Enforcing Transport-Agnostic Congestion Control via SDN in Data Centers

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*—To meet the deadlines of interactive applications, congestion-agnostic transport protocols like UDP are increasingly used side by side with congestion-responsive TCP. As bandwidth is not totally virtualized in data centers, service outage may occur (for some applications) when such diverse traffics contend for the small buffers in the switches. In this paper we present SDN-GCC, a simple and practical software-based congestion control mechanism that puts monitoring and control decisions in a centralized controller and traffic control enforcement in the hypervisors on the servers. SDN-GCC builds a congestion control loop between the controller and hypervisors without assuming any cooperation from tenants applications (i.e., transport protocol) ultimately making it deployable in existing data centers without any service disruption or hardware upgrade. SDN-GCC is implemented and evaluated via extensive simulation in ns2 as well as real-life small-scale test-bed experiment.**

***Index Terms*—Congestion Control, Data Center Networks, Rate Control, Open vSwitch, Software Defined Networks, Virtualization**

I. Introduction

To achieve tenants isolation and use resources more effectively, resource virtualization has become a common practice in today's public datacenters. In most cases, each tenant is provisioned with virtual machines with dedicated virtual CPU cores, memory, storage, and a virtual network interface card (NIC) over the underlying shared physical NIC. Typically, tenants can not assume predictability nor measureability of bounds on network performance, as no mechanisms are deployed to explicitly allocate and enforce bandwidth in the cloud. Nevertheless, cloud operators can provide tenants with better virtual network management thanks to the recent development in control plane functions. For example, Amazon introduced "Virtual Private Cloud (VPC)" [7] to allow easy creation and management of tenant's private virtual network. VPC can be viewed as an abstraction layer running on top of the non-isolated shared network resources of AWS's public cloud. Additionally, Software Defined Networking (SDN) [27] is effectively deployed to drive inter- and intra-datacenter communications with added features to make the virtualization and other network aspects easy to manage. For example, both Google [18] and Microsoft [15] have deployed fully operational SDN-based WAN networks to support standard routing protocols as well as centralized traffic engineering.

On the other hand, the data plane in intra-datacenter networks has seen little progress in apportioning and managing bandwidth to overcome congestion, improve efficiency, and provide isolation between competing (greedy) tenants. In principle, isolation can simply be achieved through static reservation [9, 14], where tenants can enjoy a predictable, congestion-free network performance. However, static reservations lead to inefficient utilization of the network capacity. To avoid such pitfall, tenants should be assigned minimum bandwidth by using the hose model [12] which abstracts the collective VMs of one tenant as if they are connected via dedicated links to a virtual switch (vswitch). In such setup, different VMs may reside on any physical machine in the datacenter, yet, each VM should be able to send traffic at its full rate as specified by the vswitch abstraction layer. Such VMs should enjoy the allocated rate regardless of the traffic patterns of co-existing VMs and/or the nature of the workload generated by competing VMs.

The following are the necessary elements that can be incorporated together for this purpose:

- An intelligent and scalable VM admission mechanism within the datacenter for VM placement where minimum bandwidth is available. To facilitate this, topologies with bottlenecks at the core switches (such as uplink over-subscription or a low bisection bandwidth) should be avoided if possible.
- A methodology to fully utilize the available high bisection bandwidth (e.g., a load balancing mechanism and/or multi-path transport/routing protocols).
- A rate adaptation technique to ensure conformance of VM sending rates to their allocated bandwidth, while penalizing misbehaving ones.

A number of interesting research works have investigated more or less successfully the first two elements of this framework [5, 11, 13, 30]. In [5, 13], highly scalable network topologies offering a 1:1 over-subscription and a high bisection bandwidth were proposed. These topologies are shown to be easily deployable in practice and can simplify the VM placement at any physical machine with sufficient bandwidth to support the VM. Efficient routing and transport protocols [11, 30] were designed for DCN to achieve a high utilization of the available capacity. Finally, in terms of traffic control, much of the recent work [6, 34] focused on restructuring TCP

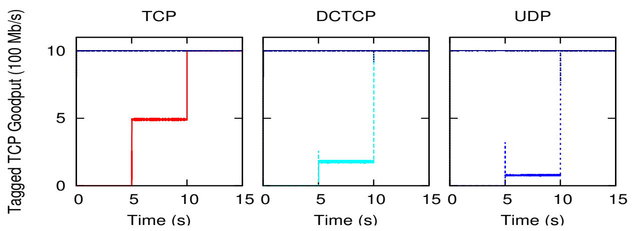


Figure 1: *Goodput of the tagged TCP flow and aggregate goodput of the two flows (navy blue line). At t=0 competitor and at t=5 tagged TCP flow starts while at t=10 competitor stops*

congestion control and its variants to efficiently utilize and fairly share bandwidth among flows (in homogeneous deployments). However, these techniques fall short of providing true isolation among tenants (e.g., a tenant may gain more bandwidth by opening parallel connections or by using aggressive transport protocol like UDP). It is common, in mutli-tenant environments, that non-homogeneous transport protocols co-exist leading to starvation of the cooperative ones [17]. To illustrate this problem via simple means, we conduct a simulation in which we compare the performance of a tagged ECN-enabled TCP NewReno flow that competes with i) another TCP flow of same type, ii) another TCP variant designed for data centers (i.e., DCTCP which deployed in a number of datacenters [20]), and; iii) another congestion-agnostic transport protocol (i.e., UDP which is used in memcached clusters of Facebook [25]). Similar to what was already known from the Internet, Fig. 1 shows that, homogeneous TCP deployments in data centers can achieve fairness, in contrast to heterogeneous deployments. We observe that TCP losses 64% and 84% of its allocated rate to DCTCP and UDP, respectively.

