Reconciling Mice and Elephants in Data Center Networks

Ahmed M. Abdelmoniem and Brahim Bensaou
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{amas, brahim}@cse.ust.hk

***Abstract*—Small switch buffers, high-speed links, short round-trip times and the composite nature of the traffic in data center networks (DCN) lead to several congestion problems that are not handled well by traditional congestion control mechanisms such as TCP. In this paper we design a simple switch-driven, flow-aware, congestion control algorithm to deal with such congestion issues. The basic idea of the proposed mechanism is reminiscent of classic congestion control in flow-aware networks such as ATM-ABR, where the switch sets a field in packet headers to enforce the sending rate at the source; less classic though are the challenges faced in designing such flow-awareness in the flow-aversive IP environment *without modifying the TCP sender and receiver algorithm*, to enable deployment in public data centers. We discuss in this paper our algorithmand give numerical results from NS-2 simulations to show its effectiveness in achieving high throughput overall, a good fairness and short flow completion times for delay-sensitive flows[1].**

***Keywords*—Congestion Control, Data Center Networks, Flow Control, Kernel Module, OpenvSwitch**

I. Introduction and Background

Cloud computing is usually host to a plethora of applications with non-homogeneous network traffic characteristics and performance requirements. These applications range from transactional, time-sensitive applications such as web searches, to bulk transfers time-insensitive, throughput-addicted applications such as backups. From the network traffic perspective, transactional applications are often executed by several workers, that generate several small traffic flows (called mice in the sequel), whose merger and consolidation produces the application data. This is typical of today's web searches, social networking, and MapReduce jobs. Such applications require not only a timely delivery of their traffic, to meet the end user delay requirement, but also a small jitter between the traffic flows, to improve the quality of the results: for instance web search results that arrive too late are discarded. On the other hand bulk transfers (called in the sequel elephants) require no delay guarantee but are only effective if provided with a high and sustained throughput. These are typical of backups, data migration, and data center (DC) synchronization.

The co-existence of such synchronous mice and bulky elephant flows with various performance requirements, in the presence of the large bandwidth and small round-trip delays experienced in DCNs (tens to hundreds of microseconds), pose great challenges to traditional congestion control mechanisms such as TCP. In particular, the switch buffers and the round trip delays being small in typical DCNs [1, 2], several congestion symptoms that cannot be simply inferred from packet losses, appear, and require specific treatment to avoid congestion collapse. Most prominently: *i)* Incast traffic congestion, where many correlated mice flows converge onto the same congested output port of a switch over a short period of time, typically as a response data for partition/aggregate type of applications. These are very commonly found in data centers; and, *ii)* Queue-buildup/Buffer-Bloating that occurs as a normal behaviour of TCP when the buffer space of the port (in deep buffered switches) or the buffer space of the switch (in shallow buffered switches) is occupied by elephant flows, leading mice flows to experiencing repeated packet drops and unnecessary increase in queuing delays.

Due to the impact and severity of these congestion symptoms, much recent work has been devoted to addressing such shortcomings of TCP in DCNs. Most of the proposed solutions fall in two categories: window based schemes (e.g., [2, 3]) or fast loss recovery schemes (e.g., [4, 5]).

In the window-based category, DCTCP [2] proposes a modification to TCP and RED active queue management that adjusts TCP's congestion window to stabilize the queue length in the switch at a predefined small threshold, guaranteeing thus short delays for incast traffic, without degrading the link utilization. ICTCP [3] also was proposed as a modification to TCP receiver to handle incast traffic. ICTCP adjusts the TCP receiver window proactively, to avoid congestion at the receiver. The experiments with ICTCP in a real testbed show that ICTCP can almost curb timeout-detected losses and achieves a high throughput for TCP incast traffic, however, since it is focused on incast it only handles congestion at the receiver and does not address buffer buildup in the switches.

