Generic Hypervisor-based Congestion Control for Data Centers: Implementation and Evaluation
HKUST-CS15-03

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

In today's datacenters, communication resources are shared by a large number of virtual machines (VMs) deployed by untrusted and uncoordinated tenants from the public cloud. The state of the art datacenter network management systems do not employ any particularly efficient network bandwidth management or congestion control mechanisms to meet the conflicting requirements and the dynamic nature of the tenants' traffic. In this report we present HyGenICC, a simple and practical system that partitions efficiently and fairly the bandwidth among competing tenants in both oversubscribed and non-oversubscribed datacenter infrastructures. HyGenICC divides the available server bandwidth among guest VMs' traffic and enforces rate adaptation in face of imminent congestion. To be easily deployable in existing data center, by design choice, HyGenICC requires no changes to guest VMs' congestion control mechanism, no changes to switching software and relies only on simple functionalities available in today's commodity switches. HyGenICC is evaluated via both ns-2 simulations and a simple software modification to the well-known Open vSwitch in a small testbed.

1 Problem Statement

Resource sharing via virtualization has become a common practice in today's public and private datacenters. Most typically, a tenant is provisioned with virtual machines each having dedicated CPU cores or virtual CPU, dedicated memory and storage space, and virtual network interface over the underlying shared physical network interface. In many practical datacenters, the control plane is very richly provisioned with many novel methods to make the virtualization easy to manage; for example, Amazon Web Services adopted a control-plane concept of "Virtual Private Cloud (VPC)" [7] where a tenant can easily create and manage its own private virtual network as an abstraction layer running on top of the shared network infrastructure of AWS's public cloud. In contrast, the data plane has seen little progress in provisioning the network bandwidth to combat congestion, improve physical network efficiency, achieve better scalability, and provide true isolation between competing VMs, allocating each its share of bandwidth and throttling careless or subversive ones.

To tackle these issues, cloud providers should deploy a network abstraction layer that represents a dedicated switch with guaranteed capacity, connecting various tenants' VMs [10]. In such setup, different VMs may reside on any machine in the datacenter, but each VM should be able to send traffic at a full line rate specified by the abstraction

layer, regardless of traffic patterns or workload nature generated by competing VMs [9].

The following are a few necessary components that can be integrated together towards this ultimate goal: *i)* a smart and scalable VM placement mechanism within the datacenter network that decouples bandwidth allocation from other resources. To achieve this, topologies with bottlenecks within the network core (such as uplink over-subscription or a low bisection bandwidth) should whenever possible be avoided; *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 control mechanism to ensure conformance of VM rates within the provided bandwidth, and to police misbehaving or non-conforming VMs.

A number of promising research works that tackled successfully the first two mechanisms are available today [3, 11, 17, 24, 16, 31, 14]. The proposals in [14, 24, 16] build scalable network topologies offering a 1:1 over-subscription and a high bisection bandwidth. These topologies are shown to be easily deployable in practice and can simplify the VM placement at any end-host with sufficient bandwidth. Works in [3, 11, 17, 31] target the second issue, achieving a high utilization of the available capacity via routing and transport protocols designed for datacenters. For the third issue, most proposed solutions [5, 33, 2, 1] focused on TCP congestion control and its variations to share bandwidth fairly across flows, nevertheless: *i)* all such protocols tend to be agnostic to the nature of VM aggregate traffic demands and cannot evenly distribute the capacity across competing VMs (for instance a VM could gain more throughput by opening parallel TCP connections); *ii)* the unfairness is exacerbated by the co-existence of several TCP flavours in the same network due to the variations in guest operating systems deployed in the VMs (e.g., TCP New Reno, compound TCP, Cubic TCP, DCTCP, and so on); *iii)* Finally, the increasing trend of relying on UDP transport in many emerging cloud applications, sounds the knell of any TCP-only solution to the problem [25].

Such issues have already been known to exist for two decades in the Internet as well as in data center setups [20]. Yet to ascertain they also exist in datacenter networks with small delays and different topologies, we conducted small scale simulation studies with rooted tree topologies and 10 clients on different end-hosts accessing different server applications residing on the same destination end-host. End-hosts are connected to the same ToR switch via a 1Gbps link with either DropTail, Adaptive RED or DCTCP AQM. In the studies, on tagged test client is configured with TCP NewReno with SACK enabled and the other clients run either similar TCP, DCTCP or UDP. As expected, the results[1] showed a large difference in achieved throughput for the target TCP client when coexisting with other transport protocols; in addition, against aggressive protocols such as DCTCP and UDP it achieved nearly zero throughput. Increasing the number of nodes exacerbates the problem further. To summarize, this evidence strongly supports the need for a solution that appeals to cloud operators and cloud tenants alike, and as such a good solution must adopt the following intuitive design requirements: R1) it should be simple enough to be readily deployable in existing production datacenters ; R2) it should be agnostic to the transport protocol in use to be able to stand the test of time; R3) it should not require changes to the tenant VM guest OS, nor assume any advanced network hardware capability other than those available in cheap commodity servers and switches; R4) it should be able to allocate bandwidth to the VMs in a fair manner; and R5) should scale well with the volume of traffic.

The remainder of the report is organized as follows, we first introduce the idea behind the design of HyGenICC in Section 2. We discuss our proposed methodology and present the proposed HyGenICC framework in Section 3. We show via ns2 simulations how HyGenICC achieve its requirements and discuss simulation results in Section 7. In Section 6, we discuss the implementation details of HyGenICC in Open vSwitch. In Section 7, we evaluate the performance of HyGenICC system in a small scale test-bed setup. In Section 8, we discuss some related work and finally, conclude the report in Section 9.

[1] it is known that similar problems in the Internet are what spurred the research on AQM and fair queuing two decades ago.

2 Introduction to HyGenICC

HyGenICC is easily deployable, it makes no assumption about the underlying network infrastructure and topology. The intuition behind it is based on the observation that ensuring a congestion-free network core as much as possible, enables available bandwidth partitioning in an efficient and completely distributed manner at the edge in hypervisor/vSwitch layer. To this end, HyGenICC maintains a *rate allocation mechanism* at each server to partition the available uplink bandwidth among VMs locally at the sending and receiving servers. In each such server, HyGenICC only needs to maintain state information per VM which meets design requirement R5. HyGenICC deploys a simple hypervisor-to-hypervisor (vSwitch-to-vSwitch) *congestion control mechanism* that relies on ECN (readily available in commodity switches) to infer core network congestion. HyGenICC operates at the IP level and does not interact directly with the VMs, which meets requirements R3 and R2. In addition, when detecting a highly congested path in the core network towards a destination (via ECN), HyGenICC performs admission control by refraining from accepting any further connections to this destination VM until the congestion subsides. Our design is highly scalable, responsive, work conserving and since it is IP based, it ensures fair bandwidth allocation even in the presence of highly dynamic and changing traffic patterns and transport protocols. The rate allocator resolves the contention among hundreds of co-located VMs at the servers, while the congestion control mechanism addresses the contention in the network core and pushes it back to the sources.

HyGenICC also allows administrators to assign a per-VM weight which directly affects the bandwidth reservation for the VM's Virtual Port (vport). The HyGenICC rate controller mechanism allocates bandwidth and polices the allocation locally at the end-host via a simple bank of drop-based token buckets. HyGenICC network-layer congestion controller handles network core congestion relying mainly on a feedback information from the receiving end-host. HyGenICC can be appealing from cloud service providers' perspective as it enables easier and more tangible bandwidth pricing and accounting.