In this paper, we propose a SDN-based generic congestion control (SDN-GCC) mechanism to address this issue. We first introduce the idea behind SDN-GCC in Section II, then discuss our proposed methodology and present SDN-GCC framework in Section IV. We show via ns2 simulation how SDN-GCC achieves its requirements with high efficiency in Section VI. The testbed experiments are presented in Section VII[1]. Finally, we conclude the paper in Section VIII.

II. Transport Isolation Problem

With the recent introduction of a significant number of new transport protocols designed for DC networks in addition to the existing protocols, the following three challenges emerged: *i)* most such protocols are agnostic to the nature of the VM aggregate traffic demands leading to inefficient distribution of the network capacity among competing VMs (for instance a VM could gain more bandwidth by opening parallel TCP connections); *ii)* many versions of TCP co-exist in DC networks (e.g., TCP NewReno/MacOS, compound TCP/Windows, Cubic TCP/Linux, DCTCP/Linux, and so on), leading to further inefficiency in addition to unfairness, and; *iii)* many DC applications rely on UDP to build custom transport protocols (e.g., [25]), that are not responsive to congestion signals, which exacerbate the unfairness to the point of causing starvation. While such problems have been revealed in the context of Internet two decades ago, recent studies [17, 20] have confirmed that such problems of unfairness and bandwidth inefficiency also exist in DCNs despite their characteristically small delays, small buffers and different topologies from those found in the Internet. As a consequence, a new solution to the problems of congestion in DC networks is needed. Such solution must be attractive to cloud operators and cloud tenants alike.

In particular, with the emergence of software defined networking, we see an opportunity to invoke the powerful control features and the global scope provided by SDN to revisit the problem from a different perspective with additional realistic design constraints. As such we propose a solution with the following intuitive design requirements: R1) simple enough to be readily deployable in existing production datacenters; R2) agnostic to (or independent of) the transport protocol; R3) requires no changes to the tenant's OS (in the VM) and makes no assumption of any advanced network hardware capability other than those available in commodity SDN switches; R4) creates a minimal overhead on the end-host.

All of today's communication infrastructure from hardware devices to communication protocols have been designed with requirements derived from the global Internet. As a result to cope with scalability and AS autonomy, the decentralized approach has been adopted, relinquishing all intelligence to end systems. Yet, to enable responsiveness to congestion regardless of the transport protocol capabilities and in time-scales that commensurate with data center delays, it is preferable to adopt centralized control as it provides a global view of congestion and is known to achieve far better performance [22, 24]. Nevertheless to reconcile existing hardware and protocols (designed for distributed networks) with the centralized approach, we impose design requirements R1-R4 on SDN-GCC. As such the core design of SDN-GCC relies on outsourcing the congestion control decisions to the SDN controller while the enforcement of such decisions is carried out by the end-hosts hypervisors.

III. Related Work

A number of recent proposals implemented different system designs for cloud network resources allocation. "Seawall" [33] is a system designed solely for sharing network capacities by achieving per-VM max-min weighted allocations using explicit end-to-end feedback messaging for rate adaptation. Seawall adds new encapsulation protocol to network stack on top of transport headers which incurs a large processing and messaging overhead as well as rendering it into a non middlebox-friendly solution. "Secondnet" [14] is a system proposed to divide network among tenants via rate limits enforcements, however, it only supports static bandwidth reservation among tenants' VMs. "EyeQ" [19] adopts per-VM max-min weighted fair sharing in the context of a full bisection bandwidth datacenter topology. Its downside is the design assumption that congestion is limited to first and last hops. "RWNDQ" [1, 3] is a fair-share allocation AQM for TCP in data

[1]Simulation and implementation code can be requested from the authors and later they will be made publicly available at http://ahmedcs.github.io/SDN-GCC

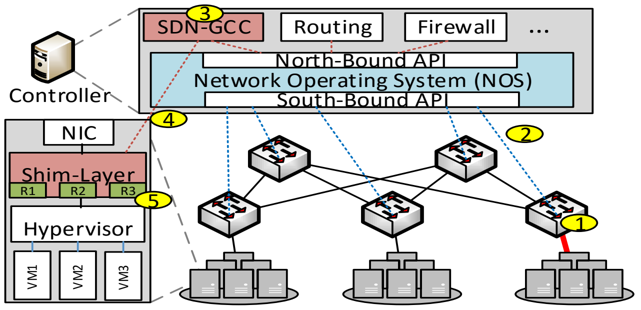


Figure 2: *SDN-GCC high-level system design: 1) congestion point; 2) network statistics; 3) congestion tracking; 4) congestion notification; 5) rate adjustment.*

centers, however it resolves contention among flows using TCP as their transport. "HyGenICC" [23] is an IP-based congestion control mechanism that relies on a collaborative information exchange between hypervisors. The solution involves the use of ECN marking as congestion indication which is aggregated and fed back between hypervisors to enable network bandwidth partitioning through dynamic (adaptive) rate limiters. In spite of the appealing performance gains achieved. In general these mechanisms have some or all of the following drawbacks:

1) **Security**: Introduction of new protocols or using reserved headers (e.g., HyGenICC uses IP reserved bits known as "Evil-bit" [10] which was used for security testing.
2) **Overhead**: Flow tracking and feedback packets crafting on a per VM-to-VM basis adds burden to the hypervisor's processing overhead.
3) **Locality**: The lack of global information about dynamic network conditions which allows hypervisors to react only to local VM-to-VM congestion.
4) **Mutli-Path**: VM-to-VM packets are not guaranteed to take same path when multipath routing (e.g., ECMP) is in use, leading to under estimation of VM-to-VM congestion.