Fast loss recovery schemes try to improve the agility of TCP in recovering from congestion events by shortening the reaction time. For instance, [4] reduces TCP's minimum retransmission timeout RTO_{min} to reduce the unnecessarily long waiting times after packet losses to enable a fast reaction to congestion losses in the presence of shallow buffers (where losses are mostly detected by timeout). In contrast [5] cleverly tries to deploy a fast congestion-detection mechanism by truncating the packet payload of congestion-causing packets, only conveying the header to the receiver. This enables a receiver-driven explicit congestion-notification upon reception of truncated packets.

[1]This work is supported in part under Grants: HKPFS PF12-16707, REC14EG03 and FSGRF13EG14.

Fast loss recovery schemes potentially solve the problems of congestion in data centers, however, they require not only switch modification, but also end-system modifications. For example, in Linux RTO_{min} is equal to 200ms and is hard-coded in the TCP source code.

While both categories achieve better performance than traditional TCP, they still suffer an important drawback inherent to their methodology: typically, in public data centers, especially those providing infrastructure as a service (IaaS), many different guest operating systems can co-exist each allowing many different versions of TCP to be used. The guest OS can also be uploaded by the tenant and the TCP configuration parameters can also be tuned by the end user. In addition, the client in many public data center applications in platform or software as a service (PaaS and SaaS) can often be outside the data center (e.g., cloud-based intrusion detection systems, web servers, and so on). This renders modification to the TCP sender, receiver or timer only feasible in privately owned data centers including the guest OS.

Despite these drawbacks, the improved performance shown by all these schemes is compelling evidence to investigate further the problems of congestion in DCNs with the aim of designing an effective solution to the problem that *does not require modification to the TCP sender, nor to the receiver.* In contrast, since the switches, routers and servers in any DCN belong to the same administrative entity, we explore a solution that only relies on simple modifications to the switch software and/or server hypervisor/host OS.

With this objective in mind, in this paper, we take a flow-aware approach similar to traditional flow-based systems like ATM-ABR or XCP [6]. The challenge that arises however is how to deploy such flow-awareness in the flow-aversive IP environment without modifying the TCP sender and receiver. This disqualifies XCP, as it is a clean-slate redesign that requires not only changes to the routers but also to the sender and receiver. To achieve our goal, the switch/router must be able to track flows, calculate a fair share for each flow that traverses it, and must have means to convey back this fair share to the source. In our approach, we invoke SDN ability to track flows to enable our network with flow awareness, and modify the switch software to rewrite TCP receiver window to communicate with the sender. TCP flow control being a fundamental part of any TCP incarnation, including XCP, our proposed mechanism would fit-in without any change to the end-hosts. Our contributions in this paper are two-fold:

- We first propose a simple switch-based congestion control mechanism called *Receiver window queue (RWNDQ)* that achieves a high efficiency by maintaining the queue occupancy within a predetermined target level; achieves a good fairness in both short and long term; and ensures mice flows complete their flow in a short time.
- We analyse the stability of our mechanism using a simple mathematical model and examine its effectiveness via extensive simulations in ns2, comparing it to TCP, DCTCP and XCP.

The remainder of the paper is organized as follows, we first discuss our proposed methodology and present the proposed RWNDQ switch queue management algorithm in Section II. We further develop a simple analytical model of RWNDQ to study its convergence in III, then evaluate its performance via simulation and compare it to alternative approaches in IV. finally conclude the paper in V.

II. RWNDQ Algorithm

A. Proposed methodology

TCP is a full-duplex protocol where the two receiving end-points allocate a receiving buffer space to enable flow control. To this end, the receiver of one direction sends back to the sender acknowledgement packets (ACKs) that include, in the 16 bit "Receive Window" field (*Rwnd*), the currently available buffer space.

In our approach we propose to overwrite *Rwnd* and the scaling option value n to indicate the bottleneck fair share of bandwidth available for a given flow on a given path between the source and destination. As the ACK from a receiver traverses the switches in the reverse path towards the sender, each switch examines the packet and modifies the *Rwnd* value taking into account the window scaling value if necessary. More specifically, at each switch along the end to end path:

1) Before an ACK is forwarded to the output port on the reverse path, the available buffer space in the port that holds the corresponding data in the forward path is sampled.
2) Based on the number of ongoing flows and the target queue occupancy, the *Rwnd* field in the ACK header is rewritten with the flow's fair share, if this latter is smaller than the current value of the receiver window.
3) As the ACK travels across the switches/routers along the reverse path, *Rwnd* is updated with the bottleneck fair share for the flow.
4) Since a TCP sender's rate is limited by the minimum of the current congestion window and the receiver window, in the absence of losses, after the slow start phase, it is expected that the sender window will be limited by the receiver window field which reflects the bottleneck link fair share.

