Efficient Switch-Assisted Congestion Control for Data Centers: an Implementation and Evaluation

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*—This paper discusses the implementation of RWNDQ, a switch-assisted algorithm to improve TCP's congestion control in data center networks (DCNs). In particular we demonstrate how this switch-driven congestion control algorithm can be deployed in a data center via a few simple modifications to the switch software. The proposed mechanism enables the switch to modify the TCP receive-window field in the packet headers to enforce a sending rate at the source, thus avoiding modification to the TCP source or receiver code. This paper describes in detail two implementations, one as a Linux kernel module and the second as an added feature to the well known software switch, Open vSwitch and presents some experimental results from their deployment in a small testbed to demonstrate the effectiveness of RWNDQ in achieving high throughput, a good fairness and short flow completion times for delay-sensitive flows.**

***Keywords*—Congestion Control, Data Center Networks, Flow Control, Implementations, Linux NetFilter, Open vSwitch**

I. Introduction

Data center (DC) environments are used to host a large number of different applications with heterogeneous traffic characteristics and performance requirements. These applications result in a mix of small but time-sensitive traffic flows such as control-messages or web-search traffic (called in the sequel mice) and long lived throughput-inclined flows such as VM-migration (called hereafter elephants). Mice and elephant flows often co-exist within the same DC Network (DCN). Unlike the Internet, DCNs which are characterized by a large bandwidth and small round-trip delay, impose many new challenges to the overlying congestion control mechanisms of TCP, mainly because TCP congestion control mechanism in all its variations *i)* is unaware to the overlying application performance requirements and traffic characteristics; and *ii)* is typically designed to achieve stability and high bandwidth utilization, while only targeting in practice fairness for long live elephant flows under ideal conditions[1].

Unlike the Internet that relies on routers with large buffers, DCNs use switches with small buffers (few megabytes to few hundred megabytes). Round trip delays in DCNs are short (few microseconds to few hundred microsecond) with a relatively large bandwidth (1Gbps to 10Gbps) [1–4]. When such networks are fed with a mixture of mice and elephants, several congestion phenomena that cannot be simply inferred from packet losses and/or delay take place[3, 4]: *i)* Incast traffic congestion: in partition/aggregate applications (e.g., MapReduce) many synchronized mice flows compete to exit from the same congested output port of a switch over a very short period of time. In the presence of small switch buffers they lead to excessive packet drops and timeouts; and, *ii)* Queue-buildup: that occurs as a normal behaviour of TCP when the buffer space is occupied by elephant flows, lead mice flows to experiencing repeated packet drops and unnecessary increase in queuing delays due to timeouts.

Due to the impact and severity of these congestion symptoms on cloud users' experience, much recent research work has been devoted to addressing the shortcomings of TCP in DCNs. These works can in general be categorized into two categories: window based schemes (e.g., [4, 5]) and fast loss recovery based schemes (e.g., [2, 6]).

The major drawback of these schemes is that they all require changes to the sender/receiver TCP stack or they propose a completely new protocol to replace TCP. As the TCP stack is in the VM guest OS and is under the control of the end user, these solutions, despite their effectiveness, turn out to be limited to private DCNs where the TCP protocol can be changed on all VMs. To cope with this issue we advocate a flow-aware approach similar to traditional flow-based system like ATM-ABR or XCP [7], where congestion control in a DCN is treated as a simple problem of flow control between the switch on one side and each TCP source on the other side. In this perspective we proposed in [8] a mechanism called *Receiver window queue (RWNDQ)* that proved to be able to achieve a high efficiency for mice and elephants by keeping a low queue occupancy as well as a good fairness in both short and long term. We have also analysed the stability of RWNDQ mechanism using a simple analytical model and examined its effectiveness through ns2 simulations comparing it to TCP, XCP and DCTCP [8]. Many challenges remain to deploy such mechanism in real systems, the most important one being, how to implement such flow-awareness in the flow-averse IP environment, while maintaining TCP sender/receiver code untouched and without storing per flow states in the switches. In our approach, we track the number of active flows on each switch port by counting SYN-ACK/FIN TCP packets and rely on the TCP receiver-window field in the TCP header to convey the fair-share back to the sources. TCP flow control being a fundamental part of any TCP variant, our proposed mechanism fits-in without any change to the sender nor the receiver. We discuss in this paper how we can achieve this and implement this algorithm effectively in a real system with the following contributions:

[1]Mice flows often do not have enough traffic to achieve their traditional TCP fair share under TCP congestion control

- We describe the implementation of RWNDQ as a run-time loadable Linux kernel module that can be used as a standalone buffer management mechanism in software switches or switches running Linux Network OS[2].
- We also describe the implementation of RWNDQ as a new added feature to the data-path of the well-known industry standard software switch Open vSwitch (OvS)[3] which is widely used in virtualized data centers.
- Finally, we build a small-scale testbed with 12 servers, Gigabit-ethernet switches and OvS on all the servers and in the core switch to study the performance of RWNDQ with multiple bottleneck links, incast traffic and buffer-bloating at both the ToR switches and core switches.

Our results show that RWNDQ handles congestion gracefully, and is able to reconcile mice and elephants by enabling the former to finish their flows quickly and the latter to achieve a high throughput and fairness, even when the network is severely congested.