Recently, SDN has seen growing number of deployments for intra/inter data center networks [4, 32]. SDN was also invoked to address complex congestion events such as TCP-Incast in data centers [2, 24]. Hence, we address the aforementioned pitfalls of former schemes by taking advantage of the rich information, flexibility and global scope of SDN framework. We show that SDN-GCC is a middle-box and mutli-path friendly solution that achieves similar design goals with lesser deployment overhead and lower CPU and network overhead. This is achieved by leveraging simple rate limiters and incorporating a network-aware SDN controller towards building a dynamic adaptive system. The essence of SDN-GCC is to address the increasing trend and shift to SDN infrastructures while keeping traditional transport protocols unchanged in current production data centers.

IV. Proposed Methodology

Figure 2 shows SDN-GCC's system design which is broken down into two parts: a network application that runs on the SDN controller (network OS). It is responsible for monitoring network states by querying the switches periodically via SDN's standard southbound API and signalling congestion; and a hypervisor-based shim-layer, that is responsible of enforcing per-VM rate control in response to congestion notification by the control application. The following scenario sketches the SDN-GCC cycle: 1) Whenever the total incoming load exceeds the link capacity, the link (in-red) becomes congested implying that senders are exceeding their allocated rates. 2) SDN-switches sends to the network OS periodic keep-alive and statistics through the established control plane between them (e.g., OpenFlow or sFlow). Whenever necessary, the switch would report the amount of congestion experienced by each output queue of its ports. 3) The SDN-GCC application co-located with the network OS (or alternatively communicating via the north-bound API) tracks congestion events in the network. 4) SDN-GCC application communicates with the SDN-GCC shim-layer of the sending servers whose VMs are causing the congestion. 5) SDN-GCC shim-layer takes corrective action by adjusting the rate-limiter of the target VM.

Table I: *Variables tracked at the shim-layer and the Controller*

Variable name (Shim-Layer)	Description
$source$	IP address of source VM
$vport$	virtual port connecting VM
$rate$	The allocated sending rate
$bucket$	The capacity of the token bucket in bytes
$tokens$	The number of available tokens
$senttime$	The time-stamp of last transmission
Variable name (application)	**Description**
$SWITCH$	List of the controlled SDN switches
$SWITCHPORT$	List of the ports on the switches
$DSTSRC$	List of destinations to sources pairs
$IPTOPORT$	List of IP to switch port pairs
$MARKS$	ECN marks reading of for each switch port

We start from a single end-host (hypervisor) connecting all VMs where bandwidth contention happens at the output link (i.e., when multiple senders compete to send through the same output NIC of the virtual switch). The hypervisor needs to distribute the available NIC's capacity among VMs and ensure compliance of the VMs' weights with the allocated shares. Hence it employs a mechanism to apply rate limiters on a per-VM basis. Table I shows the variables needed to implement a per-VM token-bucket rate limiter. Ideally, when a virtual port becomes active, its variables are initialized and the NIC's nominal capacity is redistributed among the rate limiters of currently active VMs by readjusting the rate and bucket size of all active VMs' token buckets on that NIC. Then we need to extend the allocation of single hypervisor to account for the in-network congestion caused by a network of hypervisors managing tenants' VMs.

In practice, congestion may always happen within the data center network, if the network is over-subscribed or does not provide full bisection bandwidth. SDN-GCC in an effort to account for this limitation, relies on readily available functionality in SDN switches to convey congestion events to the controller. To elaborate more, SDN-GCC controller can keep a centralized record of congestion statistics by periodically collecting state information from the switches as shown in Table I. ECN marking is chosen as a fast live congestion indication to signal the onset of possible congestion at any shared queue. However, Usage of RED and ECN marking could be avoided if drop-tail AQM keeps statistics of backlog exceeding a certain pre-set threshold.

SDN-GCC application running on top of the network OS, keeps record of each network-wide state information (e.g., congestion points). Hence, it can infer the bottleneck queues based on this information and make intelligent decisions accordingly. Whenever necessary, it sends special congestion notifications to the shim-layer to adjust the sending rate of the affected VM. Upon receiving any congestion notification The shim-layer reacts by adjusting VM's rate-limiter proportionally to the congestion level in the network and gradually increases the rate when no more congestion messages are received.

V. Design and Implementation

As explained above, SDN-GCC needs two components: shim-layer at the servers and the control application that runs on top of the network OS. These mechanisms can either be implemented in software, or hardware or a combination of both as necessary. We simplified the design and concepts of SDN-GCC so that the built system is able to maintain line rate performance at 1-10Gb/s while reacting quickly to deal with congestion within a reasonable time.