B. RWNDQ Algorithm

The main variables and parameters used in RWNDQ algorithm are described in Table I. Note that T, M and α are parameters of the algorithm that can be chosen by the DCN administrator.

Table I: *Variables and Parameters used in Algorithm 1*

Paramter name	Description
T	Timeout value for window increment interval
M	Number of increment intervals to wait for an update
α	Target level of queue occupancy
Variable name	**Description**
Q_{Rwnd}	Common receive window value for all flows
Q_{fcount}	Number of current ongoing flows
Q_{len}	Current length in bytes of the output queue
Q_{limit}	buffer size on the reverse path
Q_{ratio}	Ratio of divergence from the target queue size
W_{incr}	Window increments of one update interval
P	a packet
$ReceiveWnd(P)$	*Rwnd* and the scaling factor of packet P

RWNDQ shown in Algorithm 1 is event-driven and runs on each port of the switch to respond to two major events: packet arrivals, and window increment timer events to trigger window updates.

Upon a packet arrival: the algorithm updates the maximum packet size seen so far. If this is the first flow, then the current window is initially set to the target-queue worth of bytes then RWNDQ enters the slow-start phase to start probing for the current window size, because initially the bandwidth-delay

Algorithm 1 RWNDQ Algorithm

procedure PACKET_DEPARTURE(P)
  if $Max\_Size < Size(P)$ **then**
    $Max\_Size \leftarrow Size(P)$
  if $SYN - ACK\_bit\_set(P)$ **then**
    if $Q_{fcount} = 0$ **then**
      $Q_{Rwnd} \leftarrow \alpha \times Q_{limit}$
    else
      $Q_{Rwnd} \leftarrow Q_{Rwnd} \times \frac{Q_{fcount}}{Q_{fcount}+1}$
    $Q_{fcount} \leftarrow Q_{fcount} + 1$
  else if $FIN\_bit\_set(P)$ **then**
    $Q_{fcount} \leftarrow Q_{fcount} - 1$
    if $Q_{fcount} > 0$ **then**
      $Q_{Rwnd} \leftarrow Q_{Rwnd} \times \frac{Q_{fcount}+1}{Q_{fcount}}$
    else
      $Q_{Rwnd} \leftarrow \alpha \times Q_{limit}$
  else if $ACK\_bit\_set(P)$ **then**
    if $Q_{Rwnd} < ReceiveWnd(P)$ **then**
      $ReceiveWnd(P) \leftarrow Q_{Rwnd}$
procedure WINDOW_INCREMENT_TIMEOUT
  $Q_{ratio} \leftarrow 1 - \frac{Q_{len}}{(Q_{limit} \times \alpha)}$
  $W_{incr} \leftarrow W_{incr} + \frac{Q_{ratio} \times Max\_Size}{M}$
  $interval\_count \leftarrow interval\_count + 1$
  if $interval\_count == M$ **then**
    if $slowstart$ **then**
      $Q_{Rwnd} \leftarrow Q_{Rwnd} + 2 \times Max\_Size$
      if $Q_{len} \geq \alpha * Q_{len}$ **then**
        $slowstart \leftarrow FALSE$
    else
      $Q_{Rwnd} \leftarrow Q_{Rwnd} + W_{incr}/Q_{fcount}$
    $interval\_count \leftarrow 0$
    $W_{incr} \leftarrow 0$

product is unknown to the switch. Subsequently, for each new flow, the current window is divided equally among all flows. If the ACK bit is set, the receive window field *Rwnd* of this Packet is updated with the current local window Q_{Rwnd} .