The remainder of the paper is organized as follow, we first briefly review RWNDQ in Section II. In Section III, we discuss the implementation aspects of the Linux kernel module and show some experimental results. We then show how we added the RWNDQ as a feature of OvS and discuss testbed performance results in Section IV. We finally conclude the paper in V.

II. RWNDQ Algorithm

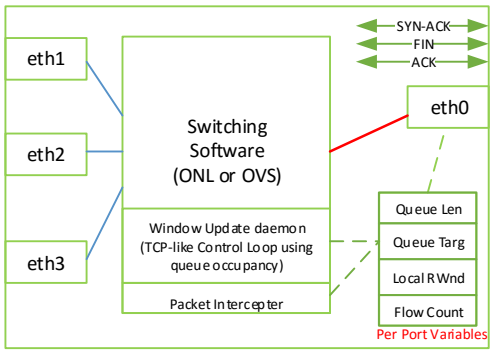


Figure 1: *RWNDQ high level system-view design*

RWNDQ system overview is given in Figure 1. RWNDQ is an algorithm that tries to maintain the queue at the switches below a given target by adjusting the senders rates. It runs in any switching point and maintains several state variables per switch port. It intercepts three types of packets and updates the state information per port accordingly: *i)* SYN-ACK: which establishes a TCP flow on both directions (forward and backward) increments the flow count associated with the output queues of both directions by 1; *ii)* FIN: which terminates a TCP flow on one direction (incoming direction only) decrements the associated flow count of that queue by 1; and *iii)* ACK: which is examined as a candidate to carry back in the receiver window field (*Rwnd*) a window update based on a (switch) local window value of the reverse direction output queue.

[2]For example, Pica8 pronto-3295: http://www.pica8.com/documents/pica8-datasheet-48x1gbe-p3290-p3295.pdf

[3]"Open Virtual Switch Project" http://openvswitch.org/

RWNDQ is an event-driven scheme that deals with two major events: packet arrival, and window update timer expiry events as follows:

A. Window update daemon

As part of the initialization step or if this is the first flow (based on intercepted SYN-ACK), then the current local window is initially set to the target-queue occupancy worth of bytes then, because initially the aggregate bandwidth-delay product is unknown to the switch, RWNDQ enters a slow-start phase to start probing for the corresponding window size. Slow-start phase is terminated as soon as the current queue occupancy exceeds the predetermined queue target. As shown in listing 1, RWNDQ implements a local TCP-Like window control loop that tracks at regular intervals the deviation of the current queue occupancy from the target one. It calculates a ratio of current queue over target which directly controls the accumulated fraction of segments (MSS) added to or subtracted from the window update variable. The algorithm waits for a number of successive updates after which the current value of the per-port local window is updated with the value of update variable (except during slow start where it adds two MSS to the window). Note that RWNDQ waits for a number of accumulated updates before it is used to update the actual value of *Rwnd* that is conveyed to the TCP sender. This enables a highly accurate estimation of the increment, while keeping the number of *Rwnd* rewrites in the packet header reasonably small.

```
1  //Window Update Daemon
2  hrtimer_restart timer_callback(hrtimer *timer)
3  {
4    timerrun=false;//assume no active ports
5   //update the local window of all active ports
6    net_device * dev=first_net_device(&init_net);
7    i=0;
8    while (dev!=NULL && i<devcount)
9    {
10      if(dev->ifindex==devind[i] && conncount[i]>0)
11      {
12      timerrun=true;
13     backlog=dev->qdisc->qstats.backlog;
14     //target is set to 25\% of buffer length
15     //left shift by 2 => divide by 4 => 25\%
16     target=(dev->qdisc->limit)>>2;
17     //exit slowstart phase
18        if(slowstart[i] &&  backlog >= target)
19          slowstart[i]=false;
20     //slowstart off =>inc/dec using the difference
21        if(!slowstart[i])
22          incr[i] += target - backlog;
23        else  //slowstart on =>add two segments
24          incr[i] += 2 * MSS[i];
25     //update local window after M inc/dec(s)
26        if (count == M)
27        {
28          localwnd[i]+=incr[i]/M;
29          localwnd[i]=MIN(localwnd[i], 65535 * ↩
                conncount[i]);
30          localwnd[i]=MAX(localwnd[i], TCP_MIN_MSS ↩
                * conncount[i]);
31          wnd[i]=localwnd[i]/conncount[i];
32          incr[i]=0;
33         }
34         i++;
35       }
36       dev = next_net_device(dev);
37     }
38   //reset counter
```

```
    if(count == M)
      count=0;
    else
      count++;
  //if there are active connections rearm timer
    if(timerrun == true)
    {
      reschedule_timer(timer);
      return HRTIMER_RESTART;
    }
    else
      return HRTIMER_NORESTART;
}
```