A. SDN-GCC End-Host Shim-Layer

SDN-GCC shim-layer processing is described in Algorithm 1. The major variables it tracks are the *rate*, the number of *tokens* and the depth of the *bucket* variables per-VM per-NIC where the per-VM rate limiters are implemented as counting token buckets where virtual NIC j has a rate $R(i,j)$, bucket depth $B(i,j)$ and number of tokens $T(i,j)$ on physical NIC i. In addition, the shim-layer will also translate the received congestion message from the controller on a per-source basis.

Initially, the installed on-system NICs are probed and the values of their nominal data rate $R(i)$, and bucket size $B(i)$ are calculated. Thereafter, when the first packet is intercepted from a new VM, NIC capacity is redistributed and a equal-share of capacity "$E(i)$" is calculated. The new value $E(i)$ is used to re-distribute the allocated rate for each active VM and then the new VM is marked active[2] As shown in Table I, the state of the communicating VM is tracked only through token bucket and congestion specific variables. Shim-layer algorithm 1 is located at the forwarding stage of the stack, on arrival or departure of a packet P, it detects the packet's outgoing port j and incoming port i. Before packet departure, the available tokens $T(i,j)$ is refreshed based on the elapsed time since the last transmission. The packet is then allowed for transmission if $T(i,j) \geq size(pkt)$, in which case $size(pkt)$ is deducted from $T(i,j)$. Otherwise. the packet is simply dropped[3]. The shim-layer intercepts only the special congestion message.

For each incoming notification, the algorithm cuts the sending rate in proportion to the rate of marking received capped by R_{min} which is a parameter set by the operator.

Algorithm 1: SDN-GCC Shim-layer

```
1  Function $Normal\_Packet\_Arrival(P, src, dst)$
2      /* i is NIC and j is VNIC index    */
3      $T(i,j) = T(i,j) + R(i,j) \times (now() - f.senttime)$;
4      $T(i,j) = MIN(B(i,j), T(i,j))$;
5      if $T(i,j) \geq Size(P)$ then
6          $T(i,j) = T(i,j) - Size(P)$;
7          $senttime(i,j) = now()$;
8      else
9          Queue until token regeneration OR Drop;
10 Function $Control\_Packet\_Arrival(P, i, j)$
11     if Packet has congestion notification message then
12         $marks = int(msg)$;
13         if $marks \geq 0$ then
14             $congdetected(i,j) = true$;
15             $elapsedtime = now() - congtime(i,j)$;
16             $markrate = \frac{marks}{elpasedtime}$;
17             $R(i,j) = R(i,j) - (markrate \times scale(C))$;
18             $R(i,j) = Max(R_{min}, R(i,j))$;
19             $congtime(i,j) = now()$;
20     else
21         Send to normal packet processing;
22 Function $Rate\_Update\_Timeout()$
23     forall i in NICs and j in VNICs do
24         if $now() - senttime(i,j) \geq 1sec$ then
25             $active(i,j) = false$;
26             redistribute NIC capacity among active flows;
27     forall i in NICs and j in VNICs do
28         if $now() - congtime(i,j) \geq T_c$ then
             $congdetected(i,j) = false$ ;
29         if $congdetected(i,j) == false$ then
30             $R(i,j) = R(i,j) + scale(C)$;
31             $R(i,j) = MIN(E(i), R(i,j))$;
```

Hence, as sources cause more congestion in the network, the amount of marks received increases and as a result the sending rates of such sources decreases proportionally until the congestion subsides. When Congestion messages become less frequent or after a pre-determined timer T_c elapses, the algorithm starts to gradually increase the source VMs' rate conservatively. The rate is increased until it reaches "$E(i)$" or congestion is perceived again leading to another reduction. Function "scale(C)" is used to scale the amount of rate increase and decrease proportional to the NICs rate and to smooth out large variations in rate dynamics.

B. SDN Network Application

SDN-GCC relies on a SDN network application to probe for congestion statistics on a regular basis from the queues of the SDN switches in the network. The application sends notification messages towards the VMs that are causing congestion on

[2]Typically, after a certain time of inactivity (e.g., 1 sec in our simulation), the variables used for VM tracking are reset and the rate allocations are redistributed among currently active VMs.

[3]Packets can be queued for later transmission, however, this approach incurs large overhead on the end-hosts

a given queue. This is accomplished by crafting a special message with those particular VMs as destinations with the data indicating the amount of marking they have caused. These messages are never delivered to the VMs and are actually intercepted by the shim-layer in the hypervisor. For simplicity, we assume that each of the involved VMs contribute equally to the congestion and hence the marks are divided equally among source VMs. SDN-GCC Controller shown in Algorithm 2 is an event-driven mechanism that handles two major events: packet arrivals of unidentified flows (miss-entries) from switches and congestion monitoring timer expiry to trigger warning messages to the involved sources if necessary.

1) **Upon a packet arrival:** extract the necessary information to establish source to destination $SDTSRC$ relationship and destination to port relationship $IPTOPORT$. This is necessary to establish associations between congested ports and corresponding sources. In addition, The timer for congestion monitoring is re-armed if it was not already.
2) **Congestion monitoring timer expiry:** for each switch sw, the controller probes for marking statistics through OpenFlow or sFlow protocols by calling function $readmarks(sw)$. The new marking rate of each switch port p is calculated. For each port, if there are new markings (due to congestion), then the controller needs to advertise this to all related sources. Thus we first retrieve the destination list of this port via function $getalldst(sw, p)$ and then for each destination retrieve the sources using $getallsrc(dst)$. The controller now piggybacks on any outgoing control message or crafts an IP message consisting of an Ethernet Header (14 bytes), an IP header (20 bytes), and a payload (2-byte) containing the number of ECN marks that have been observed in the last measurement period, divided by the number of sources. This message is created for each source concerned (sending through the port p experiencing congestion) and sent with the source IP of the dest. VM and dest. IP of the source VM (which allows the shim-layer to identify the apporporiate forwarding ports of source VM).

