SDN-based Generic Congestion Control Mechanism for Data Centers: 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

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 congestion control mechanism that puts monitoring and control decisions in a centralized controller and traffic control enforcement in the servers' hypervisors. SDN-GGC 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. SDN-GCC is evaluated via extensive simulation in the ns2 network simulator.

1 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, however, tenants can not assume predictability nor measure-ability of bounds on network performance, as no mechanisms are deployed to explicitly allocate and enforce bandwidth in the cloud. Albeit, 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)" [3] 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) [20] 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 [14] and Microsoft [11] have deployed fully operational SDN-based WAN networks to support standard routing protocols as well as centralized Traffic Engineering (TE).

On the other hand, the data plane in intra-datacenter networks has seen little progress in aportionning 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 [10, 5], 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 [8] which abstracts the collective VMs of one tenant as if they were 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, regardless of co-existing VMs' traffic patterns or the nature of the workload generated by competing VMs.

The following are the necessary elements that can be incorporated together for this purpose: *i)* 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 as much as possible; *ii)* a methodology to fully utilize the available high bisection bandwidth (e.g., a load balancing mechanism and/or multi-path transport/routing protocols); and *iii)* 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 has investigated more or less suc-

cessfully the first two elements of this framework [1, 9, 6, 23]. In [1, 9], 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 machines with sufficient bandwidth to support the VM. Efficient routing and transport protocols [6, 23] were designed for DCN to achieve a high utilization of the available capacity. Finally, in terms of traffic control, much of recent work [2, 25] focused on restructuring TCP congestion control and its variations to efficiently utilize and fairly share bandwidth among flows of the same variant. However, both lack true isolation among tenants as a tenant may gain more bandwidth by opening parallel connections. Even worse, in mutli-tenant environments, various un-friendly transport protocols co-existence leads to starvation.

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 2, then discuss our proposed methodology and present SDN-GCC framework in Section 3. We show via ns2 simulation how SDN-GCC achieves its requirements with high efficiency in Section 5. Finally, we conclude the paper in Section 9.

2 Transport Isolation Problem

With the recent introduction of a significant number of new transport protocols designed mainly for DC networks in addition to old protocols that are still in use by most applications, the following challenges have surfaced: *i)* most of these protocols are agnostic to the nature of VM aggregate traffic demands leading to inefficient distribution of the capacity across competing VMs (for instance a VM could gain more throughput 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). This exacerbates the bandwidth inefficiency and unfairness; and, *iii)* many DC applications rely on UDP to build custom transport protocols (e.g., [18]), that are not responsive to congestion signals. This sounds the knell of any solution to the problem that only relies on end-to-end TCP. While such problems have been revealed in the context of Internet two decades ago, recent studies [13, 16] have confirmed that such problems of unfairness and bandwidth inefficiency also exist in DCNs despite their characteric small delays, small buffers and different topologies from those found in the Internet. Consequently, (because of such characteristics) a new solution to

the problems of congestion in DC networks is needed, and it must appeal to both cloud operators and cloud tenants. In [17] we proposed 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 the hypervisors to enable network bandwidth partitioning through dynamic (adaptive) rate limiters. In spite of the appealing performance gains achieved, the mechanism has shown a few drawbacks: 1. **Security**: as it uses the unused IP reserved bit known as the "Evil-bit", it may raise security concerns and may not be middleboxes-friendly; 2. **Overhead**: the hypervisors need to maintain flow tracking tables on a per VM-to-VM basis, which adds burden to the hypervisor's processing overhead; and 3. **Locality**: the lack of global knowledge of the network condition forces hypervisors to react only to local VM-to-VM congestion.

In this work, we see an opportunity to invoke the powerful control features and the global scope provided by SDN to revisit the problem from a totally 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 in use; R3) requires no changes to the tenant's OS and makes no assumption of any advanced network hardware capability other than those available in commodity SDN switches; R4) minimal overhead on end-host's hypervisor.

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 datacenter delays, it is preferable to adopt centralized control as it provides a global view of congestion and is known to achieve far better performance. 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-host hypervisors.

3 Proposed Methodology

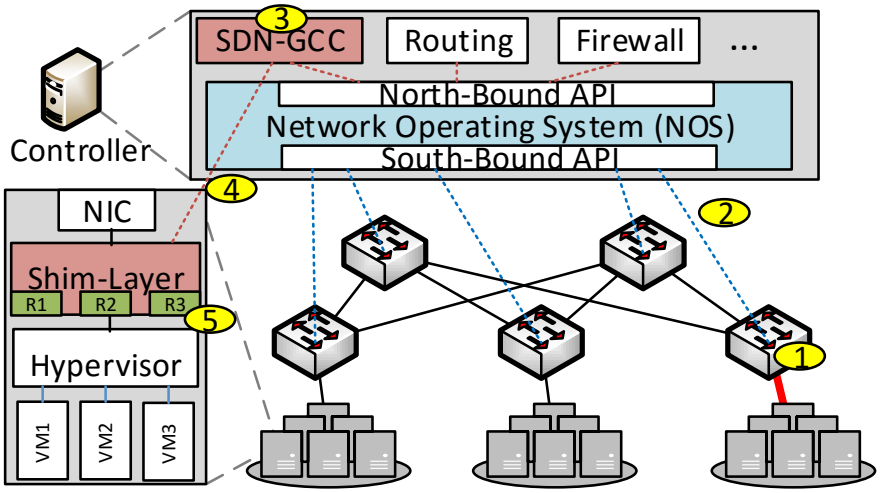


Figure 1: *SDN-GCC high-level system design: 1) congestion point; 2) network statistics; 3) congestion tracking; 4) congestion notification; 5) rate adjustment.*

Figure 1 shows SDN-GCC's system design which is broken down into two parts, a network application that runs on the SDN controller (network OS), responsible for monitoring network state by querying the switches periodically via SDN's standard southbound API and signalling congestion; and a hypervisor-based shim-layer, responsible of enforcing per-VM rate control in response to congestion notification by the control applications. 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 (i.e, 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 APP co-located with the network OS (or alternatively communicating via the north-bound API) tracks congestion events in the network. 4) SDN-GCC APP 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.