Listing 1: *RWNDQ Window Update Daemon*

B. Packet arrival

As shown in listing 2, for each new flow (again based on intercepted SYN-ACK packets), the current local window of the incoming and outgoing port is divided equally among all new active flows. However, for each torn down flow (based on intercepted FIN packets), the current window of the incoming port is redistributed equally among all currently active flows. Finally, If the ACK bit is set, the receive window field *Rwnd* of this Packet is updated with the calculated per-port fair-share local window if it is smaller than the receive window value in TCP header.

```
//Packet intercepter and modifier
void rwndq_packet_arrival(sk_buff *skb, net_device↩
     *in, net_device *out)
{
 //in port is mapped to index i
 //out port is mapped to index j
 //New connection setup
 if(tcp_header->syn && tcp_header->ack)
 {
    conncount[i]+=1;//Increment connections on ↩
        incoming port i
    conncount[j]+=1;//Increment connections on ↩
        outgoing port j
  if(conncount[i] >= 2)
      wnd[i] = wnd[i] * (conncount[i]-1) / ↩
          conncount[i];
  if(conncount[j] >= 2)
      wnd[j] = wnd[j] * (conncount[j]-1) / ↩
          conncount[j];
 }
 //Existing connection tear-down
 if(tcp_header->fin || tcp_header->rst)
  {
   conncount[i]-=1;
    if(conncount[i] >= 1)
      wnd[i]= wnd[i] * (conncount[i]+1) / ↩
          conncount[i];
 }
 //Check for possible window modification
 if(tcp_header->ack)
  {
  if(wnd[i] < tcp_header->window)
  {
   __be16 oldwnd = tcp_header->window;
   tcp_header->window = wnd[i];
   csum_replace2(tcp_header->check,oldwnd,wnd[i]);
  }
 }
}
```

Listing 2: *RNWDQ Packet Intercepter and Modifier*

RWNDQ can maintain a very small buffer occupancies which allows the switch's small buffer to absorb transient traffic bursts while keeping the line busy. Therefore it achieves a high throughput for elephants and very small queuing delay and low loss probability for mice.

RWQND as discussed uses proportional increase, proportional decrease rather than AIMD. Yet RWNDQ is still stable due to the locality of window control mechanism with the managed queues (i.e., local control loop). As soon as the queue occupancy increase above or decrease below the target threshold, the local window is shrunk or expanded in proportion, which ensures an average persistent queue occupancy level equal to target. Furthermore the increase/decrease amount is equally divided among all ongoing flows which ensures short and long term fairness among competing flows unlike end-to-end systems based on TCP.

III. RWNDQ DATA CENTER WIDE DEPLOYMENT

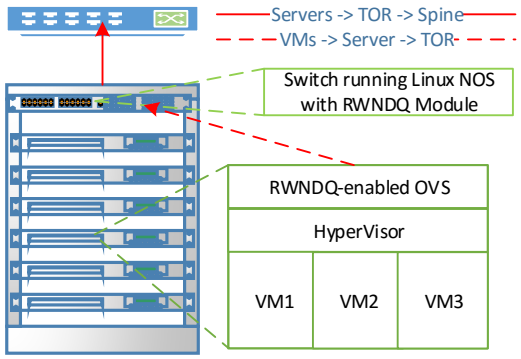


Figure 2: *RWNDQ-enabled Data Center Architecture*

Because of the promising performance shown by RWNDQ in simulation studies, we have implemented the algorithm in a real testbed to ascertain and demonstrate its practical feasibility. Two options were implemented:

- Linux Kernel Module: RWNDQ can be implemented using the Linux NetFilter framework [4] as the NetFilter is a good candidate for intercepting incoming and outgoing packets. Also the NetFilter enables incrementing the protocol stack very easily via pre-registered hooks in the processing pipeline of the network stack;
- Open vSwitch: RWNDQ can also be implemented as an added feature to OvS by patching the OvS data-path kernel module to add RWNDQ's processing logic.

Typically, a data center network consists of servers and switches interconnecting them. In oversubscribed data centers, the contention points in the network, are within the server on the outgoing interfaces between competing VMs and on the uplinks to upper layers (i.e the link from Top of Rack switch to the core switch) where multiple servers in a single rack share the uplink reach servers in other racks. In our RWNDQ system design, as shown in Figure 2, we propose to use hardware switches running Linux Network OS such as Open Networking Linux (ONL) [5] or PicOS [6] on ToR and spine level hardware

[4]NetFilter: Packet filtering framework for linux http://www.netfilter.org/
[5]Open Network Linux http://opennetlinux.org
[6]PicOS: http://www.pica8.com/white-box-switches/white-box-switch-os.php

switches to interconnect servers within the racks. The switches with PicOS can support RWNDQ as a loadable kernel module or as a patched Open vSwitch. In addition, VMs within servers are interconnected via a software OvS which is the most popular choice for most cloud management frameworks like OpenStack. RWNDQ as a kernel module and OvS-patch provide a potential for an easy deployment in production data centers at all different switching levels and possible congestion points in the network.