Upon Window increment timer elapse: Q_{ratio} is calculated to track the deviation of the current queue length from the target. The ratio controls the fraction of segments (MSS) added or subtracted from the current value of Q_{incr}. After M such updates, the current value of the common receive window is updated. In slow start, RWNDQ adds two MSS to the window, otherwise it adds the current Q_{incr} value to the window. Notice that the value of the window increment is updated M times before it is reflected in the actual value of *Rwnd* that is conveyed to the TCP sender. This enables a highly accurate estimate of the increment, while keeping the number of *Rwnd* rewrites in the packet header reasonable.

RWNDQ maintains a very low loss probability and enables the switch buffers to absorb sudden traffic bursts while achieving a high utilization. Therefore it is appealing for handling the co-existence of mice and elephants. RWQND adopts a proportional increase, proportional decrease approach. As soon as the queue length exceeds the target queue length, the window is shrunk in proportion to the excess and vice-versa. Furthermore the increase/decrease amount is equally divided among all ongoing flows. Initially and whenever the number of ongoing flows drops to 0, the algorithm goes into the slow start mode, where the window is increased by 2 MSS in each update period. When the queue exceeds the target, the algorithm goes into congestion avoidance, where the window increment is proportional to the gap between the current queue length and the target queue length.

C. Discussion, Drawbacks, Practical Solutions and Complexity

In principle RWNDQ is very effective in solving the problem of congestion, and actually avoiding it outright when there is no sudden traffic surge. However, to enable its practical deployment, *i)* the ACK packets must travel back along the reverse path taken by the corresponding data packets, *ii)* the switch must be able to track the number of ongoing flows; and, *iii)* the switch must be aware of each flow's window scaling factor to avoid semantic mismatches between the source and the switch on the value in *Rwnd* .

Two approaches are possible to achieve the first two requirements: either implement flow-awareness in the open source network OS of bare-metal switches, or, since SDN based switches are more common nowadays, one can rely on the functions already provided by SDN. We implemented and tested both approaches.

In the SDN approach, SDN capability to track flows and flow statistics can be easily invoked to address the first two requirements above. In contrast, if SDN is not available, additional knowledge of the DCN architecture and routing can enable the DCN operator to easily deploy RWNDQ. For example if single path routing is used, the learning ability of the switches can be invoked to implicitly assume that forward and reverse paths are already the same. If equal-cost multi-path routing (ECMP) is used a simple modification to the algorithm to equally divide the flow fair share among the multiple routing paths is easily deployed. In addition to track the number of active TCP flows, we can simply implement efficient packet filters to track SYN/FIN for connection establishment/tear-down without per-flow state by using counting bloom filters.

The TCP window scaling option remains an important issue. In practice this option is supposed to be activated to deal with long-fat pipes to increase the receiver window from 64KB per flow to 1GB per flow at most. However in current DCN networks, with 1Gbps interfaces and ToR switches, the scaling is not necessary as the round trip time is typically small (10 to 100 μs), enabling only a few packets per flow per RTT in normal operation to fill the pipe. According to [7], the window scaling option is supposed to be negotiated between the sender and the receiver, and to enable it, both sender and receiver must send the window scaling option in the SYN and corresponding SYN/ACK. In practice however, the scaling value is not negotiated as different implementations of TCP adopt different default values. For example by default in MacOS the scaling exponent is 3 while Linux calculates it according to the allocated receiver buffer size. Furthermore, these values can be reconfigured by the end-user to be from 0 (for no-scaling) to 14. To avoid any cognitive mismatch between the values set by RWNDQ in the *Rwnd* field and their scaled alternatives at the sender, if scaling option is negotiated then we propose to simply unify the value supported in the DCN by rewriting it in the SYN and corresponding SYN/ACK at the onset of the TCP connection via an SDN rule or directly in the switch. However, if a TCP receiver just informs its peer of its scaling value, we

propose to have a very light shim layer at the end-hosts that tracks per-flow scaling value, recomputes the receive window of outgoing ACKs and reset its value using a network-wide scaling value used by all RWNDQ-enabled switches.