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 1 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

Table 1: *Variables tracked at the shim-layer and the Controller*

Variable name (per VM)	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 (Controller)	**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

always happen within the datacenter 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 1. 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 APP running on top of the network OS, keeps record of each network-wide state information (i.e, congestion points). Hence, it can infer the

bottleneck queues based on this information and make intelligent decisions accordingly. Whenever necessary, it sends 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 after a while when no more congestion messages are received.

4 Design and Implementation

As explained above, SDN-GCC needs two components: shim-layer at the servers and the control APP 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.

4.1 SDN-GCC 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 capacity $B(i, j)$ 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 $R(i)$, bucket size $B(i)$ are calculated. Thereafter, when the first packet is intercepted from a new VM, NIC capacity is redistributed and a new capacity share "$Capacity\_Share$" is calculated. The new value is used to update the entries for each active VM on that NIC in terms of allocated rate $R(i, j)$ and then the new VM is marked as currently active on that NIC.

After a certain time of inactivity (set to 1 sec in our simulation), the variables used for VM tracking are reset and the allocation is reclaimed to be redistributed among currently active VMs. As shown in Table 1, the state of the communicating VM is tracked only through token bucket and congestion specific variables. Shim-layer algorithm 1 being 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.

For departing packets, the current value of available tokens $T(i,j)$ is retrieved and refreshed based on the elapsed time since the last transmission. Then, using the new $T(i,j)$ value, the packet is allowed for transmission if $T(i,j) \geq size(pkt)$, in this case the packet length is deducted from $T(i,j)$, otherwise the packet is dropped. For arriving packets, it is only intercepted if it is the special congestion message which is used for rate limiter adjustment.

For each incoming notification, the algorithm cuts the sending rate in proportion to the rate of marking received capped by the Min_Rate set by the operator. Hence, as sources cause more congestion in the network, the mark amount received increases and as a result their sending rates decreases proportionally until the congestion subsides. When Congestion messages becomes less frequent or after a pre-determined timer $Congestion\_Timeout$ elapses, the algorithm starts to gradually increase the source VMs' rate conservatively. The rate is increased until it reaches its "Capacity_Share" or congestion is notified again leading to another reduction. Function "scale(NIC_Cap)" is used to scale the amount of rate increase and decrease proportional to the current rate and to smooth out large variations in rate dynamics.

4.2 SDN-GCC APP

SDN-GCC APP needs to probe for congestion statistics on a regular basis from the queues of the SDN switches in the network and send a notification messages toward the concerned VMs that are creating congestion on a given queue. This is accomplished by crafting a special message to VMs with the amount of marking they have caused. For simplicity, we assume that each of the involved VMs contribute equally to the congestion and hence the marks amount is divided equally among source VMs. Noticeably, operations of SDN-GCC APP is quite simple and does not incur much processing overhead onto the central controller. The following Algorithm 2 describes the APP in detail.

SDN-GCC Controller shown in Algorithm 2 is an event-driven mechanism which implements two major event handlers: packet arrivals of unidentified flows (miss-entries) from switches and congestion monitor timer expiry to trigger warning messages to the involved sources if necessary.

1. **Upon a packet arrival:** The packet is examined to extract the necessary information to establish source to destination $SDTSRC$ relationship and destination to port relationship $IPTOPORT$. This is necessary to establish associativity between congested ports and corresponding sources. In

addition, The timer for congestion monitoring is armed if it is not currently.

2. **Congestion monitor timer expiry:** For each switch sw, the controller probes for marking statistics through OpenFlow or sFlow protocols by calling function $readmarks(sw)$ and then the new marks 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. The destination list of this port is retrieved using function $getalldst(sw, p)$ and then for each destination its sources are retrieved using function $getallsrc(dst)$. 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 destination VM and destination IP of the source VM (which allows the hypervisor shim-layer to identify the correct forwarding ports of source VM).

4.3 Implementation and Practical Issues

Any extra 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 usage.

If ECN marking is in use, the shim-layer needs to clear any ECN marking used to track congestion before delivering the packets to target VMs. In addition, to force universal marking along the path, all outgoing packets are marked with the ECN-enabled bit.

SDN-GCC is a distributed mechanism among the control APP and the shim-layer with very low computational complexity and can be integrated easily in any network whose infrastructure is based on SDN. 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. Finally, the control APP's low complexity makes it ideal for fast response to congestion (within few milliseconds time scale).

5 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 [17]; and *4)* a setup that uses proposed SDN-GCC framework. For 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 case of SDN-GCC, timeout interval for congestion monitoring and reporting APP 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.