A. RWNDQ as a loadable kernel module

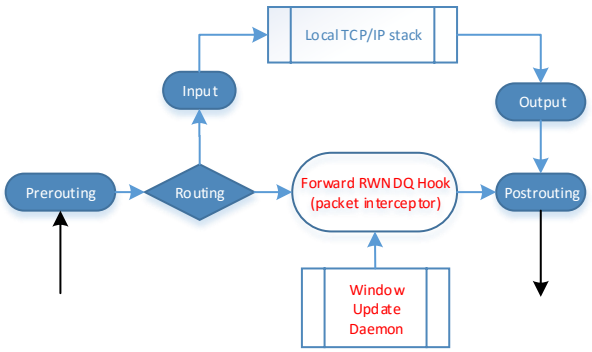


Figure 3: *NetFilter-based RWNDQ packet processing pipeline*

RWNDQ as a loadable kernel module is implemented using the NetFilter framework of the Linux operating system. In a non-virtualized environment, RWNDQ works as a hook that attaches to the data forwarding path in the Linux kernel just above the NIC driver and below the TCP/IP stack. This is a clean way of deploying RWNDQ which does not touch the TCP/IP implementation of the host operating system, making it easily deploy-able in production DCNs. In what follows, we introduce our Linux implementation which is used in this section's experiments.

RWNDQ is implemented as a NetFilter hook as shown in Figure 3. We insert the NetFilter hook at the forwarding stage of packet processing to intercept all forwarded TCP packets not destined to the host machine. Forwarding stage is executed right after the routing decision has been taken and immediately before the post-routing processing. As explained previously in the RWNDQ algorithm in section II, TCP packet headers are examined and the processing is determined based on the SYN-ACK, FIN and ACK flag bits. In addition, since in DCNs transmission and propagation delays are in the microsecond time scale, Linux kernel timers based on the HZ tick rate traditionally used in the protocol stack and the OS, are not accurate enough to keep track of the queue occupancy as they are in the millisecond time scale; therefore, we invoke Linux high-resolution timers to deal with this[7]. The high resolution timer shown in the figure as "Window udpate daemon" is used to trigger switch's local per-port receiver window values updates based on the observed queue occupancy more accurately. The operations of the RWNDQ kernel module are described as follows:

- When a SYN-ACK packet is captured by the NetFilter hook, we increment the connection counter for both the ingress and egress Ethernet port and update their local window variables respectively[8]. Note that, SYN-ACK packets is sent only when TCP connection is established by the destination host of the connection and by default all TCP connections are full-duplex.
- When a FIN packet is captured by the NetFilter hook, we decrement the connection counter for the ingress port only and update its local window variable. Note that, FIN packets are sent only when one side of the TCP connection finishes the transport of its application data and the other side of the TCP connection can still send data while the host which sent the FIN operates in half-closed state until it receives a FIN from its partner.
- When an outgoing ACK packet is captured by the NetFilter hook, its receive window in TCP header will be checked against the local window, then the receiver window value is updated only if the local receive window value, pre-computed by the RWNDQ local window update mechanism, is smaller.
- If the window is updated, the checksum of the packet is recomputed which is done using a kernel built-in function *"csum_replace2"*, which implements the update efficiently.
- The high resolution timer is responsible for triggering the local window update function on regular intervals, it is triggered on intervals smaller than the measured RTT in the network, in our setup RTTs are observed to be within a few hundred microseconds.
- For each timer expiry, we calculate the increment value using the method described in RWNDQ algorithm in Section II.
- When the timer expires M times consecutively, the local window value is updated using the accumulated increment values over the last M intervals, the increment variable is reset and a new window update cycle starts from this point.

B. Testbed setup

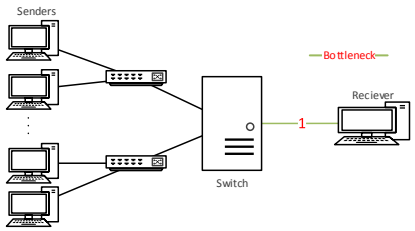


Figure 4: *A dumbell-like setup to test RWNDQ Kernel Module*

To experiment with the Linux RWNDQ kernel module, we set up a single bottleneck testbed as shown in Figure 4. The testbed consists of 6 Lenovo and 7 Dell desktops configured with core2duo processors and 4G of ram. One of the Dell desktops acts as the switch and it is equipped with 3 1Gbps Ethernet cards. In this setup, 5 machines and the switch are running Ubuntu 14.04 Desktop Edition with Linux kernel V3.13.0.34, while the other 6 machines and the receiver (master) are running CentOS 6.6 with Linux kernel V3.10.63. All

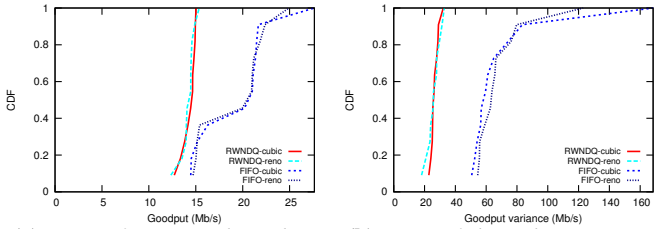
[7]https://www.kernel.org/doc/Documentation/timers/hrtimers.txt

[8]Note that, assuming current window value is in a stable state and optimal, by adding or deleting a TCP connection, the window value needs to be redistributed equally and fairly over the new number of active TCP flows