To meet requirement R1, HyGenICC can be implemented either as a hypervisor-level shim layer or as an added feature to the current Open vSwitch (OvS) data-path module, to enforce rate control without any reliance on VM co-operation nor any knowledge about traffic patterns, workloads, or used transport protocol (TCP/UDP). We have implemented and tested HyGenICC following this latter approach.

3 Proposed Methodology

First we discuss HyGenICC by imagining the datacenter network as contained within one end-host where the VMs are connected via a single virtual switch. Then, we extend this design to operate in a network of end-hosts where the datacenter fabric is treated as black box that generates congestion signals whenever necessary. In a single virtual switch connecting all VMs, bandwidth contention happens at the output link to the destination when multiple senders compete to send to the same destination through the virtual switch. Hence, any contention is local to the VMs under same virtual switch where generally, the number of competing VMs within per end-host is small (around $8 - 32$ VMs per end-host depending on the available resources). A virtual switch need to fairly allocate the available physical NICs capacities and ensure compliance of the VMs with the allocated shares. Hence it needs a mechanism that detects and accounts for active VMs and apply rate limiters on a per-VM basis to allocate the bandwidth fairly among them.

HyGenICC deploys a flow table to track the state information shown in Table 1 on a VM-to-VM flow granularity (i.e., source VM-destination VM pairs). In such flow table the source is always a local active VM residing in the end-host [2]. The details of HyGenICC's two main components, the detection and bandwidth allocator and the congestion controller are discussed next.

[2]Note that, we track a bi-directional traffic in the flow table under the assumption that the majority of traffic in datacenters are TCP [5] but later we will show how to compensate for the lack of ACKs in other types of traffic.

Table 1: *Flow attributes and variables tracked in our mechanism*

Entry name	Description
$source$	IP address of source VM
$dest$	IP address of destination VM
out_packet_count	Sent packets count
ipr_packet_count	Received packets with "IPR-bit" mark
ecn_packet_count	Received packets with ECN mark
Variable name	**Description**
$rate$	The fair share rate or speed of NIC
$bucket$	The capacity of the token bucket in bytes
$tokens$	The number of available tokens to be used for transmission

3.1 VM detection and bandwidth allocation

As soon as a virtual port becomes active (sending or receiving traffic), an associated entry is created in the flow table. Whenever a new VM becomes active on a given NIC, the NIC's nominal capacity is redistributed among the token buckets of active VMs to account for the newly one. This is done by readjusting the rate and bucket size of all active VMs' token buckets on that NIC. Any extra traffic sent by the VM in excess of its fair share is simply dropped. Our implementation testbed results have shown that this simple idea is very effective in achieving the target rates without overloading the CPU. We allocate a share for each VM on each of the outgoing NICs for two reasons: 1. if each NIC is connected to a different ToR switch for multi-path routing and failure resilience, then a guaranteed bandwidth per NIC allows the active VM to have a static reservation per NIC whenever it needs to send through that uplink; and 2. if all NICs are connected to the same ToR switch as in link aggregation setups to provide the end-host with high capacity, then each VM will send through all the VMs simultaneously and its share must be pre-allocated.

After the establishment of VM entries in each matrix, immediately the VM is rate limited on outgoing NIC by its given share and any extra traffic are simply dropped. We choose to not implement a queue based token bucket rate limits as it will introduce extra processing and memory overhead in addition it is beneficial to drop noncomplying packets at end-host level rather than dropping a packet after using network resources in the network. This simple idea of rate limiting is very efficient in achieving a high forwarding rates and ensures that our mechanism does not hog the end-host CPU.

3.2 Congestion Control Mechanism

In practice, congestion may always happen within the network as shown in Figure 1, if the network is over-subscribed or does not provide full bisection bandwidth. HyGenICC therefore relies on readily available features in switches hardware[3], only these mechanisms needs to be enabled and parameters to be correctly set. To be more abstract, HyGenICC treats datacenter network as a black box in which source servers inject traffic and the black box generates ECN marks in response to congestion towards the receivers. ECN marks are a fast proactive mechanism that can help in quickly detecting any congestion from a shared queue when buffers exceed a configured queue occupancy along packet's path.

At the end-hosts, HyGenICC uses the flow table to track for each source destination the number of IP packets received with ECN or "reserved-bit" marks, regardless of the type of transport protocol (TCP, UDP or otherwise). This information is a valuable indication of the level of congestion along the path between the source VM and the destination VM starting at that particular NIC. Since HyGenICC implements a network-layer congestion control, any ECN or other marking used to track congestion is cleared before delivering the datagrams to the VM. In addition, to

[3]Most current commodity switches used in datacenters are equipped with QoS mechanisms like Strict Priority (SP), Weighted Fair Queuing (WFQ) and Weighted Random Early Detection (WRED) in addition to the ability of ECN marking of IP packets [13].

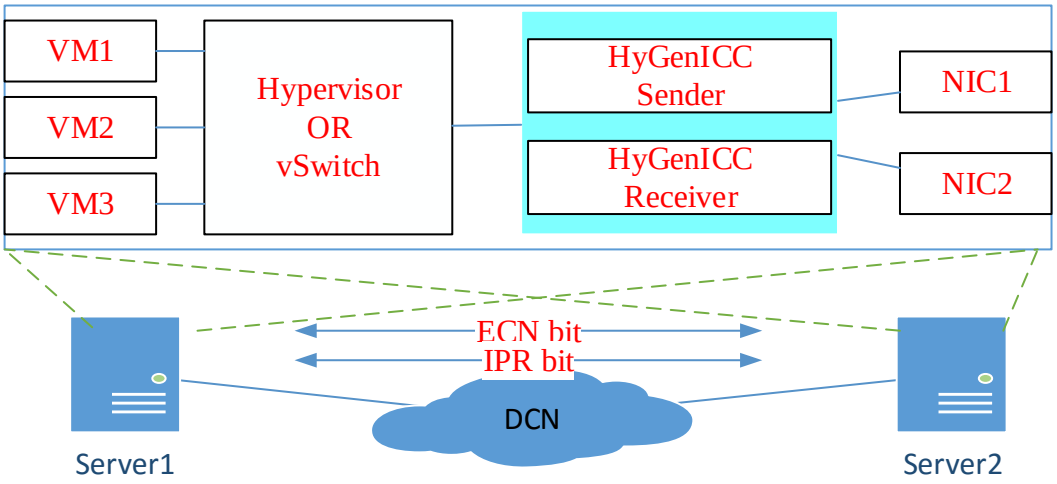


Figure 1: *HyGenICC high-level system design*

force universal ECN marking along the path, all outgoing packets are marked ECN-enabled bit.

At the receiver side, upon receiving ECN marks, HyGenICC needs to reflect the information back to the source to enable HyGenICC to reduce the sending rate of that particular source VM. To avoid introducing any additional overhead and hinder the operation of any on-path middleboxes by introducing a new protocol, we propose to piggyback the information on any returning data. For this we identify three types of traffic flows: TCP, which is by default bidirectional, other non-TCP bidirectional traffic and finally unidirectional traffic; for the three categories of traffic, we propose to use the unused reserved bit in the IP header "IPR-bit" of any reverse packet to reflect the ECN marking asynchronously to the origin. While this might be sufficient for the first two categories of traffic to carry all marking back to the source, for the third category, there might be a dramatic imbalance in the forward traffic and reverse traffic leading to some left-over marking not sent back, as a solution HyGenICC crafts a special IP packet with headers only (20 bytes of IP and 14 for Ethernet headers) and piggyback explicitly the number of remaining ECN marks on identification header field [18].