5.1 Simulation Setup

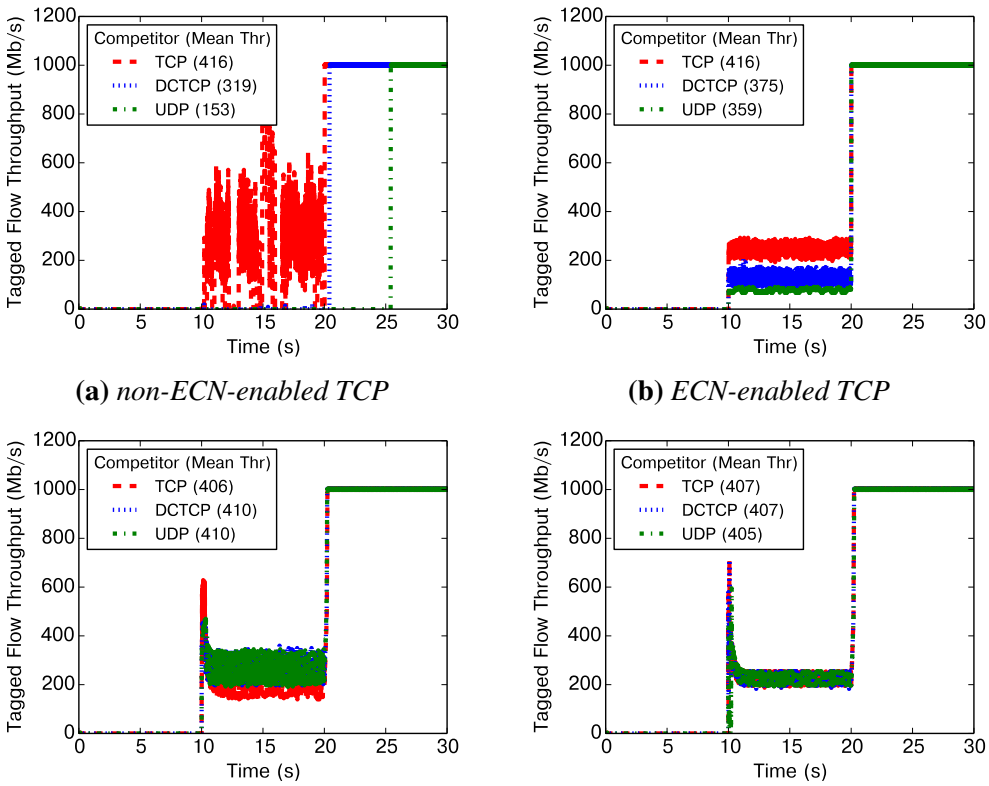
Network simulator ns2 version 2.35 [19] is used, which we have extended with both HyGenICC module inserted at the link elements in topology setup and the whole SDN-GCC framework (i.e, the controller element and hypervisor shim-layer)[1]. In addition, we patched ns2 using the publicly available DCTCP patch. We use in all our simulation experiments speed links of 1 Gb/s for sending stations, a bottleneck link of 1 Gb/s, low RTT of 100 μs and the default TCP RTO_{min} of 200 ms.

We use a single-rooted tree (Dumbell) topology with single bottleneck 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.

5.2 Simulation Results and Discussion

We simulated several scenarios that lead all to the same results. First, for purpose of clarity, we consider a toy scenario with 4 elephant flows, a tagged flow and 3 competitors. In the experiments, the tagged FTP flow uses TCP NewReno and

[1]Simulation code is available upon request from the authors.



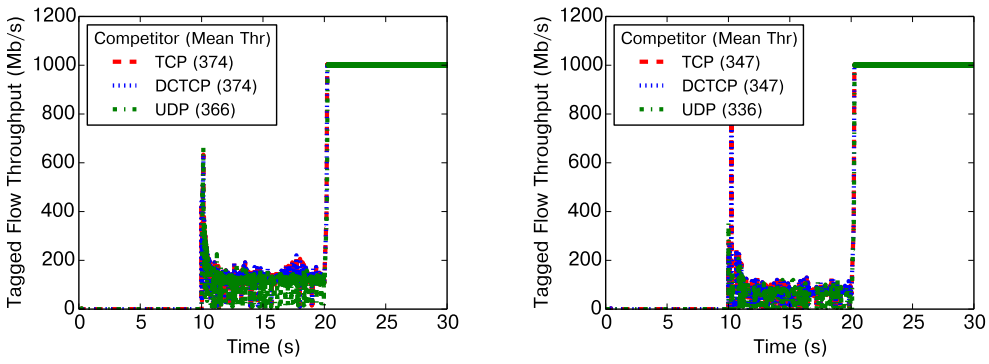
(a) *non-ECN-enabled TCP* **(b)** *ECN-enabled TCP*

(c) *HyGenICC* **(d)** *SDN-GCC*

Figure 2: *Goodput and mean of tagged TCP flow while competing with 3 senders using either TCP, DCTCP or UDP.*

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 tagged flow starts at 10^{th}sec until the end of simulation. Hence, from 0 to 10s (period 1) only the competitors occupy the bandwidth, from 10s to 20s (period 2) bandwidth should typically be shared fairly by all flows, and from 20s to 30s (period 3) the tagged flow enjoys the whole bandwidth. This experiment is designed to demonstrate work conservation, sharing efficiency, and convergence speed of SDN-GCC compared to other setups.

Figure 2 shows the instantaneous goodput of the tagged TCP flow along with the mean goodput (in the legend, the optimal mean 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) with respect to its competitor. As shown in Figure 2a, without

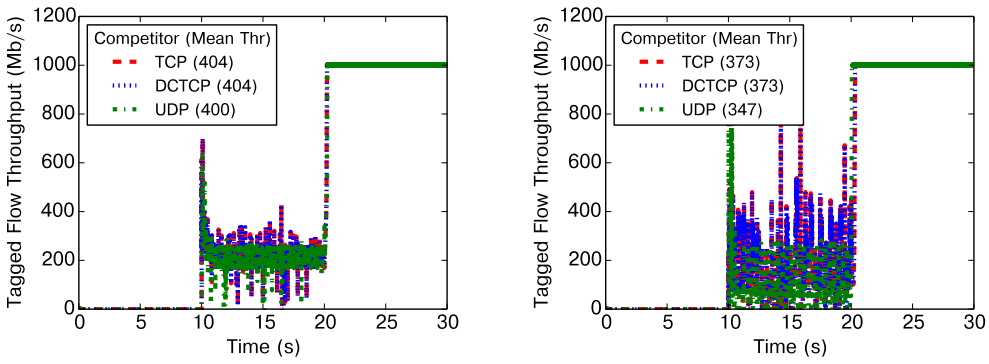


(a) *8 senders scenario* **(b)** *16 senders scenario*

Figure 3: *Goodput and mean of tagged TCP flow competing with 7 and 15 senders using either TCP, DCTCP or UDP.*

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 in grabbing bandwidth. Figure 2b 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 the ECN marks. Simulations 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 when only the tagged flow is active). Figures 2c shows that HyGenICC [17] thanks to its distributed and live adaptive rate limiters, can respond effectively to congestion events. Finally, Figures 2d 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 state to adjust the rate limiters of the competing flows that cause congestion while maintaining high network utilization.