Algorithm 2: SDN-GCC SDN application

```
1  Function $Packet\_Arrival(P, src, dst)$
2      if $IPpacket$ then
3          $\beta \leftarrow \beta + 1$;
4          $SDTSRC[P.src] = P.dst$;
5          $IPTOPORT[P.src] = P.in\_port$;
6          if Timer is not active then start $CM\_Timer(T_i)$ ;
7  Function $Congestion\_Monitor\_Timeout()$
8      forall $sw$ in $SWITCH$ do
9          $sw\_marks = readmarks(sw)$;
10         forall $p$ in $SWITCH\_PORT$ do
11             $\alpha = MARKS[sw][p] - sw\_marks[p]$;
12             $MARKS[sw][p] = MARKS[sw][p] + \alpha$;
13             if $\alpha > 0$ then
14                 $DSTLIST = getalldst(sw, p)$ ;
15                 forall $dst$ in $DSTLIST$ do
16                     $SRCLIST[dst] = getallsrc(dst)$;
17                     $\beta = \beta + size(SRCLIST[dst])$;
18                 if $totalsrc > 0$ then
19                     $m = \frac{\alpha}{\beta}$;
20                     forall $dst$ in $DSTLIST$ do
21                         forall $src$ in $SRCLIST[dst]$ do
22                             $msg = MSG ( m , dst , src )$;
23                             send msg to src;
24     Restart $CM\_Timer(T_i)$;
```

C. Implementation and Practical Issues

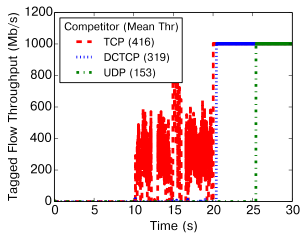
Any traffic sent by the VM in excess of its share can either be queued or simply dropped and resent later by the transport layer. In the former case, an extra per-VM queue is used for holding the traffic for later transmission whenever the tokens are regenerated. We tested both approaches and the queuing mechanism turned out to achieve marginally better performance which did not motivate its need in view of the complexity it adds. If ECN marking is not in use end-to-end, all outgoing data packets must be marked with the ECT bit. In addition, the shim-layer needs to clear any ECN marking used to track congestion before delivering the packets to the target VMs.

SDN-GCC is a distributed mechanism among the network application and the shim-layer with very low computational complexity and can be integrated easily in any network whose infrastructure is based on SDN. Due to recent advancement of memory speeds, the throughput of internal forwarding (e.g., OpenvSwitch (OvS)) of commercial desktop/server is 50-100 Gb/s, which is fast enough to handle 10's of concurrent VMs sharing a single or few physical links. Hence, the overhead of the shim-layer functions added to the OvS would not occupy much of the CPU. In addition, the shim layer at the hypervisor requires operations of $O(1)$ per packet, as a result the additional overhead is insignificant for hypervisors running on DC-grade servers. The network application is also of low complexity making it ideal for fast response to congestion (within few milliseconds time scale).

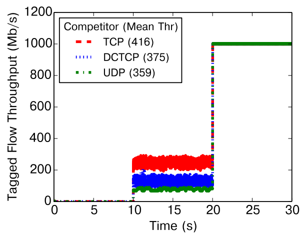
In multi-path routing, the SDN application with global view can evaluate congestion on a path-by-path basis. Consequently, the shim-layer can adapt rates to each path which it can identify via 5-tuple hash of the packets. Finally, the control plane communications in SDN networks is typically out-of-band i.e., different from the data plane [29], hence fast reaction to congestion is possible and notification messages are not interrupted by in-network middle-boxes.

VI. Simulation Analysis

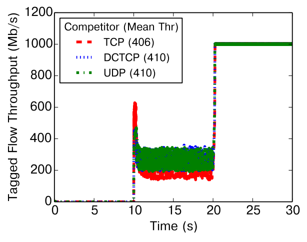
In this section, we study the performance of the proposed scheme via ns2 simulation in network scenarios with a high bandwidth-low delay. We examine the performance of a tagged VM that uses New-Reno TCP with SACK-enabled. The tagged TCP connection competes with other VMs running similar New-Reno TCP, DCTCP, or UDP in four cases: *1)* a setup that uses RED AQM with non-ECN enabled TCP; *2)* a setup that uses RED AQM with ECN enabled TCP; *3)* a setup that uses HyGenICC as the traffic control mechanism [23]; and *4)* a setup that uses proposed SDN-GCC framework. For



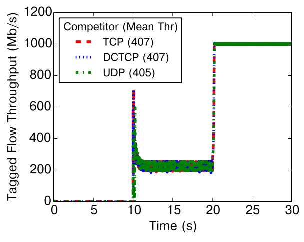
(a) *non-ECN-enabled TCP*



(b) *ECN-enabled TCP*



(c) *HyGenICC*



(d) *SDN-GCC*

Figure 3: *Goodput and mean of tagged TCP flow while competing with 3 senders using either TCP, DCTCP or UDP.*

HyGenICC, there is a single parameter settings of timeout interval for updating flow rates which should be larger than a single RTT, in the simulation this value is set to 500 μs (i.e., 5 RTTs). However, in the case of SDN-GCC, timeout interval for congestion monitoring and reporting application of the controller is set to a large value of 5ms (i.e., 50 RTTs). In all simulation experiments, we adjust RED parameters to achieve marking based on instantaneous queue length at the threshold of 20% of the buffer size.