At the sender, upon receiving "IPR-bit" marks, the source decreases the VM's current allocated rate in proportion to the amount of marks and gradually increases it rate when no congestion bits are observed in a period. So the current sending rate will be in relation to the congestion level in the network

4 Algorithmic Implementation

As explained above, HyGenICC needs two mechanisms: rate limiters at the source server and congestion controller that run from source to destination server. These mechanisms can either be implemented in software, or hardware or a combination of both as necessary. For testing purposes we built HyGenICC in a small-scale test-bed as an add-on feature in the kernel datapath module of the public open-source OpenvSwitch implementation. We simplified the design and concepts of HyGenICC 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 datacenter's short RTT timescale.

4.1 HyGenICC sender

HyGenICC sender processing is described in Algorithm 1. At the senders HyGenICC tracks the *rate*, the number of *tokens*, the depth of the *bucket* and the fill-rate variables per-VM per-NIC where the per-VM rate limiters are implemented as counting token buckets that have a rate $R(i,j)$ each, a bucket capacity $B(i,j)$ each and number of tokens $T(i,j)$ each. In addition, the sender will also handle the received congestion signals from different destinations on a per-source basis.

4.1.1 Rate Allocation

Initially, the installed on-system NICs are probed and the values of their nominal data rate $R(i)$, a bucket capacity $B(i)$ and tokens $T(i)$ are calculated correspondingly. Then, when packets start flowing from each source VM, NIC capacities are redistributed and a new fair rate "$Fair\_Rate$" is calculated and used to update the entries for each active VM in the rate, tokens and bucket matrices, and are marked as currently active on all outgoing NICs.

After a certain time of inactivity [4], the bucket entries for a VM are reset and its allocation is reclaimed and redistributed among currently active VMs. As shown in Table 1, flow-table entries are established immediately after arrival of first packet using source-destination IP address. First, on arrival or departure of each packet P, its outgoing port j and incoming port i are detected. Then, the current available tokens $T(i,j)$ is retrieved and refilled based on the elapsed time since the last transmission. Then, using new $T(i,j)$, 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.

4.1.2 Congestion Reaction

The sender module reacts on regular intervals to incoming "IPR-bit" and cuts the sending rate in proportion to the amount of marking received. Hence, sources causing congestion in the network will receive "IPR-bit" signals and will react by decreasing their sending rates proportionally until the congestion subsides and congestion signals start disappearing at which time sources start gradually to increase their rates. The process will increase the rate conservatively and if no feedback arrives within $Congestion\_Timeout$ second, the rate is increased fast until it reaches its "Fair_Rate" or an "IPR-bit" is detected again.

4.2 HyGenICC receiver

At the receiver, HyGenICC needs to track incoming congestion ECN marks from the network on a per-source-destination basis and feedback this information by piggybacking it on outgoing packets heading back to corresponding sources. Hence, the operations of the receiving part is quite simple and does not incur much processing overhead onto incoming traffic. The receiver processing is described in Algorithm 2.

Each incoming packet is checked for ECN mark and the number of packets with and without the mark are traced in the flow table, Table 1, and immediately the ECN mark is cleared before re-injecting the packet in the normal packet processing path. For each ECN marked packet, an IPR-bit mark is reflected in the first available outgoing packet to that destination (it could be a TCP ACK if the flow is TCP or a UDP reply data packet) until all the ECN marks are cleared. However, when ingress and egress traffic are out of balance on a given flow, non-reflected ECN marks may start to accumulate at the receiver, to address this issue, we periodically use an explicit ICMP-like feedback packet to convey the remaining amount of ECN marks to the source. On a regular intervals close to an RTT, we scan through

[4] Inactivity timeout is set to 1 sec in simulations

Algorithm 1 HyGenICC Sender Algorithm

procedure PACKET_DEPARTURE(P, i, j)
    look up flow entry f in flow table
    $T(i,j) = T(i,j) + R(i,j) \times (now - f.senttime)$
    $T(i,j) = MIN(B(i,j), T(i,j))$
    if $T(i,j) \geq Size(P)$ **then**
        $T(i,j) = T(i,j) - Size(P)$
        $f.senttime = getcurrenttime()$
        Enable ECN Capable bits (ECT) in IP header
    else
        Drop the packet
procedure PACKET_ARRIVAL(P, i, j)
    look up flow entry f in flow table
    if Packet is congestion feedback message **then**
        $f.feedback = f.feedback + int(P.data)$
        $f.rbdetected = true$
        $f.feedbacktime = now$
        Drop the packet
    else if Packet is "IPR-bit" marked **then**
        $f.feedback = f.feedback + 1$
        $f.rbdetected = true$
        $f.feedbacktime = getcurrenttime()$
        Clear the mark and forward to the VM
procedure TIMER_TIMEOUT
    for each flow f in $FlowTable$ **do**:
        if $f.senttime - now \geq 1sec$ **then**
            $f.active = false$
            Reset f entry in Flow Table
            redistribute NIC capacity among active flows
    for each Active flow f in $FlowTable$ **do**:
        if f.feedbacktime - time() $\geq$ Congestion_Timeout **then**
            $f.rbdetected = false$
        if $f.rbdetected == false$ **then**
            $R(i,j) = R(i,j) + \frac{NIC\_Speed}{100}$
        else if $f.feedback \geq 0$ **then**
            $R(i,j) = R(i,j) - (f.feedback \times \frac{NIC\_Speed}{1000})$
        else
            $R(i,j) = R(i,j) + \frac{NIC\_Speed}{1000}$
            $f.feedback = 0$
        $R(i,j) = MAX(0, MIN(Fair\_rate, R(i,j)))$

the flow table for any flow with remaining ECN marks and that has not sent feedback for $Feedback\_Timeout$. If any is found, then an IP packet is created with unused protocol ID value and the value of ECN marks added as a 2-bytes payload of this packet addressed to the source of the flow. This event is infrequent and unlikely to exist but if so, will not incur much network overhead as the packet size would be 36 bytes (14-bytes Ethernet header + 20-bytes IP header + 2-bytes payload data). To compress further the explicit feedback, the 2 bytes payload can be piggybacked instead in the IP header identification field.

Algorithm 2 HyGenICC Reciever Algorithm

procedure PACKET_ARRIVAL(P, i, j)
    look up flow entry f in flow table
    if Packet is ECN marked **then**
        $f.ecnmarks = f.ecnmarks + 1$
        Clear the mark and forward to the VM
    if f.feedbacksenttime - time() $\geq$ feedback_timeout **then**
        Create IP feedback message and send to $f.source$
        $f.feedbacksenttime = getcurrenttime()$
        $f.ecnmarks = 0$
procedure PACKET_DEPARTURE(P, i, j)
    look up flow entry f in flow table
    if $f.ecnmarks \geq 1$ **then**
        Set "IPR-bit" flag in IP header
        $f.feedbacksenttime = getcurrenttime()$
        $f.ecnmarks = f.ecnmarks - 1$

5 Simulation Analysis

We study the performance of our algorithm via ns2 simulation in network scenarios with a high bandwidth-low delay (as is the case in data centers). We compare the performance achieved by a tagged VM using TCP when competing against other VMs using TCP, DCTCP, and UDP in 1) our system, 2) a system that does not use such traffic management and relies on end-to-end congestion control and 3) a system that uses a central control node to perform static fair bandwidth allocation. We have compared two TCP flavours with ECN and without ECN to show that TCP's reactive nature to ECN is not enough to achieve fair allocation especially when competing with non-responsive flows running UDP.