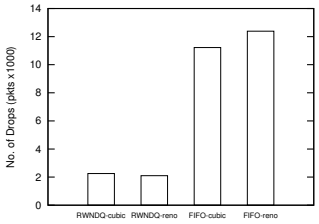
machines are running an Apache web server hosting the default small *"index.html"* webpage of size 11.5KB. We rely on two well-known measurement applications for our experiments, iperf [9] for generating elephants and Apache benchmark [10] for generating synchronized mice. The base RTT in our testbed is around $\approx 200\mu$s. We allocate a static buffer size of 85.3KB to all ports in the network using Linux Traffic Control (Linux TC). In all experiments, we set up the queuing discipline *Qdisc* of each Ethernet port in the network to *bfifo* queue with limit of 85.3KB bytes. This value matches the buffer size for each port in a switch like pronto 3295, that has 4MB of shared buffer memory used by 48 ports, leaving a buffer of 85.3KB for each port on the switch. We evaluated the experiments with both cubic TCP and new-reno TCP, the only available congestion control mechanisms in both Linux kernel versions (3.10 and 3.13).

C. Experimental Results

Eleven iperf traffic flows (elephants) are started at the same time from each of the senders towards the receiver which is connected to one of the switch ports.The flows send continuously for 50 secs and throughput samples are collected over 0.5 secs intervals. At the 20^{th} second, each sender starts Apache Benchmark to request *"index.html"* 1000 times (mice) and report statistics on the completion times.



(a) *CDF of average throughput* **(b)** *CDF of throughput variance*

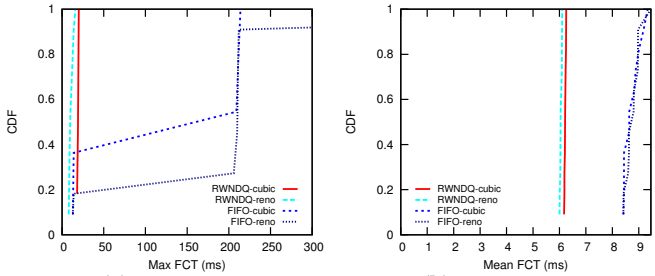


(c) *Total drops of the bottleneck link*

Figure 5: *Elephant TCP flows performance: RWNDQ vs FIFO in a the Linux testbed with one bottleneck-link with new-reno and cubic TCP sources*

Figure 5a and 5b show that RWNDQ helps both TCP variants to achieve a very close fair-share throughput to TCP-FIFO yet reduces the variations of the reclaimed throughput during elephant sessions, even with sudden surges of mice traffic. Figure 5c shows RWNDQ's ability to significantly reduce packet drops at the bottleneck link by $\approx 80 - 85\%$.

The reduction of packet drops benefits mice flows by avoiding unnecessary timeouts. As Figure 6a shows, mice tail flow completion time (FCT) is less than 200ms, which is the



(a) *Max FCT* **(b)** *Average FCT*

Figure 6: *Mice TCP flows performance: RWNDQ vs FIFO in a the Linux testbed with one bottleneck-link with new-reno and cubic*

default RTO_{Min} in Linux. Finally, according to Figure 6b, RWNDQ allows competing mice to finish quickly and at approximately the same time.

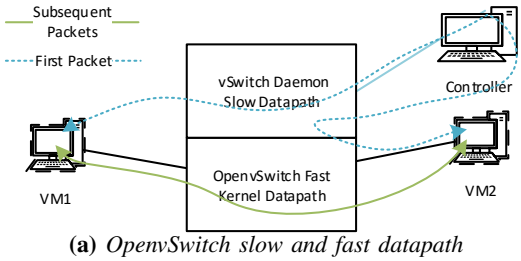
IV. RWNDQ-enabled Open vSwitch

We further investigated the implementation of RWNDQ in OvS as this latter already implements SDN based flow tracking. We patched the Kernel data-path modules of OvS with the same functions described earlier in the RWNDQ Linux-kernel module. In this case however, we did not use the NetFilter hook, we instead added RWNDQ functions in the processing pipeline of the packets that pass through the kernel datapath module of OvS. In a virtualized environment, RWNDQ-enabled OvS can process the traffic for inter-VM, Intra-Host and Inter-Host communications. This is an efficient way of deploying RWNDQ on the host operating system of the switch by only applying a patch and recompiling OvS module, making it easily deploy-able in today's production DCs.

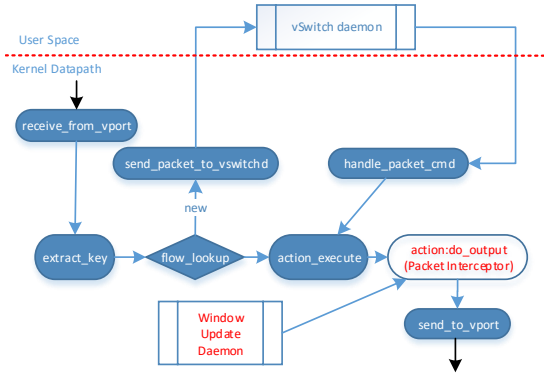
As show in Figure 7, OvS is mainly composed of two parts, the data path kernel module and the user-space vSwitch daemon that communicates with the controller using OpenFlow protocol over encrypted SSH connections. OvS is flow-aware by design and all flow decision entries in the forwarding table are inserted by a local or a remote controller. Whenever, a packet arrives at any port of the switch, its flow key is hashed and examined against current active flows in the table. If the entry for that flow could not be found, the packet is immediately forwarded to the controller for establishing the identity of this flow and setting up the forwarding entries in all involved switches of the network. Primarily, any packet is processed by the kernel fast data-path only if its flow entry is active in the forwarding table, in such case, the packet is forwarded immediately without experiencing any further delays. RWNDQ's packet interceptor and its packet handling logic described in Section III-A are inserted in processing of *do_output* action function as shown in Figure 7b. TCP packets being forwarded are intercepted and their window is updated if necessary. Local per-port window values are updated on a regular basis by the window update daemon.