In terms of processing complexity, RWNDQ is a very simple algorithm with very low complexity and can be integrated easily in switches or routers. For example it can be implemented in Linux based routers as a module using the NetFilter framework as a hook, to enable modifications to the packet headers prior to their forwarding by IP. This requires $O(1)$ per packet. RWNDQ can also cope with Internet checksum recalculation easily and efficiently after header modification, by applying a straightforward one's-complement add and subtract operations on three 16-bit words [8]: $CSum_{new} = CSum_{old} + Rwnd_{new} - Rwnd_{old}$. In addition, since RWNDQ is designed to deal with TCP traffic only, tracking the number of flows can be achieved in a scalable manner by monitoring SYN/SYN-ACK and FIN/FIN-ACK bits. This also requires $O(1)$. Disabling or unifying the window scaling factor also requires $O(1)$.

All in all, all the operations required by our algorithm are $O(1)$ and most importantly involve only the switches/routers under the control of the DC operator. In particular, no modification to the TCP source or receiver algorithms is needed to adapt the sending rate to the bottleneck link capacity.

III. Convergence of RWNDQ mechanism

Since RWNDQ adopts proportional increase, proportional decrease it is important to verify its convergence and stability. We can simply model RWQND behaviour by considering the three parts that make up the system: window increment/update at the switch, source window adjustments in response to switch feedback (assuming TCP congestion control is disabled), and queue behaviour. Similar to [9] we adopt a fluid approach to model how RWNDQ reacts proportionally to the extent of congestion and updates *Rwnd* at predetermined constant intervals in the switch before conveying this information in the ACKs to the sources. This leads us to a model that is centred around the switch where all calculations are based on the time advances. Recall that T is the increment interval duration, let MSS be the maximum segment size and denote by $\alpha \times B$ the target queue with B being the buffer size. At the beginning of operation, $W(t)$, the window size in the switch is initially set to $\alpha \times B$ bytes. By modeling the window dynamics of RWNDQ-enabled switch into a discrete time model with respect to T, $W(t)$ can be written as:

$$W(t) = \begin{cases} W(t-T) + \beta(t) & \text{if } t = kMT, \\ W(t-T) & \text{otherwise,} \end{cases} \quad (1)$$

where, k is a positive integer and $\beta(t)$ is the average value of $Q_{ratio}(t)$ over the different increment intervals expired during one update interval. Simply put, $\beta(t)$ is the number of MSS by which the window should be increased/decreased in the update interval preceding t. That is:

$$\beta(t) = \begin{cases} 0 & \text{if } t = kMT^{+}, \\ \frac{MSS}{M}\left(1 - \sum_{j=1}^{M} \frac{Q(t-jT)}{\alpha B}\right) & \text{otherwise.} \end{cases} \quad (2)$$

Notice that instead of reducing the queue dynamics in the update interval to the final value only, our calculation of $\beta(t)$ takes into account past queue fluctuations from the start of the interval by averaging all M values. The queue dynamics can be described as follows :

$$Q(t) = \left[Q(t-T) + \frac{T}{RTT} W\left(t - \frac{RTT}{2}\right) - CT\right]^{+}; \quad (3)$$

that is, the queue at time t receives a window-full of bytes that was calculated half an RTT earlier (to account for the propagation of the ACK from the router to the source and the propagation of the data from the source to the router, arriving at time t). For simplicity, we assume there is no congestion and the RTT fluctuates very little, hence the ACKs do not queue up in the reverse direction.

It is expected that the persistent queue converges to Q_{target} as t goes to infinity. To support this claim, we use the above model and run numerical experiments in Matlab. We assume traffic sources are connected to 1 switch with 83 packets of buffer size and a target occupancy of 20% = 16.6 pkts. The capacity of the output link is 10Gb/s and the RTT is 100 μs. Figure 1 shows the mean queue size over time for two scenarios (with and without slow start) as obtained from Matlab simulation. The figure supports our intuition that, at the beginning the mean queue occupancy is zero until the pipe is filled. Then it increases steadily until it converges to the target queue as time goes to infinity. In addition, slow start seems to improve the speed of convergence dramatically. That is, slow start leads the queue occupancy to the target very fast, then proportional increase proportional decrease maintains the window around the target queue occupancy while reacting with agility to congestion.