For HyGenICC, there is a single parameter settings of timeout interval for updating flow rates which should be larger than a single RTT, in simulations we set it to 5 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 rather than using the weighted average queue length.

5.1 Simulation Setup

We use ns2 version 2.35 [26], which we have extended with a HyGenICC module inserted at the link elements in topology setup[5]. In addition, we patched ns2 using the publicly available DCTCP patch[4]. We compare TCP newReno with SACK-enabled when competing against TCP, DCTCP and UDP under the three systems. We considered two cases, one where TCP is ECN-bit responsive and one when TCP is not[6]. We use in 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 RTO_{min} of the default 200 ms.

We use a rooted tree topology with single bottleneck at the destination and run the experiments for a period of 15 sec. The buffer size of the bottleneck link is set to more than the bandwidth-delay product in all cases (100 Packets or 150 KBytes respect.), the IP data packet size is 1500 bytes.

[5] Simulation code is available upon request from the authors
[6] Without ECN, DCTCP will act exactly as TCP

5.2 Simulation Results and Discussion

We simulated several scenarios that lead all to the same results. So for ease of exposition and clarity of the figures we consider a toy scenario with 2 elephant flows, a tagged flow and a competitor. In the experiments, the tagged flow always uses TCP newReno and competes with other flows (in the toy scenario only one other flow) all using the same protocol either TCP newReno, DCTCP or UDP. The competitor flows start at the beginning and finish at the 10th second whereas the tagged flow starts at the 5th second and runs to the end of the simulation. So typically from 0 to 5s only the competitors occupy the bandwidth, from 5s to 10s bandwidth is shared by the two groups, and from 10s to 15s only the tagged flow uses the bandwidth. This experiment is designed to demonstrate the work conservation-ability, the fairness, and the convergence speed of HyGenICC compared to other alternatives.

Figure 2 shows the goodput and aggregate goodput for TCP at the destination in the 2 flows scenario. As shown in Figure 2a, without any rate limits, TCP struggles to grab any bandwidth when competing with DCTCP and UDP and its throughput is not stable when competing with TCP without ECN. Figure 2b suggests that ECN can partially solve the problem by allowing TCP flow to be responsive to congestion events at the network, however the achieved throughput reaches the fair share only when the competitor uses the same TCP protocol. This is attributed to the fact that DCTCP reaction to ECN marks as it does not cut its window by half like TCP does. Figures 2c and 2d show that a centralized node assigning per VM static rates can achieve perfect fairness but is not efficient as it does not achieve work-conservation. Figures 2e and 2f show that HyGenICC dynamic rate limiters that respond to congestion signals achieve both fair rate allocation and work conservation regardless of the competing transport protocol. Hence, HyGenICC is able to converge to the current network-wide fair-share for the TCP flow in all cases and keeps the network links fully utilized all the time.

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

We also implemented the algorithm in a real testbed as an added feature of the popular Open vSwitch (OvS) [28]. In the next section, we will discuss the implementation details along with the evaluation results from the testbed experiments.

6 HyGenICC in OpenvSwitch

We have decided to implement the algorithm in a real testbed to demonstrate its practical feasibility. Two options were available where one of them were implemented and deployed while the second one was left for future work as follow:

- Open vSwitch: HyGenICC implementation is realized as an added feature to OVS by adding HyGenICC's sender and receiver algorithm's logic by patching the OVS datapath kernel module;
- Linux Kernel Module: HyGenICC can be implemented using Linux netfilter framework [23] which is used for intercepting the packets using a pre-registered hooks in the processing pipeline of the network stack which was left for future work.

Open vSwitch can operate both as a soft switch running within the hypervisor, and as the control stack for switching silicon. It has been ported to multiple virtualization platforms and switching chipsets. It is the default switch in XenServer [12], the Xen Cloud Platform and also supports Xen, KVM, Proxmox VE and VirtualBox. It has also been integrated into many virtual management systems including OpenStack, openQRM, OpenNebula and oVirt. The kernel datapath is distributed with Linux, and packages are available for Ubuntu, Debian, and Fedora. Open vSwitch is also supported on FreeBSD and NetBSD. Open vSwitch is also available on all Pica8's bare-metal switches [29],

current PicOS's OVS implementation runs as a process within PicOS, providing the OpenFlow interface for external programmability.

Typically, a data center network consists of servers and switches interconnecting them as shown in Figure 3. In oversubscribed data centers, the contention points in the network as shown in red, are within the server and the uplinks to upper layers (i.e the link from Top of Rack switch to the core switch). Contention within the server happens due to the over-subscription of server's NIC by many active VMs which can be up to 80 VMs or more and the contention in the uplink when multiple servers in a single rack share it to reach servers in other racks. In our HyGenICC system design, we propose to use production commodity switches with ECN marking capability as ToR and Core layers for interconnecting servers within the datacenter. Further, VMs within servers are interconnected using a software OVS which is the most popular choice for most cloud management frameworks like OpenStack. HyGenICC as an OVS-patch provide a potential for easy deployment in production datacenters at all servers with minimal administration overhead.

We have fully implemented HyGenICC in OpenvSwitch as it is the virtualization switching technology used by the famous OpenStack cloud managment software [27]. We simply patched the Kernel datapath module of OpenVSwitch with the same functions described above of HyGenICC[7]. We have added HyGenICC functions in the processing pipeline of the packets that pass through the kernel datapath module of OpenVSwitch. In a virtualized environment, HyGenICC-enabled OpenvSwitch can process the traffic for inter-VM, Intra-Host and Inter-Host communications. This is an efficient way of deploying HyGenICC on the host operating system of end-hosts under Hypervisor layer as shown in Figure 3. The patch file needs only to be applied to OVS source code then OpenvSwitch module is recompiled and deployed, making it an attractive option for immediate deployment in today's production datacenters.

As shown in Figure 4, OpenVswitch is mainly composed of two parts, the data path kernel module and the user-space vswitch daemon which communicates with the controller using OpenFlow protocol over encrypted SSH connection. OpenvSwitch is flow-aware by design and all flow decision entries in the forwarding table are inserted by a local or a remote controller. Whenever, a packet arrives at any port of the switch, its flow key is hashed and examined against current active flows in the table. If the entry for that flow could not be found, the packet is immediately forwarded to the controller for establishing the identity of this flow and setting up the forwarding entries in all involved switches of the network. Primarily, any packet is processed by the kernel fast data-path only if its flow entry is active in the forwarding table, in such case, the packet is forwarded immediately without experiencing any further delays.

6.1 HyGenICC Modules

Below are the main procedures that constitute the HyGenICC 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[8]. We show also the timer_callback function which is our rate update daemon that fires every interval. In addition, we show the hygenicc_send and hygenicc_send functions which are responsible for handling the arrival and departure events of the packets in the Open Vswitch, they are injected the packet processing pipeline of the Open vSwitch kernel datapath module[9].