A. Testbed Setup

For experimenting with our patched OvS, we set up a testbed as shown in Figure 8, it is similar to the testbed in



(a) *OpenvSwitch slow and fast datapath*



(b) *OpenvSwitch-based RWNDQ packet processing pipeline*

Figure 7: *RWNDQ OpenvSwitch-based implementation*

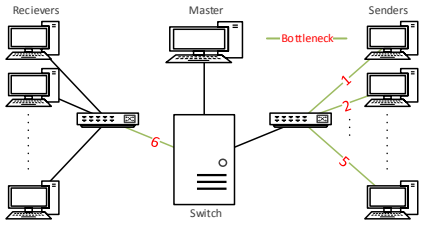


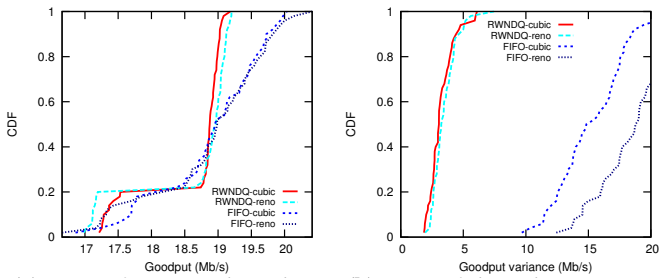
Figure 8: *Small testbed for RWNDQ-enabled OvS*

Section III, however, in this case, all machines' internal ports are connected to the patched OvS and the CentOS and Ubuntu hosts are connected to different 1 Gb/s D-Link dumb-switch. Here also, different scenarios are set up to reproduce both incast and buffer-bloating situations, however in this case multiple-bottleneck links in the network exist as shown in Figure 8. The bottlenecks at the senders are created by creating multiple ports on the OvS and binding an iperf or an Apache process to each one of them.

B. Experimental Results

The goals of the experiments are to: *i)* show that with the support of RWNDQ, TCP can support many more connections and maintain high link utilization; *ii)* show that with the support of RWNDQ, TCP can overcome incast congestion situations in the network; *iii)* measure RWNDQ's impact on the FCT of mice flows; and, *iv)* explore RWNDQ's performance in buffer-bloating situations where mice compete with elephants.

1) Incast Scenarios: We produce an incast-like scenario with synchronized senders all converging to the same output port resulting in excessive pressure on the output buffer in links 1-5 as well as link 6. First, we generated 10 iperf clients at same time from each of the 5 senders destined to a separate iperf server listening on a separate port on the receivers. This results in 50 senders continuously sending for 50 secs and iperf is set to generate the throughput samples over 0.5 sec intervals. In the following we show the CDF of the average achieved throughput, the variance of the throughput samples and the total packet drops experienced at the bottleneck links during the experiment. Figure 9 shows that in a medium load situation, RWNDQ helps TCP in achieving a balanced distribution of bandwidth among competing senders and reduces the variations of their reclaimed bandwidth during the lifetime of a TCP connection. Table Ia clearly shows how RWNDQ switch queue management is able to reduce the number of packet drops at bottleneck links by $\approx 92 - 99\%$ (nearly two orders of magnitude), reducing considerably unnecessary timeouts for TCP connections and allowing the flows to finish at approximately the same time. This due to RWNDQ dividing the effective window (bandwidth-delay product + target buffer length) equally among competing flows, allowing them to achieve nearly the same throughput and hence very similar flow completion times.



(a) *CDF of average throughput* **(b)** *CDF of throughput variance*

Figure 9: *Comparison of TCP's performance with RWNDQ vs. FIFO with 50 elephants*

We repeat the same experiment but this time we increase the number of iperf (elephant) senders per host to 40 (resulting in a total 200 elephants). Again, Figure 10 supports our claims, and shows that even in a high load, RWNDQ still helps TCP (new-reno and cubic) achieve a balanced distribution of bandwidth and maintains a very low variation of throughput for TCP connections involved in the incast. We observe that the variation for some TCP flows without RWNDQ reaches ≈400Mbps, the reason being, for some time intervals, a few flows grab most of the bandwidth while the others achieve nearly zero throughput. Table Ib shows that RWNDQ is still able to keep a very low packet drop rate by one order of magnitude compared to TCP without the assistance of RWNDQ mechanism at the switch.