A. Simulation Setup

We use ns2 version 2.35 [26] which we patched to include the different mechanisms. We use in all our simulation experiments link speed of 1 Gb/s for stations, small RTT of 100 μs and the default TCP RTO_{min} of 200 ms. We use a single-rooted tree (Dumbell) topology with single bottleneck link of 1Gb/s at the destination, and run the experiments for a period of 30 sec. The buffer size of the bottleneck link is set to be more than the bandwidth-delay product in all cases (100 Packets), the IP data packet size is 1500 bytes.

B. Simulation Results and Discussion

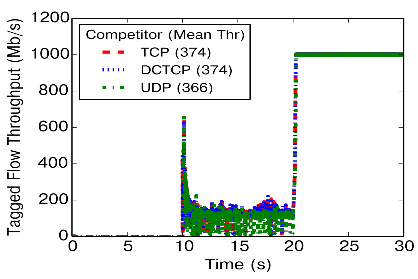
For clarity, we first consider a toy scenario with 4 elephant flows (the tagged flow and 3 competitors). In the experiments, the tagged FTP flow uses TCP NewReno and competes either with 3 FTP flows using TCP newReno, DCTCP or 3 CBR flows using UDP. Competitors start and finish at the 0^{th} and 20^{th}sec respectively, while the tagged flow starts at the 10^{th}sec and continues until the end. Hence, from 0 to 10s (period 1) only the competitors occupy the bandwidth, from 10s to 20s (period 2) the bandwidth is shared by all flows, and from 20s to 30s (period 3) the tagged flow uses the whole bandwidth. This experiment demonstrates work conservation, sharing fairness, and convergence speed of SDN-GCC compared to other setups.

Figure 3 shows the instantaneous goodput of the tagged TCP flow along with the mean goodput with respect to its competitor (in the legend, the optimal average goodput of tagged TCP would be 0Mb/s for period 1, 250Mb/s for period 2, 1000Mb/s for period 3 and 416Mb/s for all the periods). As shown in Figure 3a, without any explicit rate allocation mechanisms and without ECN ability, TCP struggles to grab any bandwidth when competing with DCTCP and UDP flows as DCTCP and UDP are more aggressive. Figure 3b suggests that ECN can partially ease the problem, however the achieved throughput reaches the allocated share only when the competitor uses the same TCP protocol. This can be attributed to the fact that TCP reacts conservatively to ECN marks unlike DCTPC which reacts proportionally to the fraction of ECN marks. Simulation with a static rate limit of 250 Mb/s (fair-share), show that a central rate allocator assigning rates per VM can achieve perfect rate allocation with no work-conservation (Utilization is 250 Mb/s in period 3). Figures 3c shows that HyGenICC [23] thanks to its distributed and live adaptive rate limiters, can respond effectively to congestion events. Finally, Figures 3d suggest a similar result as HyGenICC can be achieved with the help of a regular control messaging from a central controller whenever necessary. Hence, SDN-GCC can efficiently leverage its global view of network status to dynamically adjust the rate limiters controlling the competing flows that cause congestion and yet achieve work conservative high network utilization.

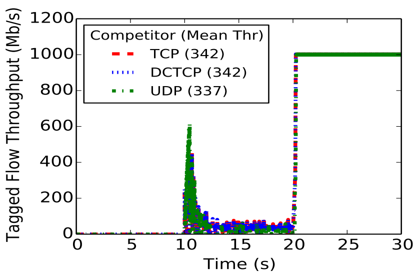
SDN based schemes are questioned for their scalability which is currently under active research [21]. Figures 4a , 4b and 4c suggests that SDN-GCC can scale well with an increasing number of senders. The tagged TCP flow and competing flows, starting at 10^{th}, adjust their rates due to the incoming control messages when the controller starts observing congestion in the network. The adjustment messages trigger flow rate changes up and down until they reach the equilibrium point where sources start oscillating slightly around the target share of $\approx \frac{1Gb}{8} \approx 125Mb$, $\approx \frac{1Gb}{16} \approx 62.5Mb$ and $\approx \frac{1Gb}{32} \approx 31.25Mb$ respectively. In SDN environments, controller delays are of major concern. To study the effect of controller delay, Figures 4d, 4e and 4f shows the same 4 senders scenario but with smaller control delay of 10 RTTs and larger delay of 100 RTT and 500 RTT. Fig 4d shows that to achieve faster convergence smaller switch-controller-hypervisor delay is always preferable. Figs 4e and 4f shows that flows oscillations and convergence period increases as the controller delay increases to 10ms and 50ms. This behavior is expected.