Listing 1: *Find Active VMs*

[7] source code or patch file is available up on request from the authors

[8] We use a different mechanism to react to congestion rather than the AIAD algorithm described in section 4. We use the average of the moving average of the fraction of marked packets in each interval

[9] HyGenICC 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

```
//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 * EVIL_DEV_MAX + j] = false;
        reset_hygenicc_dev(j);
      }
      else
      {
        isactive[i * EVIL_DEV_MAX + j] = true;
        active_count++;
      }
    }
  }
  if(active_count>1)
   fair_rate[i] = eth_rate[i] / active_count;
}
```

Listing 2: *HyGenICC Update Rates Procedure*

```
//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;
       rate[i * DEV_MAX + j] = MAX(MIN_RATE, fair_rate[i] - lostrate);
       }
  }
}
```

Listing 3: *HyGenICC Rate Update Daemon*

```
//Rate Update Daemon
hrtimer_restart timer_callback(hrtimer *timer)
{
     int i,j;
  long int now=jiffies;
    if(hygenicc_enabled() && hygenicc_timerrun)
    {
     //calculate average alpha value of each VM
       for(i=0; i<ethdevcount; i++)
       for(j=0; j<virtdevcount; j++)
     if(isactive[i * DEV_MAX + j])
         avgalpha[j]=track_get_avg_alpha(virtipaddress[j]);
     else
      avgalpha[j]=0;

    //Update rates based on the new average alpha
```

```
    update_rates();

    //Restart the timer for a new interval
        ktime_t ktnow = hrtimer_cb_get_time(&hygenicc_hrtimer);
        hygenicc_ktime = ktime_set(0 , interval * ((unsigned long) 1E3L) );
        hrtimer_forward(&evil_hrtimer, ktnow, hygenicc_ktime);
        return HRTIMER_RESTART;
    }
    else
        hygenicc_timerrun=false;
    return HRTIMER_NORESTART;
}
```

Listing 4: *HyGenICC Sender Packet Handler*

```
//Sender Packet Handler
void hygenicc_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 * EVIL_DEV_MAX + j]=jiffies;
    sent[i * EVIL_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 * EVIL_DEV_MAX + j]= MAX(0, tokens[i * EVIL_DEV_MAX + j] - skb->len);
         else
     {
             kfree_skb(skb);
             return;
         }
     //enable ECN on all outgoing packets
         enable_ecn(ip_header);

     if(flow)
         {
             if(flow->ecn_packet_count >0)
             {
                 insert_rb_bit(ip_header);
                 flow->ecn_packet_count = MAX(0, flow->ecn_packet_count-1);
         flow->ecn_lastfeedback = jiffies;
       }
      }
     else
              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 5: *HyGenICC Receiver Packet Handler*

```
//Receiver Packet Handler
void hygenicc_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)
   {
       rb_count = ntohs(ip_header->id);
       track_reverse_key_extract(skb, &track_key);
       flow = track_tbl_lookup(&track_key, sizeof(track_key));
       if(flow)
    {
           flow->rb_packet_count += rb_count;
      flow->rb_lastfeedback = jiffies;
    }
       kfree_skb(skb);
       return;
   }

  if (m>=0 && n>=0)
   {
    //check and clear ECN bit
       ecn_pkt =is_ecn(skb, ip_header->tos);
       if(ecn_pkt)
     clear_ecn(ip_header);