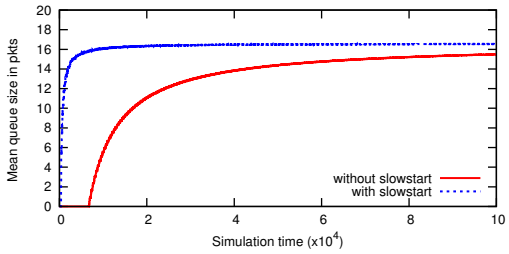


Figure 1: *Algorithm stability and convergence speed*

IV. Simulation Analysis and comparison with other alternatives

In this section, we study the performance of our algorithm via ns2 simulation in network scenarios with a low bandwidth-delay product (as is the case in data centers). We compare our system to DCTCP and XCP and demonstrate how it outperforms them. We have also compared our mechanism to TCP-RED which we omit here as its performance was far off the 3 approaches.

For RWNDQ, the values of α, T and M are chosen based only on the target level of congestion that can be tolerated regardless of capacity, delay, and number of sources. In the

simulation experiments, we set α to 20% of the buffer size, T to 50 μs and M to 10 intervals leading to an update interval every 500 μs. DCTCP and XCP parameters are set according to their recommended settings with K (the target queue occupancy) of DCTCP set to 17% of the buffer size.

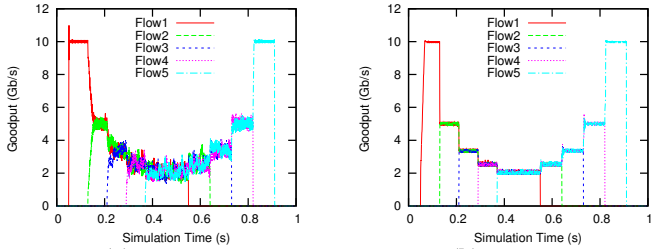
A. Simulation Setup

We use ns2 version 2.35 [10], which we have extended with an RWNDQ module. In addition, we modified ns2 TCP module, since the receiver window interaction between TCP sender and receiver (Flow Control) in ns2 does not follow the standard TCP flow control implementation. We compare TCP NewReno with SACK-enabled over RWNDQ management to DCTCP (which includes a modification of TCP and AQM) and XCP (which is a complete clean-slate approach). For DCTCP, we use a patch for ns2.35 available from the authors [11] and for proper operation, ECN-bit capability is enabled in the switch and TCP sender/receiver. For XCP, we use the version available in the ns2.35 [10] distribution. We use in our simulation experiments high speed links of 11 Gb/s for sending stations, a bottleneck link of 10 Gb/s, low RTT of 100 μs and RTO_{min} of 2 ms as opposed to the default 200 ms.

We use a dumbbell topology and run the experiments for a period of 1 sec. The buffer size of the bottleneck link is set to the bandwidth-delay product in all cases (83 Packets or 125 KBytes respect.), the IP data packet size is 1500 bytes.

B. Simulation Results and Discussion

First, we simulate a scenario with 5 elephant flows that start and stop each in a predetermined order to test the convergence and fairness. Figures 2a and 2b show the results for the goodput of the 5 elephant flows scenario. RWNDQ is able to converge faster to the fair-share for each active flow and on average it achieves a better fairness with a lower variance than DCTCP.