Figure 3 suggests that SDN-GCC can scale well with an increasing number of senders. The tagged TCP flow and competing flows, starting at 10^{th}, adjusts 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 $\approx \frac{1Gb}{8} \approx 125Mb$ and $\approx \frac{1Gb}{16} \approx 62.5Mb$ respectively. To study the effect of controller delay, Figure 4 shows the same 4 senders



(a) *10ms monitor interval* **(b)** *50ms monitor interval*

Figure 4: *Same as scenario in Fig 2 but 10ms (100RTT) and 50ms (500RTT) control period are used.*

scenario but with larger control delay of 100 RTT and 500 RTT. It suggests that flows oscillations and convergence period increases as the controller delay increases to values of 10ms and 50ms, respectively. This behavior is expected due to slower control message arrivals leading to slower reaction at the shim-layer and lesser rate adjustments. An out of band control path [22] would avoid any added overhead and achieve faster controller monitoring of switches and reporting to the servers.

6 Implementation Details of SDN-GCC framework

Below are the main procedures that constitute the SDN-GCC OpenvSwitch implementation. We show here the find_active function which given the in and out ports, it returns the corresponding index of these ports in the matrices. Also, we present the update_rates which updates each VMs rate based on the current level of congestion in the network[2]. We show also the timer_callback function which is our rate update daemon that fires every interval. In addition, we show the sdngcc_send and sdngcc_receive functions which are responsible for handling the arrival and departure events of the packets in the OpenvSwitch, they are injected into the packet processing pipeline of the OpenvSwitch kernel datapath module.

[2]We use a different mechanism to react to congestion rather than the AIAD algorithm described in section **??**. We use the average of the moving average of the fraction of marked packets in each interval

```
//Finding the fair rate among active VM
void find_active(void)
{
  for(i=0; i<ethdevcount; i++)
  {
    unsigned int avg_sent=0,active_count=0;
    fair_rate[i]=eth_rate[i];
    for(j=0; j<virtdevcount; j++)
    {
      if(lastsent[i * DEV_MAX + j]>0 && now - lastsent[i * DEV_MAX + j] >= 1000)
      {
        isactive[i * DEV_MAX + j] = false;
        reset_hygenicc_dev(j);
      }
      else
      {
        isactive[i * DEV_MAX + j] = true;
        active_count++;
      }
    }
  }
  if(active_count>1)
   fair_rate[i] = eth_rate[i] / active_count;
}
```

Listing 1: *Find Active VMs*

```
//Refill Tokens Procedure
void update_rates(void)
{
    int i,j,k;
  //Find the active VMs
  findactive();
    for(i=0; i<ethdevcount; i++)
    {
     for(j=0; j<virtdevcount; j++)
        if(isactive[i * DEV_MAX + j])
       {
           bucket[i * DEV_MAX + j] =  MAX(MIN_BUCKET, depth * (fair_rate[i] >> 3) * interval);
          unsigned int lostrate = (fair_rate[i] * avgalpha[j]) >> 10;
```

```
14          rate[i * DEV_MAX + j] = MAX(MIN_RATE, fair_rate[i] - ↩
                lostrate);
15        }
16    }
17 }
```

Listing 2: *SDN-GCC Update Rates Procedure*

```
1  //Rate Update Daemon
2  hrtimer_restart timer_callback(hrtimer *timer)
3  {
4       int i,j;
5    long int now=jiffies;
6      if(sdngcc_enabled() && sdngcc_timerrun)
7      {
8       //calculate average alpha value of each VM
9         for(i=0; i<ethdevcount; i++)
10        for(j=0; j<virtdevcount; j++)
11      if(isactive[i * DEV_MAX + j])
12           avgalpha[j]=track_get_avg_alpha(virtipaddress[j]);
13      else
14       avgalpha[j]=0;
15
16     //Update rates based on the new average alpha
17     update_rates();
18
19     //Restart the timer for a new interval
20         ktime_t ktnow = hrtimer_cb_get_time(&sdngcc_hrtimer);
21         sdngcc_ktime = ktime_set(0 , interval * ((unsigned long↩
               ) 1E3L) );
22         hrtimer_forward(&sdngcc_hrtimer, ktnow, sdngcc_ktime);
23         return HRTIMER_RESTART;
24     }
25     else
26         sdngcc_timerrun=false;
27     return HRTIMER_NORESTART;
28 }
```

Listing 3: *SDN-GCC Rate Update Daemon*

```
1  //Sender Rate Limiter Handler
2  void sdngcc_send(sk_buff *skb, vport *inp ,vport *outp,↩
       sw_flow_key *key)
```

```
{
  int i,j;
  identify_devices(inp,outp,&i,&j);

  //Find the flow entry that match with this packet
  track_key_extract(skb, &track_key);
   flow = ovs_track_tbl_lookup(&track_key, sizeof(track_key));

  if(i>=0 && j>=0)
   {
    virt_isactive[i * DEV_MAX + j]=true;
    lastsent[i * DEV_MAX + j]=jiffies;
    sent[i * DEV_MAX + j]++;

    //Add new tokens before sending the packet based on the ↩
       based interval since last sent
    ktime_t now_ktime=ktime_get();
        int delta_us = ktime_us_delta(now_ktime, lastupdate[i *↩
            DEV_MAX + j]);
    int newtokens = (rate[i * DEV_MAX + j] >> 3) * delta_us;
        tokens[i * DEV_MAX + j] = MIN(bucket[i * DEV_MAX + j], ↩
            tokens[i * DEV_MAX + j] + newtokes);
        lastupdate[i * DEV_MAX + j] = now_ktime;

    //If enough tokens are avaliable send the packet, otherwise↩
        drop
     if(skb->len <= tokens[i * DEV_MAX + j])
                tokens[i * DEV_MAX + j]= MAX(0, tokens[i * ↩
                   DEV_MAX + j] - skb->len);
         else
     {
             kfree_skb(skb);
             return;
         }

     if(!flow)
              flow=insert_flow(track_key);

      flow->used=jiffies;
           flow->out_byte_count+=skb->len;
           flow->out_packet_count++;
   }

   if(skb && outp)
            ovs_vport_send(outp, skb);
```

```
}
```