VII. Testbed implementation of SDN-GCC

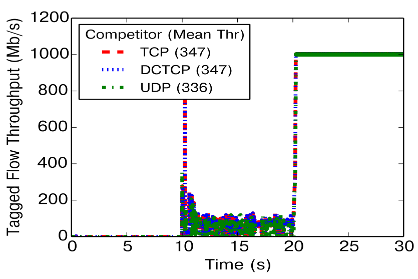
We implemented SDN-GCC Control application as a separate application program in python for any python-based controller (e.g., Ryu [31] SDN framework in our testbed). Since, OpenStack along with other popular cloud and virtualization management software use OpenvSwitch [28] as their end-host (hypervisor) networking layer. We implemented SDN-GCC shim-layer as a patch to the Kernel datapath module of OvS. We added the token-bucket rate limiters and the congestion message handler (i.e., the shim-layer) in the packet processing pipeline in the datapath of OvS. In a virtualized environment, OvS forwards the traffic for inter-VM, Intra-Host and Inter-Host



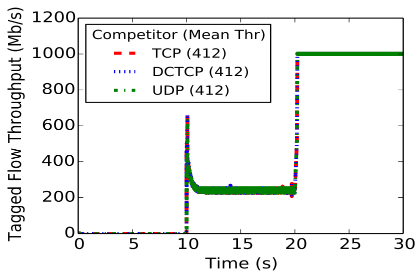
(a) *8 senders scenario*



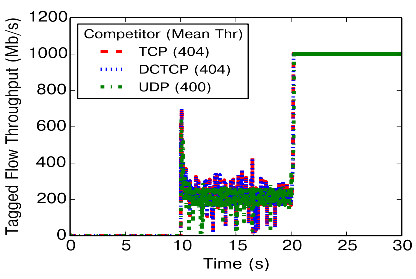
(b) *16 senders scenario*



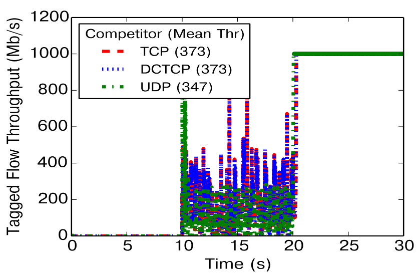
(c) *32 senders scenario*



(d) *1ms monitor interval*

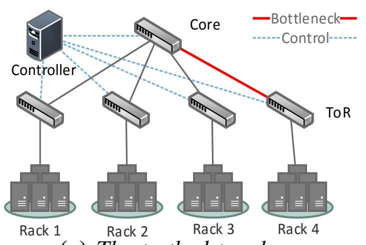


(e) *10ms monitor interval*



(f) *50ms monitor interval*

Figure 4: *Goodput and mean of tagged TCP flow while: (a-c) competing with 7, 15 and 31 senders using either TCP, DCTCP or UDP. (d-f) competing with 4 flows but a control period of 1ms (10RTT), 10ms (100RTT) or 50ms (500RTT) is used.*



(a) *The testbed topology*

(b) *The actual testbed*

Figure 5: *A real testbed for experimenting with SDN-GCC framework*

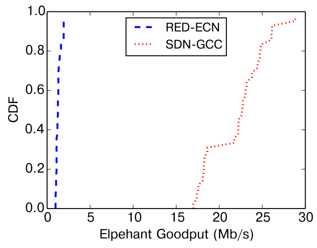
communications. This leads to an easy and straightforward way of deploying the shim-layer at the end-hosts by only applying a patch and recompiling the OvS kernel module, introducing minimal impact on the operations of production DC networks with no need for a complete shutdown. Specifically, deployment can be carried out by the management software responsible for admission and monitoring of the data center.

We set up a testbed as shown in Fig. 5. All machines' internal and the outgoing physical ports are connected to the patched OvS. We have 4 virtual racks of 7 servers each (rack 1, 2 and 3 are senders and rack 4 is receiver) all servers are installed with Ubuntu Server 14.04 LTS running kernel version (3.16) and are connected to the ToR switch through 1 Gb/s links. Similarly, the machines are installed with the iperf [16] program for creating elephant flows and the Apache web server hosting a single webpage **"index.html"** of size 11.5KB for creating mice flows. We setup different scenarios to reproduce both incast and buffer-bloating situations with bottleneck link in the network as shown in Fig. 5. Various iperf and/or Apache client/server processes are created and associated with their own virtual ports on the OvS at the end-hosts. This allows for scenarios with large number of flows in the network to emulate a data center with various co-existing applications. In experiments, we set the controller monitoring interval to a conservative value of $300ms$ whereas the network RTT ranges from $\approx 300\mu s$ without queuing and up to $\approx 300\mu s - 2ms$ with in-network queuing.

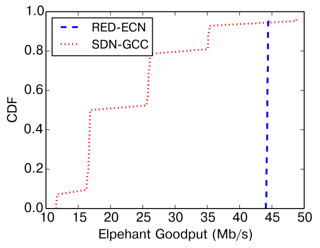
We run a scenario in which TCP and UDP elephant flows are competing for bandwidth and to test the agility of SDN-GCC, a burst of mice TCP flows is introduced to compete for bandwidth in a short-period of time. We first generate 7 synchronized TCP iperf flows and another 7 UDP iperf flows from each sending rack for 20 secs resulting in 42 ($2 \times 7 \times 3 = 42$) elephants at the bottleneck. At the 10^{th}sec, we use Apache Benchmark [8] to request **"index.html"** webpage (10 times) from each of the 7 web servers on each sending rack ($7 \times 6 \times 3 = 126$ in total). Figs. 6a and 6b show that the TCP elephants are able to grab their share of bandwidth regardless of the existence of non-well-behaved UDP traffic. In addition, Fig. 6c and 6d suggests that mice flows still benefit from SDN-GCC by achieving a smaller and nearly smooth (equal) flow completion time on average with a smaller standard deviation demonstrating SDN-GCC's effectiveness in apportioning the link capacity. In summary, SDN-GCC effectively tackles congestion and allocates the capacity among various flow types as expected.