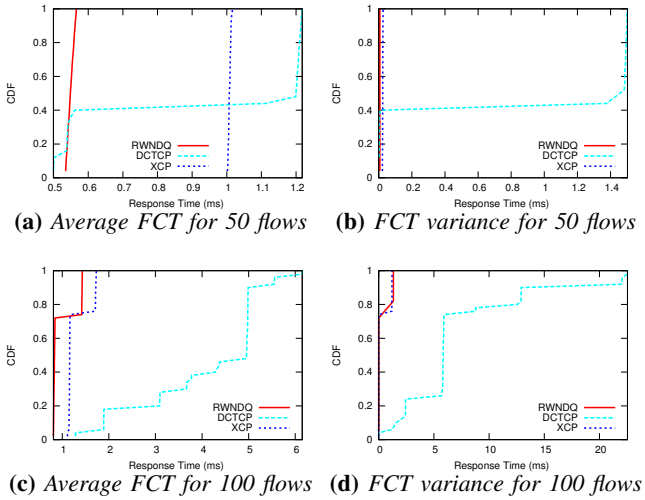


(a) *DCTCP* **(b)** *RWNDQ*

Figure 2: *Goodput of 5 flows that start/stop in a predetermined order showing the convergence speed to the fair-share*

We then simulate two other scenarios with 50 and 100 sources respectively half of which are elephants and the other half mice FTP flows to trigger incast and buffer-bloating situations. All sources start at same time at the beginning, and while elephants keep sending at full speed during the whole simulation period, mice flows who finish their flow very quickly, restart sending for another 5 epochs during the simulation. In each of these epochs the different mice flows start in a random order and each flow sends 10KBytes of data. The interval between the start of two consecutive mice flows is randomly chosen with an average equal to a packet transmission time divided by the number of flows. This allows for the creation of the incast problem where the start times of mice are correlated.

We study the CDF of the average and the variance of the flow completion time (FCT) of mice flows over the incast rounds, packet drop rate from mice flows, the persistent queue size, the goodput of elephant flows and the link utilization in the two scenarios.

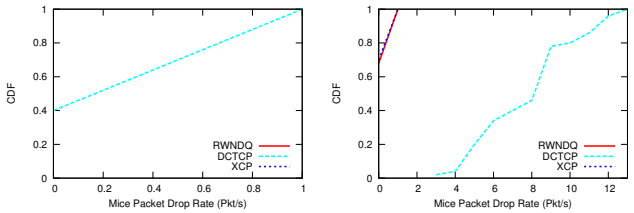


(a) *Average FCT for 50 flows* **(b)** *FCT variance for 50 flows*

(c) *Average FCT for 100 flows* **(d)** *FCT variance for 100 flows*

Figure 3: *CDF of Average FCT and variance for mice*

According to Figures 3a, 3b, 3c and 3d, RWNDQ is able to achieve a faster average FCT compared to DCTCP and XCP and this becomes more conspicuous as the number of flows increases. The variance in the AFCT was comparable to that of XCP, zero for the 50 flows case and less than 10ms in the 100 flows case; in contrast, for DCTCP it reached 33ms in the 100 flows case.

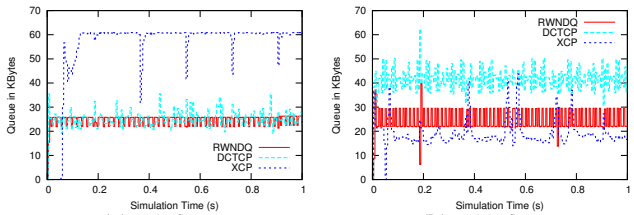


(a) *50 flows* **(b)** *100 flows*

Figure 4: *CDF of the packet drop rate from mice*

Figures 4a and 4b show the drop rate from mice traffic. The figures clearly show that RWNDQ achieves comparable results as XCP and achieves a lower drop probability than DCTCP. Besides, as the number of flows increases, the number of drops increases further, yet still, RWNDQ maintains a lower drop probability compared to DCTCP and XCP. This obviously is the reason for the lower average FCT, as retransmission delay account for the largest portion of high FCTs in our simulations.