Listing 4: *SDN-GCC Rate Limiter Handler*

```
//Receiver Packet Handler
void sdngcc_receive(sk_buff *skb, vport *inp ,vport *outp,sw_flow_key *key)
{
  int m,n;
  identify_devices(inp,outp, &m,&n);

  //Handle Special IP feedback message
  if(ip_header->protocol == FEEDBACK_PACKET_IPPROTO)
   {
       mark_count = ntohs(ip_header->id);
       track_reverse_key_extract(skb, &track_key);
       flow = track_tbl_lookup(&track_key, sizeof(track_key));
       if(flow)
    {
           flow->mark_packet_count += mark_count;
      flow->mark_lastfeedback = jiffies;
    }
       kfree_skb(skb);
       return;
   }
}
```

Listing 5: *SDN-GCC Receiver Packet Handler*

Below are the main procedures that constitute the SDN-GCC SDN Controller implementation in Python programming language. We show here the switch_features function which at start-up reads out switch configuration when it registers with the controller for first time. Also, we present the install_miss_table which handles the rules to be inserted whenever a miss-event happens at the switches. switch_ports function reads out switch's ports names and store them in a table for accounting the congestion per port. SendIp function is the core of our controller which sends Raw IP packets to the SDN-GCC modules to inform them of the amount of marking and hence throttle back the mis-behaving flows. add_flow is the function that handles add custom flow requests other than the normal forwarding rule inserted by the table miss function. Finally, packet_in function handles all packets that switches can not handle and sent to the controller as PACKET_IN message for

consulting controller's action about it[3].

```
//Read switch features at start up Handler

def switch_features_handler(self, ev):
    """Handle switch features reply to install table miss flow entries."""
  global maindpid
        dpid = ev.msg.datapath_id
 #datapath = ev.msg.datapath
        #[self.install_table_miss(datapath, n) for n in [0, 1]]

 self.dstsrc_iptable.setdefault(dpid, {})
 self.port_marks.setdefault(dpid, {})
 self.port_totalmarks.setdefault(dpid, {})
 self.ip_to_port.setdefault(dpid, {})
 self.mac_to_port.setdefault(dpid, {})
        self.dstsrc_table.setdefault(dpid, {})
 self.port_num_to_name.setdefault(dpid, {})
 maindpid = dpid
 self.getswitchports(dpid)
```

Listing 6: *SDN-GCC switch feature*

```
#set up rules for table miss

def install_table_miss(self, datapath, table_id):
        """Create and install table miss flow entries."""
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto
        empty_match = parser.OFPMatch()
        output = parser.OFPActionOutput(ofproto.OFPP_NORMAL)
        write = parser.OFPInstructionActions(ofproto.
            OFPIT_WRITE_ACTIONS,[output])
        instructions = [write]
        flow_mod = self.create_flow_mod(datapath, 0, 0, 0,
            table_id, empty_match, instructions)
        datapath.send_msg(flow_mod)
```

Listing 7: *SDN-GCC table miss rule*

[3]SDN-GCC OpenvSwitch module and the SDN controller consists of other functions and data structures which were not shown for the brevity purposes. For the complete source code or the patch, please contact one of the authors

```
#Get port information from the switch

def getswitchports(self, dpid):
 """get port information from switch having id of dpid."""
 global mainport,searchstr
 p = system.getports(sw)
 str1 = StringIO(p.stdout.read())
 ports = np.genfromtxt(str1, usecols=(0,1), delimiter=' ', ↩
     dtype=None, unpack=False)
 for port in ports:
  #print port[0], port[1]
  self.port_num_to_name[dpid][port[0]] = port[1]
 print self.port_num_to_name[dpid]

 for port in self.port_num_to_name[dpid]:
  portname = self.port_num_to_name[dpid][port]
  str1= str1.parseandextract()
  if portname == 'p1p2':
   mainport=port
   searchstr=str1
  print str1
  p = subprocess.Popen("ssh root@switch %s" % str1, shell=True,↩
      stdout=subprocess.PIPE)
  string = StringIO(p.stdout.read())
  marks=0
  try:
    marks=int(string.getvalue())
     except ValueError:
          pass
  print port, ":", portname, " marked=", marks
```