2) Buffer-Bloating Scenarios: We reproduce a buffer-bloating scenario in which mice traffic compete with elephant flows to see if RWNDQ can reconcile the two classes. Similar to the previous experiment, we first generate 10 synchronized iperf elephant connections continuously sending for 50 secs from each sender resulting in 50 elephants at link 6. We use Apache benchmark to request *"index.html"* webpage (representing mice flows) from each of the web servers ($6 \times 5 = 30$ in total) running on the same machines where elephants are sending. Note that, we run Apache benchmark, at the 20^{th}sec, requesting

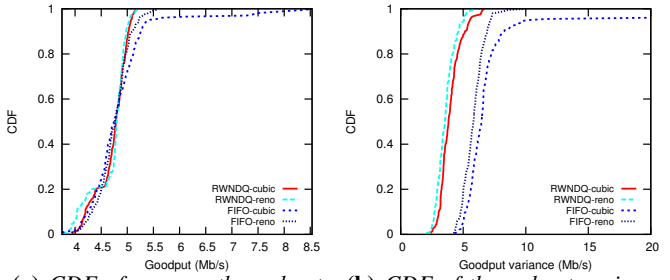
Table I: *Number of packet drops experienced at each of the 6 bottleneck links labeled 1 to 6 in Figure 8*

(a) *50 elephants scenario*

	Reno		Cubic	
	RWNDQ	FIFO	RWNDQ	FIFO
1	10	4992	33	4605
2	5	4913	21	4548
3	10	4676	19	4319
4	18	4860	29	4530
5	12	4857	44	4520
6	531	331	320	357

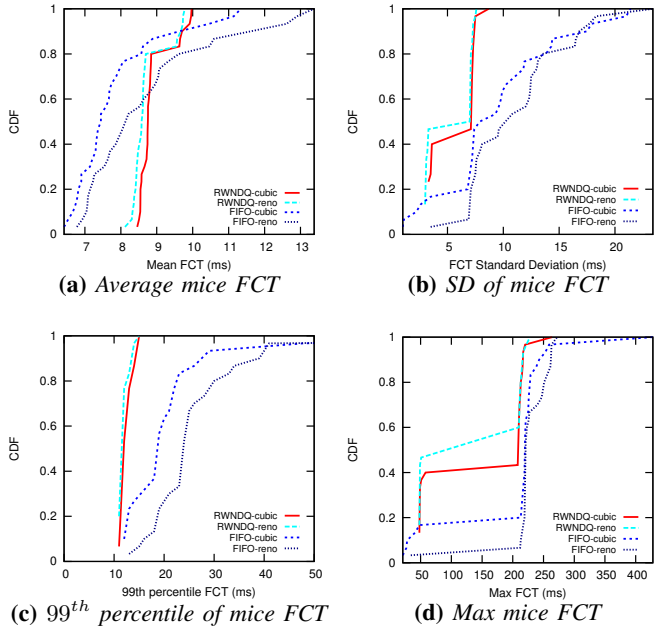
(b) *200 elephants scenario*

	Reno		Cubic	
	RWNDQ	FIFO	RWNDQ	FIFO
1	1750	30934	2184	30422
2	1671	30851	2361	30767
3	2544	27486	2418	28276
4	1632	30620	2152	30210
5	1547	30860	2249	30540
6	3394	12901	3516	23432



(a) *CDF of average throughput* **(b)** *CDF of throughput variance*

Figure 10: *Comparison of TCP's performance with RWNDQ vs. FIFO with 200 elephants*



(a) *Average mice FCT* **(b)** *SD of mice FCT*

(c) 99^{th} *percentile of mice FCT* **(d)** *Max mice FCT*

Figure 11: *Comparison of mice FCT for RWNDQ vs. FIFO where 50 elephants compete with 30 mice*

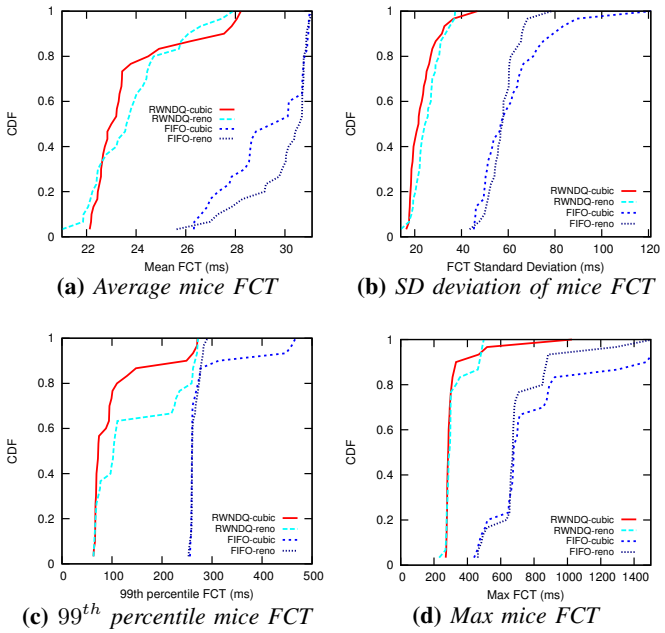
the webpage 1000 times then it reports different statistics over the 1000 requests. The performance of elephants was close to what has been presented in the incast scenario experiments. Now, Figure 11 shows that, in medium load, RWNDQ achieves a good balance in meeting the conflicting requirements of elephants and mice. The competing mice flows benefit under RWNDQ by achieving a nearly equal FCT on average with very small standard deviation compared to TCP with FIFO as shown in Figure 11a and 11b. In addition, as RWNDQ efficiently regulates the flows and keeps the drop rate near to zero, in Figure 11c, the 99^{th} percentile for RWNDQ never crosses the 200ms threshold which is the default RTO_{min} of Linux, as opposed to TCP with FIFO which can be attributed to the timeouts caused by high drop rates. In Figure 11c, the maximum FCT $\approx 40-60\%$ of the flows are below the 200ms with RWNDQ compared to only $\approx 1-18\%$ for TCP with FIFO.

