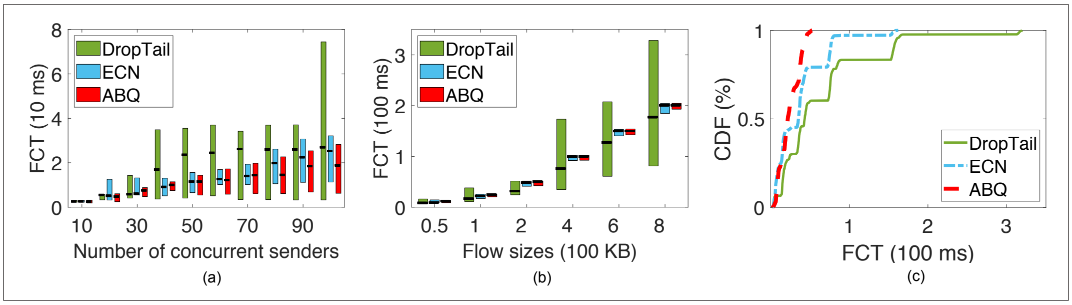

FIGURE 5. FCTs: a) The 1st, 50th, 99th percentile FCT with varying number of senders and 32 KB flows; b) the 1st, 50th, 99th percentile FCT with varying flow sizes and 30 senders; c) the CDF of short flow FCTs in at-scale DCN.

Flow Completion Time: To observe the impact of ABQ on FCT in congestion environments, this subsection conducts two types of incasts. One is with a varying number of senders and fixed flow size (32 KB in Fig. 5a), the other is with varying flow sizes and a fixed number of senders (30 senders in Fig. 5b). We plot the 1st, 50th and 99th percentile FCTs of three congestion mechanisms in Fig. 5.

As shown in Fig. 5a, with the growing number of senders, all the 50th-percentile FCTs of three mechanisms increase. ABQ's 99th percentile FCTs hold low. Besides, ABQ's 1st and 50th percentile FCTs are not large. ABQ has 90 percent and 10 percent decrease on average in the 99th-percentile FCT over DropTail and ECN, respectively. DropTail has 99th-percentile FCTs due to its severe packet drops.

In Fig. 5b, both ABQ and ECN achieve 99th-percentile FCTs. From this observation, ABQ and ECN are both effective in transporting long flows, because both ABQ and ECN use ECN mechanism to relieve congestion, leading to similar effects on long flows. Although DropTail achieves 1st-percentile FCTs, its 99th-percentile FCTs are still largely due to drastic packet-drops, which is attributed to its proactive congestion control.

Benchmark at Scale

To measure ABQ's performance on typical DCNs, we use NS3 to conduct an at-scale datacenter network. We adopt fat-tree topology in which a core switch node is used to connect all the Top of Rack (ToR) switches. The servers under ToR connect to ToR via 1-Gb/s links, and the ToRs connect to the core switch via 1×number-of-servers-under-a-ToR Gb/s. The number of servers under a ToR is 25 and the number of ToR is 40. The switch configurations stay unchanged.

We randomly choose five receivers out of 1000 servers to receive data. Short flow sizes comply with the Pareto distribution with mean 50 KB and shape 1.2. The arrival times of flows comply with exponential distributions with means of 300 μs, 400 μs, 500 μs, 600 μs and 700 μs. Meanwhile, there are five random long-lived flows as background traffic, each of which is 30 MB, occupying 75 percent of the total traffic in the DCN.

The short flow FCTs are plotted in Fig. 5c, where ABQ achieves small FCTs. DropTail has a long tail of FCTs while ECN has comparable FCTs. The figure shows that the ECN mechanism in ECN and ABQ are effective to cope with short flows.

We record the information of long flow throughput and packet drops in ToR switches in Table 1. In Table 1, ABQ achieves the high throughput of long flows while ECN has a moderate performance for long flows. DropTail has a bias to long flows which leads to lower throughput.

In Table 1, ABQ drops fewer packets than the other two mechanisms, which is the key factor for its performance.

Limitations

ABQ is the first trial to enable switch input queue to solve the congestion problem. Although our results are encouraging, there are several limitations to our work.

First, ABQ needs to change the design of the switch. We implement ABQ in NetFPGA to prove that it can be deployed on the hardware and the cost is low. This article intends to open the view of congestion control in DCNs and provide initial ideas to improve switch hardware.

Second, ABQ is based on symmetric traffic. When there is non-symmetric traffic in DCNs, ABQ may buffer the "wrong" Ack packets which may not correspond to the data packets that cause the congestion in the current switch. To solve this problem, we can use the flow level ECMP to make the flow stay symmetric.

Conclusion

Active Buffer Queueing (ABQ) is a new Active Queue Management (AQM) scheme for datacenter transport to solve the congestion problem. It can relieve congestion by enabling a switch to adjust the transmission rate. ABQ deploys an input queue in a switch to buffer Ack packets when the corresponding output queue is congested. This article initially measures the performance of ABQ and implements an ABQ prototype on NetFPGA. Our experiments show that, compared with DropTail and ECN, ABQ can achieve the highest Goodput with small queue size. ABQ can cooperate with other AQMs whose feedback mechanism is based on the switch output queue at the same time.

Acknowledgment

This work is supported by the National Science Foundation of China (61825204, 61932016, 61972039, 61602039); the National Key R&D Program of China (2018YFB0803405); the Beijing Outstanding Young Scientist Program (BJJWZY-JH01201910003011); and the Beijing Municipal Natural Science Foundation (4192050).