Listing 8: *SDN-GCC switch port information*

```
#Send out markings seen by the controller
def sendip(self, dstip, srcip,  payload=0):
     """Send raw IP packet on interface."""

 #create a raw socket
 try:
     s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.↩
        IPPROTO_RAW)
 except socket.error , msg:
     print 'Socket could not be created. Error Code : ' + str(↩
```

```
        msg[0]) + ' Message ' + msg[1]
    sys.exit()

# tell kernel not to put in headers, since we are providing it, when using IPPROTO_RAW this is not necessary
# s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# now start constructing the packet
packet = '';

# ip header fields
ip_ihl = 5
ip_ver = 4
ip_tos = 0
ip_tot_len = 0  # kernel will fill the correct total length
ip_id = min(65535, payload)   #Id of this packet is the number of marks 54321
ip_frag_off = 0
ip_ttl = 255
ip_proto = 143
ip_check = 0    # kernel will fill the correct checksum
ip_saddr = socket.inet_aton ( srcip )   #Spoof the source ip address if you want to
ip_daddr = socket.inet_aton ( dstip)

ip_ihl_ver = (ip_ver << 4) + ip_ihl

# the ! in the pack format string means network order
ip_header = pack('!BBHHHBBH4s4s' , ip_ihl_ver, ip_tos, ip_tot_len, ip_id, ip_frag_off, ip_ttl, ip_proto, ip_check, ip_saddr, ip_daddr)

  # final full packet - syn packets dont have any data
packet = ip_header #+ payload
    #print datetime.now(), '-> Sent marking packet to: ', dstip, ' message : ', packet

#Send the packet finally - the port specified has no effect
s.sendto(packet, (dstip , 0 ))    # put this in a loop if you want to flood the target
s.close()
#print 'Sent marking packet to: ', dstip, ' from: ', srcip, ' with marking= ', payload

   def check_connections(self):
```

```
global pipebuffer, RTT, totalmarks, searchstr

if searchstr == ' ':
 Timer(sampleinterval, self.check_connections, ()).start()
 return
p = subprocess.Popen("ssh root@switch %s" % searchstr, shell=True, stdout=subprocess.PIPE)
string = StringIO(p.stdout.read())
currentmarks=0
try:
 currentmarks=int(string.getvalue())
   except ValueError:
        pass
marks = max(0, currentmarks - totalmarks)
#print 'checking RED at ', datetime.now(), 'marks: ', marks
if marks > 0:
     #for dpid in self.ip_to_port:
  #for dst in self.ip_to_port[dpid].keys():
  #print 'checking RED at ', datetime.now(), ' marks: ', marks, 'port: ', self.ip_to_port[dpid][dst], ':', mainport
  #if self.ip_to_port[dpid][dst] == mainport:
       dstnum = len(dstlist)
       totalsrc = 0
       for dst in dstlist:
  srclist = [src for (src, dst_) in self.dstsrc_iptable.get(maindpid).items()  if dst_ == dst]
  totalsrc = totalsrc + len(srclist)
       if totalsrc > 0:
      for dst in dstlist:
       srclist = [src for (src, dst_) in self.dstsrc_iptable.get(maindpid).items()  if dst_ == dst]
       srcnum = len(srclist)
       print datetime.now(), 'port: ', mainport , 'caused ECN -> dst: ', dst, 'src: ', srclist, ' totalmarks: ', totalmarks, 'newmarks: ', marks
   if srcnum > 0:
        amount = marks/ srcnum #totalsrc
        for src in srclist:
                self.sendip(src, dst, amount)
totalmarks = currentmarks
Timer(sampleinterval, self.check_connections, ()).start()
```

Listing 9: *SDN-GCC Marking Notification*

```
#Add flow function
def add_flow(self, datapath, in_port, dst, actions):
        ofproto = datapath.ofproto

 match = datapath.ofproto_parser.OFPMatch(
            in_port=in_port, dl_dst=haddr_to_bin(dst))

        mod = datapath.ofproto_parser.OFPFlowMod(
            datapath=datapath, match=match, cookie=0,
            command=ofproto.OFPFC_ADD, idle_timeout=0, ↩
               hard_timeout=0,
            priority=ofproto.OFP_DEFAULT_PRIORITY,
            flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
        datapath.send_msg(mod)
```

Listing 10: *SDN-GCC Add flow to switch flow table*

```
#Handle In-Packets that miss flow tables in the switches
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
 in_port = msg.in_port
 dpid = datapath.id

        pkt = packet.Packet(data=msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)

        if eth.ethertype == ether_types.ETH_TYPE_LLDP: #or eth.↩
           ethertype != ether_types.ETH_TYPE_IP or eth.↩
           ethertype != ether_types.ETH_TYPE_ARP:
            # ignore lldp packet
            return
        dst = eth.dst
        src = eth.src

 dstip=""
 srcip=""
 try:
  ip = pkt.get_protocol(ipv4.ipv4)
  arph = pkt.get_protocol(arp.arp)
  if ip is None and arph is None:
   return
```

```
26   if ip is not None:
27    dstip = ip.dst
28                          srcip = ip.src
29    if not srcip or srcip == '255.255.255.255': #or not dstip or↩
          dstip == '255.255.255.255':
30       return
31    #print ip
32
33   elif arp is not None:
34          dstip = arph.dst_ip
35                          srcip = arph.src_ip
36    if not srcip or srcip == '255.255.255.255' or not dstip or ↩
       dstip == '255.255.255.255':
37       return
38    #print arph
39
40
41   #if self.dstsrc_iptable[dpid].get(dstip, None) is None:
42   # self.dstsrc_iptable[dpid][dstip]=0
43   self.dstsrc_iptable[dpid][srcip]=dstip
44
45   #if self.ip_to_port[dpid].get(srcip, None) is None:
46   # self.ip_to_port[dpid][srcip]=0
47   self.ip_to_port[dpid][srcip] = msg.in_port
48
49   if msg.in_port == mainport  and srcip is not None:
50    if srcip not in dstlist:
51     dstlist.append(srcip)
52
53   if self.port_marks[dpid].get(msg.in_port, None) is None:
54    self.port_marks[dpid][msg.in_port]=0
55
56   if self.port_totalmarks[dpid].get(msg.in_port, None) is None:
57    self.port_totalmarks[dpid][msg.in_port]=0
58
59   #print 'srcip: ', srcip, 'srcmac: ', src, 'dstip: ', dstip, '↩
       dstmac: ', dst, 'portnum: ', msg.in_port
60
61   if self.timeron == False:
62    print 'timer started at ', datetime.now()
63    Timer(sampleinterval, self.check_connections, ()).start()
64    self.timeron=True
65
66 except msg:
67            print 'Not an IP packet Error Code : ' + str(msg↩
```

```
                [0]) + ' Message ' + msg[1]
68              return
69
70  if self.dstsrc_table[dpid].get(dst, None) is None:
71   self.dstsrc_table[dpid][dst]=0
72  self.dstsrc_table[dpid][src]=dst
73
74   # learn a mac address to avoid FLOOD next time.
75   self.mac_to_port[dpid][src] = msg.in_port
76
77
78  if self.port_num_to_name[dpid].get(msg.in_port, None) is None:
79   self.port_num_to_name[dpid][msg.in_port]=None
80
81          if dst in self.mac_to_port[dpid]:
82              out_port = self.mac_to_port[dpid][dst]
83          else:
84              out_port = ofproto.OFPP_FLOOD
85
86          actions = [datapath.ofproto_parser.OFPActionOutput(↩
                out_port)]
87
88          # install a flow to avoid packet_in next time
89          if out_port != ofproto.OFPP_FLOOD:
90              self.add_flow(datapath, in_port, dst, actions)
91
92          data = None
93          if msg.buffer_id == ofproto.OFP_NO_BUFFER:
94           data = msg.data
95
96          out = datapath.ofproto_parser.OFPPacketOut(
97              datapath=datapath, buffer_id=msg.buffer_id, in_port↩
                    =in_port,
98              actions=actions, data=data)
99          datapath.send_msg(out)
```