       //check and clear RB bit
       rb_pkt=is_rb_bit(skb, ntohs(ip_header->frag_off));
       if(rb_pkt)
            clear_evil(ip_header);
   }
}
```

7 Experimental Evaluation

In this section, we build a small-scale testbed to experimentally evaluate HyGenICC in a real setup to understand the following behaviors:

- Performance : We setup test cases where different flavors of source transport protocols (TCP and UDP) used by VMs to transmit and how the system can regulate rate allocation and penalize misbehaving sources.
- Congestion handling : We test how network congestion control benefit all applications regardless of transport protocols (TCP and UDP) they are running on top of.
- scalability : We test HyGenICC's scale in face of increasing number of VMs allocated within each end-host up to the numbers seen in real production datacenters.
- Responsiveness: We stress HyGenICC with a large number of misbehaving UDP streams and how the system is still be able to guarantees conformance to the allocated fair rates.

- System overhead: We evaluate the main CPU overhead of HyGenICC addition to the OpenvSwitch and show that its simplicity does not add to CPU utilization.

.

7.1 Testbed Setup

To experiment with the HyGenICC enabled OpenvSwitch, as shown in Figure 5, we set up a small-scale testbed consisting of 6 Lenovo and 7 Dell desktops configured with core2duo processors and 4G of ram. One of the Dell desktops acts as the switch and it is equipped with three 1 Gbps Ethernet cards.In this setup, 7 machines are running Ubuntu 14.04 Desktop Edition with Linux kernel V3.13.0.34 while the other 6 machines are running CentOS 6.6 with Linux kernel V3.10.63. All machines are running an Apache web server hosting **"index.html"** webpage of size 11.5KB. We mainly use two well-known measurement applications for our experiments, iperf [19] for generating both TCP and UDP elephants and Apache benchmark [8] for generating short-lived web-like traffic. The base RTT in our testbed is around $\approx$200μs. We allocate a static buffer size of 100KB to all NICs in the end-hosts and switch server using Linux Traffic Control (Linux TC) [22]. In all experiments, we set up the queuing discipline Qdisc of each Ethernet port for end-hosts as a bfifo queue with limit of 100KB bytes while the switch's ports are configured with RED queuing discipline with MIN and MAX marking thresholds equal to 10% of buffer capacity and queue sample weight set to 1 to mark based on instantaneous queue length. We evaluated the experiments with the well-known new-reno TCP, which is readily available as one of TCP congestion control mechanisms in both Linux kernel versions (3.10 and 3.13). In our setup, All VMs ports[10] are connected to the patched OpenvSwitch and the CentOS and Ubuntu hosts are connected to a different 1 Gb/s D-Link dumb-switch. A number scenarios are set up to produce incast and buffer-bloating situations with TCP and UDP where multiple-bottleneck links in the network exist indicated as red links.

7.2 Experimental Results

The goals of the experiments are to: *i)* show that with the support of HyGenICC, TCP are not penalized due to co-existence of misbehaving UDP traffic; and *ii)* show that HyGenICC can efficiently allocate bandwidth among all types of transport protocols and maintain high link utilization; and *iii)* show that HyGenICC can benefit mice flows by achieving faster Flow Completion Time (FCT).

7.2.1 TCP and UDP start at the same time

We produce an buffer-bloating scenario with synchronized senders all converging to the same output port per sending host and per destination at the switch level resulting in excessive pressure on the output buffer in uplinks 1 and 2 as well as ToR link 3. First, we generate 4, 8 and 16 long-lived iperf clients for 30 seconds at same time from each of the 2 senders destined to a separate iperf server listening on a separate port on the receivers where half of them are using TCP and the other half is using UDP. This results in 8, 16 and 32 senders continuously sending for 30 secs and iperf is set to generate the throughput samples over 0.5 secs intervals. For mice traffic, we use Apache benchmark to request **"index.html"** webpage (representing mice flows) from each of the web servers ($2 \times 2 = 4$ in total) running on the same machines where elephants are sending. Note that, we run Apache benchmark, at the 10^{th}sec and the 20^{th}sec, requesting the webpage 100 times then it reports different statistics over the 100 requests. In the following we show the CDF of the average achieved throughput for UDP and TCP and average and 99^{th} of flow completion time (FCT).

[10]VM ports are created by creating multiple ports on the OpenvSwitch and binding an iperf or an Apache benchmark process to each one of them.

Figures 6 to 8 show that, under different load situations, HyGenICC helps the competing VMs whether using TCP or UDP as the transport protocol in achieving a balanced distribution of bandwidth among competing senders. Figures 6c, 6d, 7c, 7d, 8c and 8d show that, HyGenICC achieves a good balance in meeting the conflicting requirements of elephants and mice. The competing mice flows benefit under HyGenICC, due to the fair bandwidth allocation, by achieving a nearly equal FCT on average with very small variation among different mice flows compared to TCP without HyGenICC as shown in figs. 6a, 7a and 8a. In addition, as HyGenICC efficiently controls the congestion in the network and adpat the rate limiters to current congestion level, infigs. 6b, 7b and 8b, the 99^{th} percentile of completion time for HyGenICC never crosses the 200ms threshold which is the default RTO_{min} of Linux, as opposed to Reno operating in environment without HyGenICC where 99^{th} percentile can reach up to .