VIII. Conclusion and Future Work

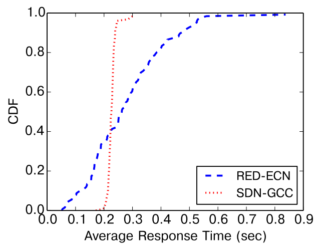
In this paper, we set to build a system that relies of the pervasive availability of SDN capable switches in datacenters to provide a centralized congestion control mechanism with a small deployment overhead onto production data centers. Our system achieves better bandwidth isolation and improved application performance. SDN-GCC is a SDN framework that can enforce efficient network bandwidth allocation among



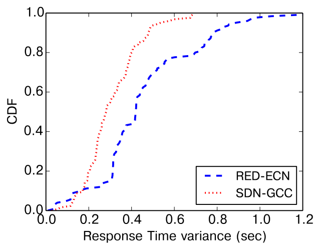
(a) *AVG elephant TCP goodput*



(b) *AVG elephant UDP goodput*



(c) *AVG FCT for mice*



(d) *SD of FCT for mice*

Figure 6: *Testbed experiment involving 126 TCP mice competing against 21 TCP (same variant) and 21 UDP elephants*

competing VMs by employing simple building blocks such as rate limiters at the hypervisors along with an efficient SDN application. SDN-GCC is designed to operate with low overhead, on commodity hardware, and with no assumption of tenant's cooperation which makes a great composition for the deployment in SDN-based data center networks. SDN-GCC was shown via simulation and deployment that it can efficiently divide network bandwidth across active VMs by enforcing the target rates regardless of transport protocol in use. Further testing of SDN-GCC using realistic workloads in our testbed is currently part of our ongoing work.

References

[1] A. M. Abdelmoniem and B. Bensaou, "Efficient switch-assisted congestion control for data centers: an implementation and evaluation," in *Proceedings of IEEE IPCCC*, Dec. 2015.

[2] A. M. Abdelmoniem and B. Bensaou, "Incast-Aware Switch-Assisted TCP congestion control for data centers," in *Proceedings of IEEE GlobeCom*, 2015.

[3] A. M. Abdelmoniem and B. Bensaou, "Reconciling mice and elephants in data center networks," in *IEEE CloudNet*, 2015.

[4] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 40, p. 63, 2010.

[7] Amazon, "AWS Virtual Private Cloud (VPC)," http://aws.amazon.com/vpc/.

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

[9] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," *ACM CCR*, vol. 41, 2011.

[10] C. Bellovin, "The security flag in the ipv4 header," 2003, https://www.ietf.org/rfc/rfc3514.txt.

[11] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "Cloudnaas: A cloud networking platform for enterprise applications," in *Proceedings of ACM Symposium on Cloud Computing*, 2011.

[12] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive, "A flexible model for resource management in virtual private networks," in *Proceedings of ACM SIGCOMM*, 1999.

[13] B. A. Greenberg, J. R. Hamilton, S. Kandula, C. Kim, P. Lahiri, A. Maltz, P. Patel, S. Sengupta, A. Greenberg, N. Jain, and D. A. Maltz, "VL2: a scalable and flexible data center network," in *Proceedings of ACM SIGCOMM*, 2009.

[14] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *6th CoNext Conference*, 2010.

[15] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," in *Proceedings of SIGCOMM*, 2013.

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

[17] S. M. Irteza, A. Ahmed, S. Farrukh, B. N. Memon, and I. A. Qazi, "On the coexistence of transport protocols in data centers," in *Proceedings of IEEE ICC*, 2014.

[18] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM*, 2013.

[19] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: Practical network performance isolation at the edge," in *10th USENIX NSDI Conference*, 2013.

[20] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *Proceedings of 12th NSDI*, 2015.

[21] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279 – 293, 2017.

[22] Y. Lu and S. Zhu, "SDN-based TCP Congestion Control in Data Center Networks," in *Proceedings of IEEE IPCCC*, 2015.

[23] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "HyGenICC: hypervisor-based generic IP congestion control for virtualized data centers," in *Proceedings of IEEE ICC*, 2016.

[24] A. M. Abdelmoniem, B. Bensaou, and A. J. Abu, "SICC: sdn-based incast congestion control for data centers," in *Proceedings of IEEE ICC*, 2017.

[25] R. Nishtala, H. Fugal, and S. Grimm, "Scaling memcache at facebook," *Proceedings of 10th USENIX NSDI*, 2013.

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

[27] Open Networking Foundation, "SDN Architecture Overview," Open Networking Foundation, Tech. Rep., Dec 2013.

[28] OpenvSwitch, "Open Virtual Switch project," http://openvswitch.org/.

[29] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *ACM HotSDN workshop*, 2013.

[30] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath TCP," in *Proceedings of ACM SIGCOMM*, 2011.

[31] Ryu Framework Community, "Ryu: a component-based software defined networking controller," http://osrg.github.io/ryu/.

[32] Scott Raynovich, "A Look at Key SDN Deployments," https://www.sdxcentral.com/articles/analysis/key-sdn-deployments/2016/06/.

[33] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the data center network," in *NSDI*, 2011.

[34] 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, 2013.