Listing 11: *SDN-GCC Handle In-Packet arrival*

7 Testbed implementation of SDN-GCC

We implemented SDN-GCC Control APP as a separate application program in python programming language for any python-based controllers (i.e, Ryu [7] SDN framework in our testbed). We have also patched the Kernel datapath module of

OpenvSwitch (OvS) [21] with SDN-GCC shim-layer[4]. 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 communications[5]. 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 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, to all servers in the DC network.

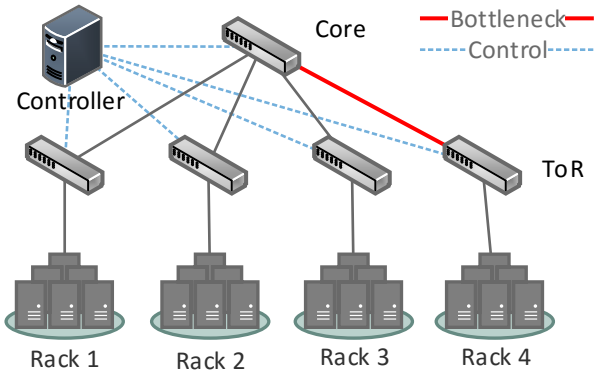


Figure 5: *A real testbed for experimenting with SDN-GCC framework*

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 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 [12] 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 situa-

[4]OpenStack along with other popular cloud and virtualization management software use OpenvSwitch as their end-host (hypervisor) networking layer

[5]Due to recent advancement of memory speeds, the throughput of internal forwarding (i.e, 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, we believe that the overhead of the shim-layer functions added to the OvS would not hog the CPU and hence the achievable throughput.

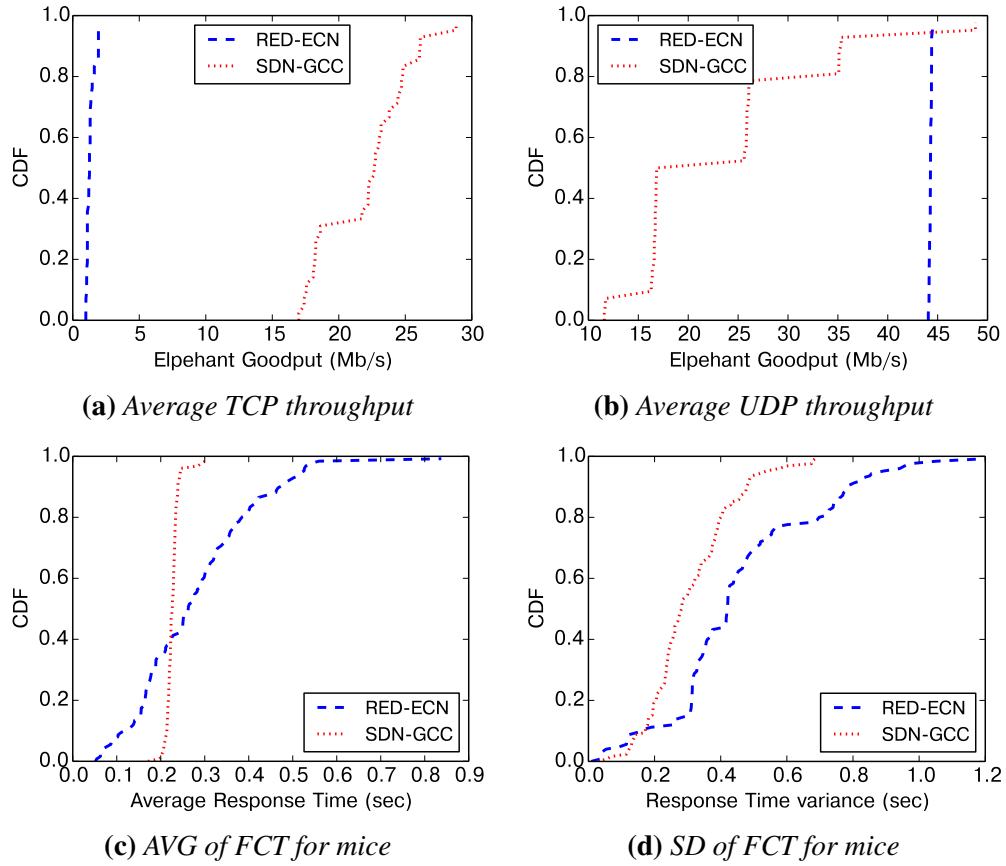
tions with bottleneck link in the network as shown in Fig. 5. Various iperf and/or Apache client/server processes are created and each is associated with its own virtual port on the OvS at the end-hosts. This allows us to create scenarios with large number of flows in the network to emulate a data center with many co-existing VMs running various applications. In the experiments we have set the controller monitoring interval to a conservative value of 300 ms whereas the network RTT ranges from 300μs without queuing and up to 1-2 ms with excessive queuing.