7.2.2 UDP start at t=5 to t=25

Here, we repeat the same experiments above for the 8 and 16 senders but instead TCP iperf clients starts first and UDP clients starts at the 5^{th} second and ends at the 25^{th} second. The intent of this experiment is to observe the effect of congestion-unresponsive UDP flows when they are introduced in the network while TCP flows are already taking all the bandwidth. As shown in figs. 9 and 10, HyGenICC is able to efficiently achieve its goals of allocating the bandwidth for incoming UDP flows without greatly affecting the performance of currently active TCP flows. Figures 9a and 10a clearly show that, with help of HyGenICC, mice flow's low-latecny requirement is achieved as well even with the existence of large number of elephant flows in the network. The mean FCT under HyGenICC are quite small and the CDF curve is smooth in contrast to what is observed without HyGenICC. In addition, figs. 9b and 10b show that, the 99^{th} percentile of the FCT of mice flows without HyGenICC is experiencing lots of timeouts and in some cases are not able to establish the connection at all as indicated by 99^{th} percentile FCT values. Meanwhile HyGenICC can help mice flows to avoid possible timeouts by efficiently managing and allocating the bandwidth and as a result the FCT of mice on the tail 99^{th} percentile is greatly reduced.

7.2.3 TCP start at t=5 to t=25

Here, we repeat the same experiments above for the 8 and 16 senders but instead UDP iperf clients starts first and TCP clients starts at the 5^{th} second and ends at the 25^{th} second. The intent of this experiment is to observe whether TCP flows can garbs their own share of the network when the UDP flows are already taking all the bandwidth. As shown in figs. 11 and 12, HyGenICC is able to efficiently achieve its goals of allocating the bandwidth among competing entities in the network even-though there are congestion-unresponsive flows like UDP which are taking all network bandwidth. Figures 11a and 12a clearly show that, with HyGenICC in place, mice flows are not blocked by the bandwidth-hogging elephants. The mean FCT under HyGenICC are quite small and the CDF curve is smooth in contrast to what is observed without HyGenICC. In addition, figs. 10b and 11b show that, the 99^{th} percentile of the mice flows without HyGenICC is experiencing lots of timeouts and in some cases are not able to establish the connection at all as indicated by 99^{th} percentile FCT values. Meanwhile HyGenICC can avoid timeouts by managing and allocating the bandwidth efficiently and hence it greatly reduces the FCT of mice on the tail 99^{th} percentile.

8 Related Work

HyGenICC can be comparable or complementary to a number of works on cloud network resource allocation that have been proposed recently. Seawall [32] is a system proposed for sharing network bandwidth, it provides per-VM max-min weighted fair share using explicit feedback end-to-end congestion notification based on losses for rate adaptation. Seawall requires modifications to network stack which incurs a large overhead and may interfere with

middleboxes operations. SecondNet [15] is designed to divide network among tenants and enforce rate limits, but is limited to providing static bandwidth reservation between pairs of VMs. Oktopus [10] argues for predictability by enforcing a static hose model using rate limiters. It computes rates using a pseudo-centralized mechanism, where VMs communicate their pairwise bandwidth consumption to a tenant-specific centralized coordinator. This control plane overhead limits reaction times to about 2 seconds which is inadequate for the fast changing and dynamic traffic nature in datacenters. FairCloud [30] designs better policies for sharing bandwidth and explored fundamental trade-offs between network utilization, minimum guarantees and payment proportionality, for a number of sharing policies. EyeQ [21] provides per-VM max-min weighted fair shares in the context of a full bisection bandwidth datacenter topology where congestion is limited to the first and the last hops. pFabric [6] proposes a switch design which ensures very small buffers by assigning priorities (e.g., based on remaining bytes in that flow or deadline) and forwarding based on strict priority. pFabric achieves near-optimal flow completion times, but suffers from starvation problem for long flows with low priorities. By simplifying rate limiters and incorporating congestion control to make them dynamic entities rather than static, HyGenICC can achieve similar objectives as these proposals in an easy to deploy manner with minimal CPU and network overhead. HyGenICC is designed to operate with commodity infrastructure and traditional protocols used by current production datacenter/cloud, to be a readily deployable. Finally, HyGenICC leverages the popularity of Open vSwitch usage by cloud management frameworks like openstack to implement its mechanism with minor modifications that do not require any new protocols, software and hardware.

9 Conclusion and future work