References

[1] L. Lv *et al.*, "Communication-Aware Container Placement and Reassignment in Large-Scale Internet Data Centers," *IEEE JSAC*, vol. 37, no. 3, 2019, pp. 540–55.

[2] Y. Zhang *et al.*, "Going Fast and Fair: Latency Optimization for Cloud-Based Service Chains," *IEEE Network*, vol. 32, no. 2, 2017, pp. 138–43.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Trans. Networking*, vol. 1, pp. 397–413.

[4] S. Athuraliya *et al.*, "REM: Active Queue Management," vol. 15, pp. 48–53.

[5] IEEE. 802.11qbb, Priority Based Flow Control, 2011.

[6] P. Cheng *et al.*, "Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers," *Proc. NSDI*, 2014.

[7] W. Bai *et al.*, "Enabling ECN in Multiservice Multi-Queue Data Centers," *Proc. 13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 16, 2016.

[8] M. Alizadeh *et al.*, "Data Center TCP (DCTCP)," *Proc. SIGCOMM 2010*.

[9] Y. Zhu *et al.*, "Congestion Control for Large-Scale RDMA Deployments," *Proc. SIGCOMM*, 2015.

[10] R. Mittal *et al.*, "RTT-Based Congestion Control for the Datacenter, *Proc. SIGCOMM*, 2015.

[11] I. Cho *et al.*, "Credit-Scheduled Delay-Bounded Congestion Control for Datacenters," *Proc. SIGCOMM*, 2017.

[12] M. Handley *et al.*, "Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance," *Proc. SIGCOMM*, 2017.

[13] B. Montazeri *et al.*, "A Receiver-Driven Low-Latency Transport Protocol using Network Priorities," *Proc. SIGCOMM*, 2018.

[14] V. Vasudevan *et al.*, "Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication," *Proc. SIGCOMM*, 2009.

Biographies

Lei Xu received his bachelor degree in computer science from Beijing Institute of Technology, China in 2006. He received his Ph.D. from the Department of Computer Science & Technology, Tsinghua University, Beijing, China in 2018. His research interests include datacenter networking, router security and block chain security. He is now working at ArxanChain CO.LTD for block chain technologies.

Value	DropTail		ECN		ABQ	
	Long Thrp. (Mb/s)	ToR Drop (#)	Long Thrp. (Mb/s)	ToR Drop (#)	Long Thrp. (Mb/s)	ToR Drop (#)
Smallest	653	673	743	420	818	258
Small	788	673	821	438	819	260
Median	789	750	829	449	823	270
Large	792	763	833	487	836	281
Largest	795	841	928	581	837	309

TABLE 1. The throughput and packet drops in ToR switches in at-scale networks.

Ke Xu received his Ph.D. from the Department of Computer Science & Technology, Tsinghua University, Beijing, China, where he serves as a full professor. He has published more than 100 technical papers and holds 20 patents in the research areas of next generation Internet, P2P systems, Internet of Things (IoT), and network virtualization and optimization. He is a member of ACM and has guest-edited several special issues in IEEE and Springer journals. He is a senior member of IEEE.

Tong Li received his B.S. degree from the Department of Computer Science, Wuhan University, China in 2012, and his Ph.D. degree from the Department of Computer Science & Technology, Tsinghua University, Beijing, China in 2017. His research interests include network protocols, measurement, and cloud/edge computing.

Kai Zheng received the B.S. degree from Beijing University of Posts and Telecommunications, China, in 2001, and Ph.D. degrees from Tsinghua University, China, in 2006. He joined Huawei Technologies in 2015 and is now Director & Chief Architect of Distributed Communication. Before that, he was a senior research staff member at IBM. His current research interests include datacenter networking, software defined (transport layer) protocols, WAN optimizations, IoT protocols, and so on. He is a senior member of IEEE.

Meng Shen received the B.Eng degree from Shandong University, Jinan, China in 2009, and the Ph.D. degree from Tsinghua University, Beijing, China in 2014, both in computer science. Currently he serves at the Beijing Institute of Technology, Beijing, China, as an associate professor. His research interests include privacy protection for cloud and IoT, blockchain applications, and encrypted traffic classification. He received the Best Paper Runner-Up Award at IEEE IPCCC 2014. He is a member of the IEEE.

Xiaojiang Du is a tenured professor in the Department of Computer and Information Sciences at Temple University, Philadelphia, USA. He received his B.S. and M.S. degrees in electrical engineering from Tsinghua University, Beijing, China in 1996 and 1998, respectively. He received his M.S. and Ph.D. degrees in electrical engineering from the University of Maryland College Park in 2002 and 2003, respectively. His research interests are wireless communications, wireless networks, security, and systems. He has authored over 300 journal and conference papers in these areas, as well as a book published by Springer. He has been awarded more than US$5 million in research grants from the U.S. National Science Foundation (NSF), Army Research Office, Air Force, NASA, the State of Pennsylvania, and Amazon. He won the best paper award at IEEE GLOBECOM 2014 and the best poster runner-up award at the ACM MobiHoc 2014. He serves on the editorial boards of three international journals. He is a Senior Member of IEEE and a Life Member of ACM.

Xinle Du received his bachelor degree in computer science from Xidian University, Xi'an, China in 2018. He is working toward his Ph.D. degree supervised by Prof. Ke Xu in the Department of Computer Science & Technology, Tsinghua University, Beijing, China. His research interests include datacenter networking.