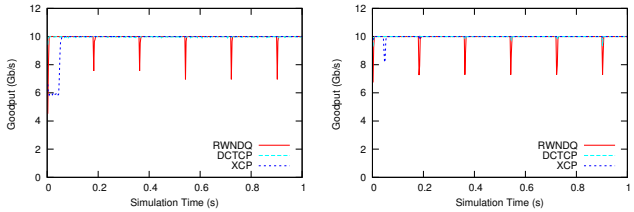
Figures 5a and 5b show the persistent queue length over time. In both cases it is nearly the same for RWNDQ and



(a) *50 flows* **(b)** *100 flows*

Figure 5: *Persistent queue length over time*

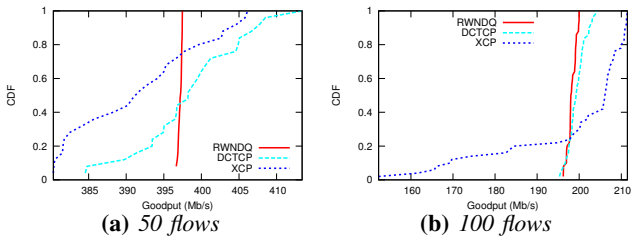
always close to the target queue. This shows the scalability of RWNDQ as it can nearly stabilize the queue length at the target level even when the traffic volume is high. In contrast for DCTCP, the persistent queue in the 100 flows case is more than double its value in the 50 flows case. However, for XCP, with a larger number of competing flows, the queue occupancy decreases as the feedback values are divided among larger estimated number of flows causing a fractional increase in congestion window, which eventually get floored at the source to the nearest integer losing the feedback increments.



(a) *50 flows* **(b)** *100 flows*

Figure 6: *Bottleneck link utilization over time*

Figures 6a and 6b show the bottleneck link utilization and reveal that DCTCP and XCP can achieve nearly full utilization all the time with little decrease during incast, while RWNDQ has a slightly higher decrease in the utilization at the beginning of incast period. This is due to RWNDQ reacting fast and conservatively to the sudden surge of excess traffic. Nevertheless, as shown in Figures 7a and 7b the average goodput of elephants in RWNDQ is comparable to both DCTCP and XCP with significantly improved fairness among competing elephants. This fairness can be attributed to the faster convergence time of RWNDQ and the accurate estimation of congestion level at the switch.



(a) *50 flows* **(b)** *100 flows*

Figure 7: *CDF of the average goodput of elephants*

To summarize this simulation study, RWNDQ seems to be able to smooth oscillations and reach a high link utilization, small queue size, and fair rate allocation among competing flows. Besides, the protocol showed a high degree of robustness in face of varying and sudden traffic surges.

V. Conclusion and Future Work

In this paper, we set to reconcile between the conflicting requirements of elephant and mice flows that are known to make up the lion's share of DCNs traffic. We found that the persistent queue size must be maintained low enough to enable the system to absorb bursts of incast traffic, achieving a faster average flow completion time for mice flows, and sustaining a high throughput for elephants. We proposed the RWNDQ mechanism as a switch-driven flow-aware rate matching algorithm that only relies on the existing flow-control mechanism of TCP to feedback queue occupancy levels to TCP senders. A number of detailed simulations showed that RWNDQ can achieve its goals efficiently while outperforming the most prominent alternative approaches. Last but not least, knowing that in most public data centers the TCP sender and/or receiver are outside the control of the DCN operator, RWNDQ has as a design requirement to avoid modifying the TCP congestion control algorithm to enable true deployment potential in real public DC networks.

References

[1] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking - WREN '09*, pp. 73–82, 2009.

[2] 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. 40, p. 63, 2010.

[3] 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, pp. 345–358, 2013.

[4] 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, p. 303, 2009.

[5] 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 the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pp. 17–28, 2014.

[6] D. Katabi, M. Handley, and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks," in *Proc. ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM'02)*, 2002.

[7] IETF.org, "TCP Extensions for High Performance." http://tools.ietf.org/html/rfc1323.

[8] J. Postel, "RFC 793 - Transmission Control Protocol," 1981. http://www.ietf.org/rfc/rfc793.txt.

[9] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED," *ACM SIGCOMM Computer Communication Review*, vol. 30, pp. 151–160, 2000.

[10] NS2, "The network simulator ns-2 project." http://www.isi.edu/nsnam/ns.

[11] M. Alizadeh, "Data Center TCP (DCTCP)," 2012. http://simula.stanford.edu/%7Ealizade/Site/DCTCP.html.