We run a scenario in which TCP and UDP elephant flows are competting for bandwidth in addition to a by-passing burst of mice TCP traffic which competes with them for a short-period of time to see if SDN-GCC can manage efficiently the traffic of various application. We first generate 7 synchronized TCP iperf flows and another 7 synchronized 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 [4] 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 **??** suggests that, the 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 which further demonstrates SDN-GCC's effectiveness in apportioning the bandwidth.

To summarize this simulation and experimental study, SDN-GCC seems to be able to efficiently allocate the bandwidth among various flow types and alleviate possible congestion in network core.

8 Related Work

SDN-GCC can be considered as comparable or complementary work to a number of recent proposals designed for cloud resource allocation. [24] designed a system called "Seawall" for sharing network bandwidth, which achieves per-VM max-min weighted allocations using explicit end-to-end feedback messaging for rate adaptation. Seawall requires introduction of new protocol to network stack which incurs a large processing and messaging overhead and may not be middlebox-friendly. Authors of [10] designed "Secondnet" to divide network among tenants and enforce rate limits, but is limited to static bandwidth reservation among tenants' VMs. EyeQ [15] provides per-VM max-min weighted fair shares in the context of a full bisection bandwidth datacenter topology where congestion By



(a) *Average TCP throughput*

(b) *Average UDP throughput*

(c) *AVG of FCT for mice*

(d) *SD of FCT for mice*

Figure 6: *Testbed scenario where 126 mice competing with 21 TCP and 21 UDP elephants*

leveraging simple rate limiters and incorporating network-aware SDN controller to build dynamic adaptive system, SDN-GCC is able to achieve similar design goals of these proposals in a more intuitive and less deployment, CPU, network overhead manner. The essence of SDN-GCC is to address the increasing trend and shift to SDN infrastructures while traditional transport protocols is still in use in current production datacenters.

9 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 competing VMs by employing simple building blocks such as rate limiters at the hypervisors along with an efficient SDN APP. 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 datacenter networks. SDN-GCC was shown via simulation analysis that it can efficiently divide network bandwidth across active VMs by enforcing target rates regardless of transport protocol in use. Finally, further evaluation of SDN-GCC in a large data center simulations and real testbed experiments with more complex scenarios is part of our ongoing work.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM*, 2008.

[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 CCR*, 40:63, 2010.

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

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

[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. *ACM SIGCOMM CCR*, 41(4), 2011.

[6] 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.

[7] R. S. F. Community. Ryu: a component-based software defined networking framework. http://osrg.github.io/ryu/.

[8] 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.

[9] 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.

[10] 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.

[11] 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 ACM SIGCOMM*, 2013.

[12] iperf. The TCP/UDP Bandwidth Measurement Tool. https://iperf.fr/.

[13] 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.

[14] 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 *Proceedings of ACM SIGCOMM*, 2013.

[15] 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.

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

[17] 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.

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

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

[20] Open Networking Foundation. SDN Architecture Overview. Technical report, Open Networking Foundation, Dec 2013.

[21] OpenvSwitch. Open Virtual Switch project. http://openvswitch.org/.

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

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

[24] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the 8th USENIX NSDI Conference*, 2011.

[25] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Transactions on Networking*, 21, 2013.

Algorithm 1 SDN-GCC Shim-layer Algorithm

/*i and j are index of the NIC and VNIC, respectively*/
procedure PACKET_DEPARTURE(P, i, j)
    $T(i,j) = T(i,j) + R(i,j) \times (now() - f.senttime)$
    $T(i,j) = MIN(B(i,j), T(i,j))$
    if $T(i,j) \geq Size(P)$ **then**
        $T(i,j) = T(i,j) - Size(P)$
        $senttime(i,j) = now()$
    else
        Queue until token regeneration OR Drop
procedure CONTROL_PACKET_ARRIVAL(PKT, i, j)
    if Packet has congestion notification message **then**
        $marks = int(msg)$
        if $marks \geq 0$ **then**
            $congdetected(i,j) = true$
            $elapsedtime = now() - congtime(i,j)$
            $markrate = \frac{marks}{elpasedtime}$
            $R(i,j) = R(i,j) - (markrate \times scale(NIC\_Cap))$
            $R(i,j) = Max(Min\_Rate, R(i,j))$
            $congtime(i,j) = now()$
    else
        Send to normal packet processing
procedure TIMER_TIMEOUT
    for each i in NICs and j in VNICs **do**:
        if $now() - senttime(i,j) \geq 1sec$ **then**
            $active(i,j) = false$
            redistribute NIC capacity among active flows
    for each i in NICs and j in VNICs **do**:
        if $now() - congtime(i,j) \geq Cong\_Timeout$ **then**
            $congdetected(i,j) = false$
        if $congdetected(i,j) == false$ **then**
            $R(i,j) = R(i,j) + scale(NIC\_Cap)$
            $R(i,j) = MIN(Capacity\_Share, R(i,j))$

Algorithm 2 SDN-GCC Controller Algorithm

```
1:  procedure Packet_Arrival(P, src, dst)
2:      if IPpacket then
3:          β ← β + 1
4:          SDTSRC[P.src] = P.dst
5:          IPTOPORT[P.src] = P.in_port
6:          if Timer is not ON then
7:              start Congestion_Monitor_Timer
8:  procedure Congestion_Monitor_Timeout
9:      for each sw in SWITCH do
10:         sw_marks ← readmarks(sw)
11:         for each p in SWITCH_PORT do
12:             α ← MARKS[sw][p] − sw_marks[p]
13:             MARKS[sw][p] ← MARKS[sw][p] + α
14:             if α > 0 then
15:                 DSTLIST ← getalldst(sw, p)
16:                 for each dst in DSTLIST do
17:                     SRCLIST[dst] ← getallsrc(dst)
18:                     β ← β + size(SRCLIST[dst])
19:                 if totalsrc > 0 then
20:                     m ← α/β
21:                     for each dst in DSTLIST do
22:                         for each src in SRCLIST[dst] do
23:                             msg ← MSG ( m , dst , src )
24:                             send msg to src
25:     Restart Congestion_Monitor_Timer(T_i)
```