Again, we repeat the high load experiment with 200 elephants and introduce the 30 competing mice. As shown in Figure 12, RWNDQ is able to satisfy the requirements of latency-sensitive mice even-though they are outnumbered by elephants. Figure 12a and 12b show that mice flows are not blocked by the bandwidth-hogging elephants. The mean FCT under RWNDQ are small and the CDF curve is smooth, according to the standard deviation, in contrast to what is achieved with FIFO. In addition, Figure 12c and 12d show that, the tail and the 99^{th} percentile of the FCT of TCP with FIFO is experiencing timeouts as indicated by FCT values of over 250ms. Meanwhile RWNDQ avoids timeouts by managing the queue efficiently and hence it greatly reduces the FCT of mice on the tail 99^{th} percentile on average by $\approx 60\%$.

3) RWNDQ system overhead: To quantify system overhead introduced by the RWNDQ packet intercepting and modifying module, we measured CPU usage on the server operating as the switch between the other servers which is equipped with Intel Core2 Duo CPU running at 2.13GHz and 4 GBytes of Ram. We rerun the high load experiment with 200 elephants and the 30 competing mice. We achieved a high link utilization of $\approx$ 900-935 Mbps goodput while the extra CPU usage introduced by RWNDQ is $\approx$ 1% compared with the case where the RWNDQ module is not enabled.

4) Summary of the experimental Results: In summary the experimental results reinforce the results obtained in the simulation study conducted in [8]. In particular, they show that:

- RWNDQ helps in reducing mice traffic latency and maintains sufficient throughput for elephants. In the experiments both elephants and mice are able to achieve their requirements as stated in the Introduction.
- RWNDQ can easily handle congestion, in low to high load incast or buffer-bloating scenarios, while nearly saturating the link at rate of $\approx$ 900-935 Mbps, which matches our



(a) *Average mice FCT* **(b)** *SD deviation of mice FCT*

(c) 99^{th} *percentile mice FCT* **(d)** *Max mice FCT*

Figure 12: *Comparison of mice FCT with RWNDQ vs. FIFO where 200 elephants compete with 30 mice*

findings from our NS-2 simulations.

- RWNDQ achieved all this without any modification to the TCP congestion control mechanism at the source nor to the receiver and seems to scale well in our testbed.

V. Conclusion and Future Work

In this paper, our target was to show the ease of implementation for RWNDQ mechanism and its immediate deployability in data center networks. RWNDQ was built to fulfil the non-compatible requirements of elephant and mice flows that account for the majority of datacenter traffic. RWNDQ is designed to maintain a small persistent queue size to give room for the available buffer space to absorb any transient bursts of incast traffic. Hence, it can decrease the average flow completion time of mice flows, yet maintain a high throughput for elephants. RWNDQ is a switch-assisted flow-control system that builds on top of the existing flow-control of TCP to feedback queue occupancy levels to TCP senders. Because simulation results showed how promising is RWNDQ, and motivated by the demand for an improved congestion control mechanism for public DCNs where changes to the TCP used by the tenants is precluded, we have designed RWNDQ as a ready to deploy switch algorithm in real public DC networks. In addition, to prove the realistic feasibility of our approach, we implemented RWNDQ as a standalone Linux kernel module easily deployable on hardware switches running a Linux network OS and as an added feature to the well known open vSwitch kernel datapath module for deployment in current virtualized DC networks.

The results of our experiments strongly suggest that a switch-based approach like RWNDQ is a good approach to be able to handle incast and buffer-bloating situations simultaneously. In the future, we plan to test RWNDQ on real hardware switches with a larger scale testbed; to this end we have already acquired two bare-metal (white-box) switches with a number of high end servers equipped with a total of hundred cores and 96 interface cards, which enable us to test our implementations under stress with several thousands of mice and elephant flows.

References

[1] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. a. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *Proceedings of the 6th Usenix Conference on File and Storage Technologies (Fast '08)*, pp. 175–188, 2008.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] A. M. Abdelmoniem and B. Bensaou, "Reconciling mice and elephants in data center networks," in *IEEE 4th International Conference on Cloud Networking (CloudNet)*, (Niagara Falls, Canada), Oct. 2015. Copy of the manuscript is avaliable upon request.

[9] iperf, "The TCP/UDP Bandwidth Measurement Tool." https://iperf.fr/.

[10] Apache, "Apache HTTP server benchmarking tool." http://httpd.apache.org/docs/2.2/programs/ab.html.