In this report, we set to design, build, implement and evaluate HyGenICC a system that can fit easily within current production cloud setups to achieve better bandwidth isolation and improved application performance. HyGenICC is a hypervisor (vswitch) level framework to enforce fair and guaranteed network bandwidth allocation among competing VMs. Our intuitive analysis and small-scale testbed experiments show that simple mechanisms like rate limiting token buckets allied to ingress-egress congestion control protocol can lead to a simple, scalable and readily deployable system design for cloud network resource isolation. HyGenICC is built with three main objectives in mind, low overhead, commodity hardware, and no changes to network hardware or VMs protocol stack. This constitutes a great incentive for deployment in today's production datacenter networks. HyGenICC requires minimal human intervention and can flexibly and efficiently divide network bandwidth across active VMs by giving each VM endpoint a predictable minimum bandwidth and hence bounded latency. Regardless of the transport protocol used by the applications residing in the VMs and even with the existence of misbehaving or bandwidth-hungry traffic, HyGenICC can achieve its design goals. We plan to test HyGenICC on a larger scale testbed with high end servers with a total of 168 CPU cores and 96 interface cards, which enable us to test our implementations under stress with several thousand flows.

References

[1] A. M. Abdelmoniem and B. Bensaou. Incast-Aware Switch-Assisted TCP congestion control for data centers. In *2015 IEEE Global Communications Conference: Next Generation Networking Symposium (GC' 15 - Next Generation Networking)*, San Diego, USA, Dec. 2015.

[2] A. M. Abdelmoniem and B. Bensaou. Reconciling mice and elephants in data center networks. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet) (CLOUDNET'15)*, pages 7–12, Niagara Falls, Canada, Oct. 2015.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

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

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM Computer Communication Review*, 40:63, 2010.

[6] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. *Proceedings of the 11th ACM Workshop on Hot Topics in Networks - HotNets-XI*, pages 133–138, 2012.

[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] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications ACM*, 53(4):50–58, Apr. 2010.

[10] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):242, 2011.

[11] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: A cloud networking platform for enterprise applications. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 8:1–8:13, New York, NY, USA, 2011. ACM.

[12] Citrix. Best practices for using open vswitch with xenserver. http://support.citrix.com/servlet/KbServlet/download/37340-102-710035/Best%20Practices%20for%20using%20Open%20vSwitch%20with%20XenServer.pdf.

[13] EdgeCore. EdgeCore AS4600-54T Datacenter ToR switch. http://www.edge-core.com/ProdDtl.asp?sno=424&AS4600-54T.

[14] 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 the ACM SIGCOMM 2009 conference on Data communication*, volume 09, pages 51–62, 2009.

[15] 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 *Proceedings of the 6th International COnference on emerging Networking EXperiments and Technologies*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[16] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, 2008.

[17] C. Hopps. Analysis of an equal-cost multi-path algorithm, 2000. https://tools.ietf.org/html/rfc2992.

[18] IETF. Internet Protocol (IP) specification, 1981. http://tools.ietf.org/html/rfc791.

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

[20] S. M. Irteza, A. Ahmed, S. Farrukh, B. N. Memon, and I. A. Qazi. On the coexistence of transport protocols in data centers. In *2014 IEEE International Conference on Communications (ICC)*, pages 3203–3208. IEEE, June 2014.

[21] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 297–312, Berkeley, CA, USA, 2013. USENIX Association.

[22] kernel.org. Linux Traffic Control command. http://lartc.org/manpages/tc.txt.

[23] NetFilter. NetFilter Packet Filtering Framework for linux. http://www.netfilter.org/.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat, and R. N. Mysore. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09 Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 39–50, 2009.

[25] R. Nishtala, H. Fugal, and S. Grimm. Scaling memcache at facebook. *NDSI'13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, pages 385–398, 2013.

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

[27] OpenStack. OpenvSwitch setting in OpenStack framework. http://docs.openstack.org/admin-guide-cloud/content/under_the_hood_openvswitch.html.

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

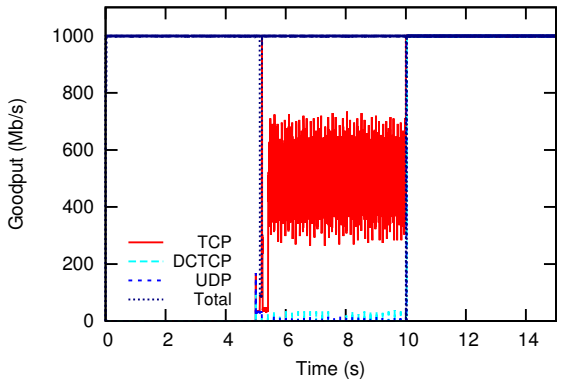
[29] Pica8. Pica8: Open vswitch (ovs) overview. http://www.pica8.com/open-technology/open-vswitch.php.

[30] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. *Computer Communication Review*, 42:187–198, 2012.

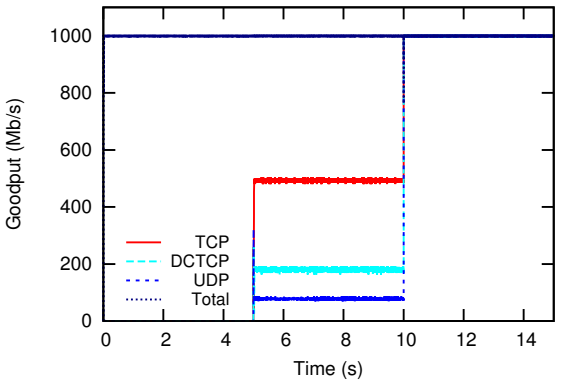
[31] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM - SIGCOMM '11*, volume 41, page 266, New York, New York, USA, Aug. 2011. ACM Press.

[32] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, page 23, 2011.

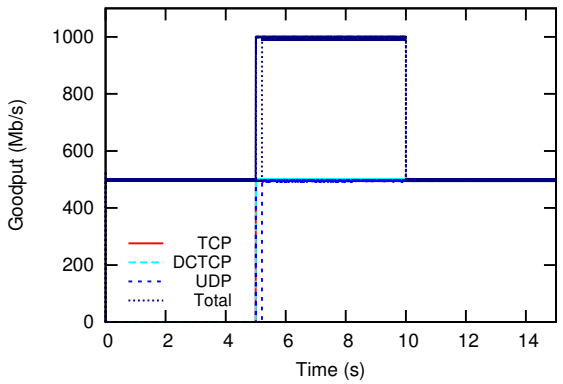
[33] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM Transactions on Networking*, 21:345–358, 2013.



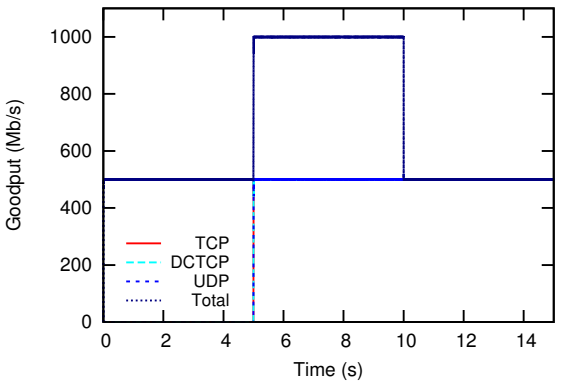
(a) *No ratelimit without ECN*



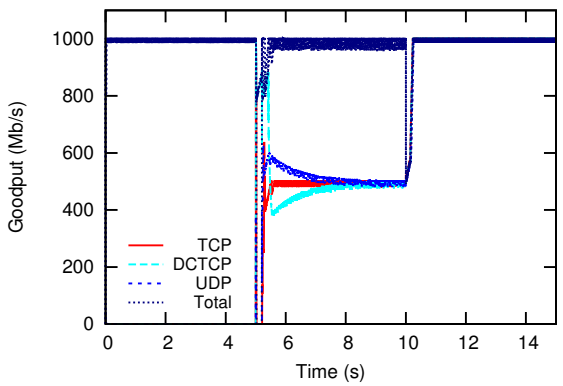
(b) *No ratelimit with ECN*



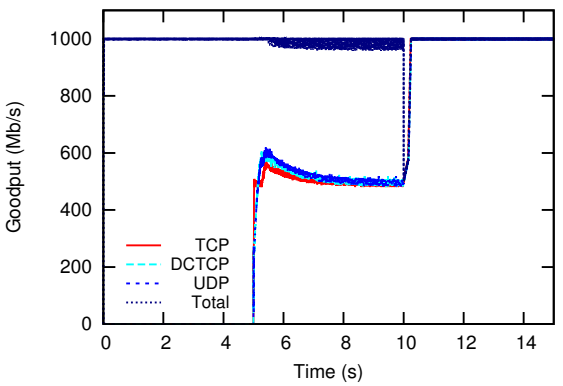
(c) *Static ratelimit without ECN*



(d) *Static ratelimit with ECN*



(e) *HyGenICC ratelimit without ECN*



(f) *HyGenICC ratelimit with ECN*

Figure 2: *Goodput of a single TCP flows and Aggregate goodput at the destination*

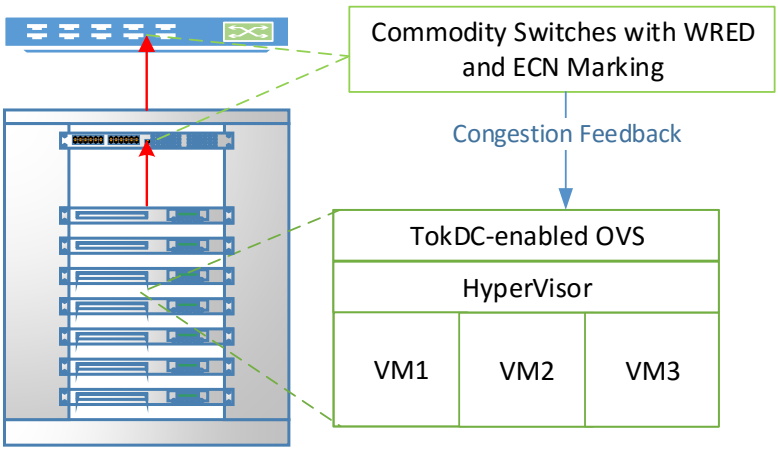


Figure 3: *HyGenICC datacenter system*

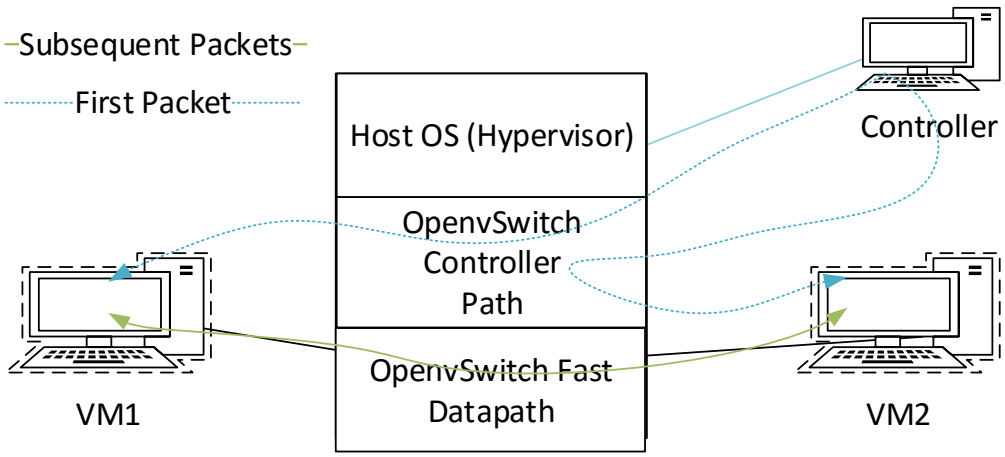


Figure 4: *OpenvSwitch Packet Processing Path*

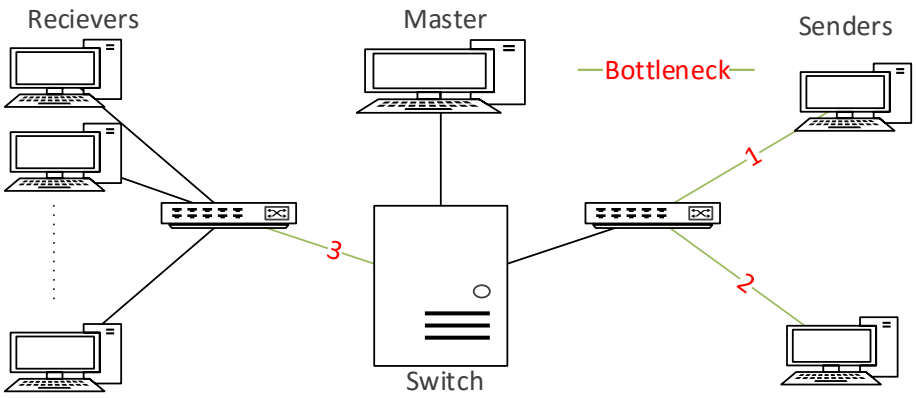
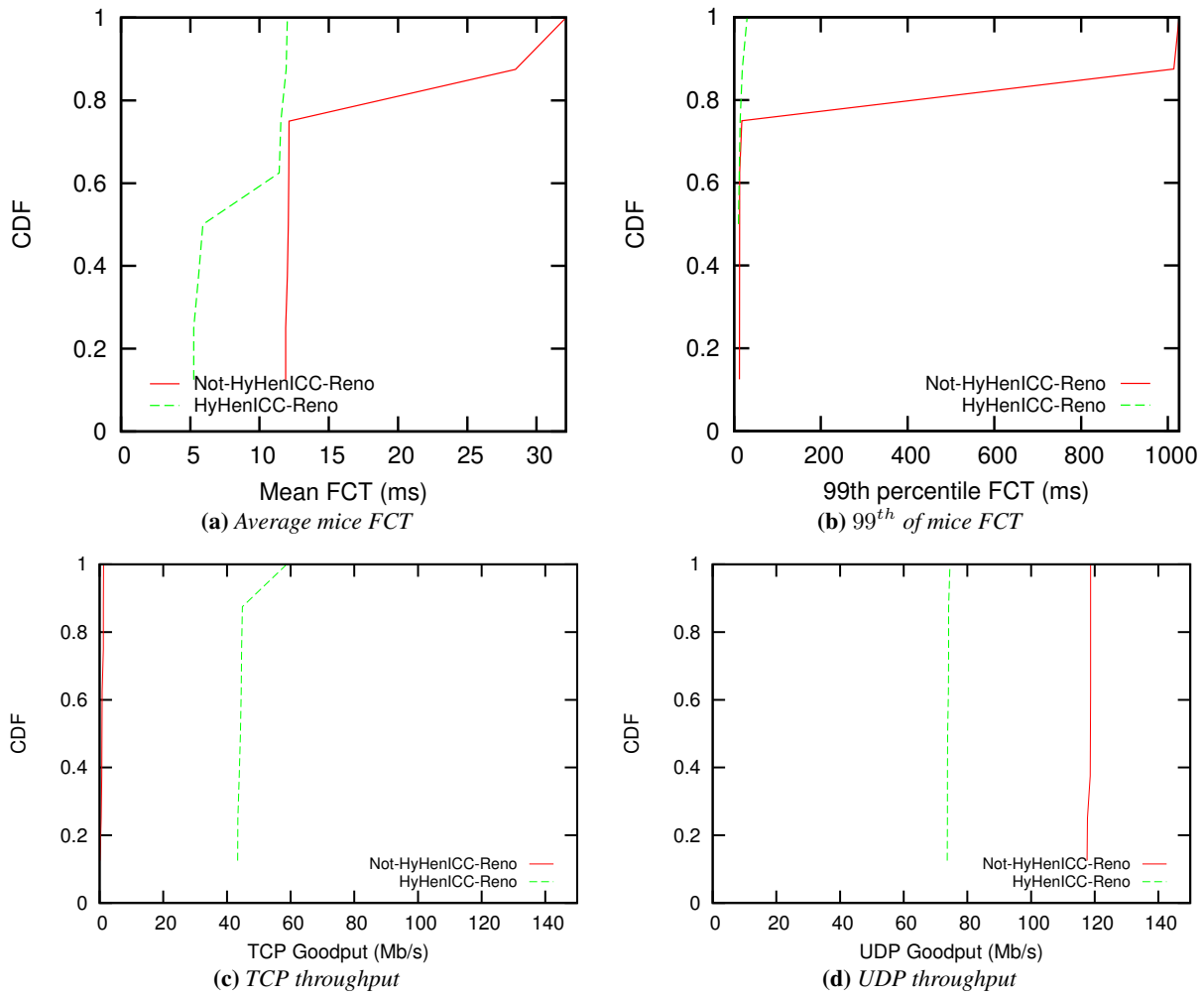


Figure 5: *Private small cloud testbed for testing RWNDQ-enabled OpenvSwitch*



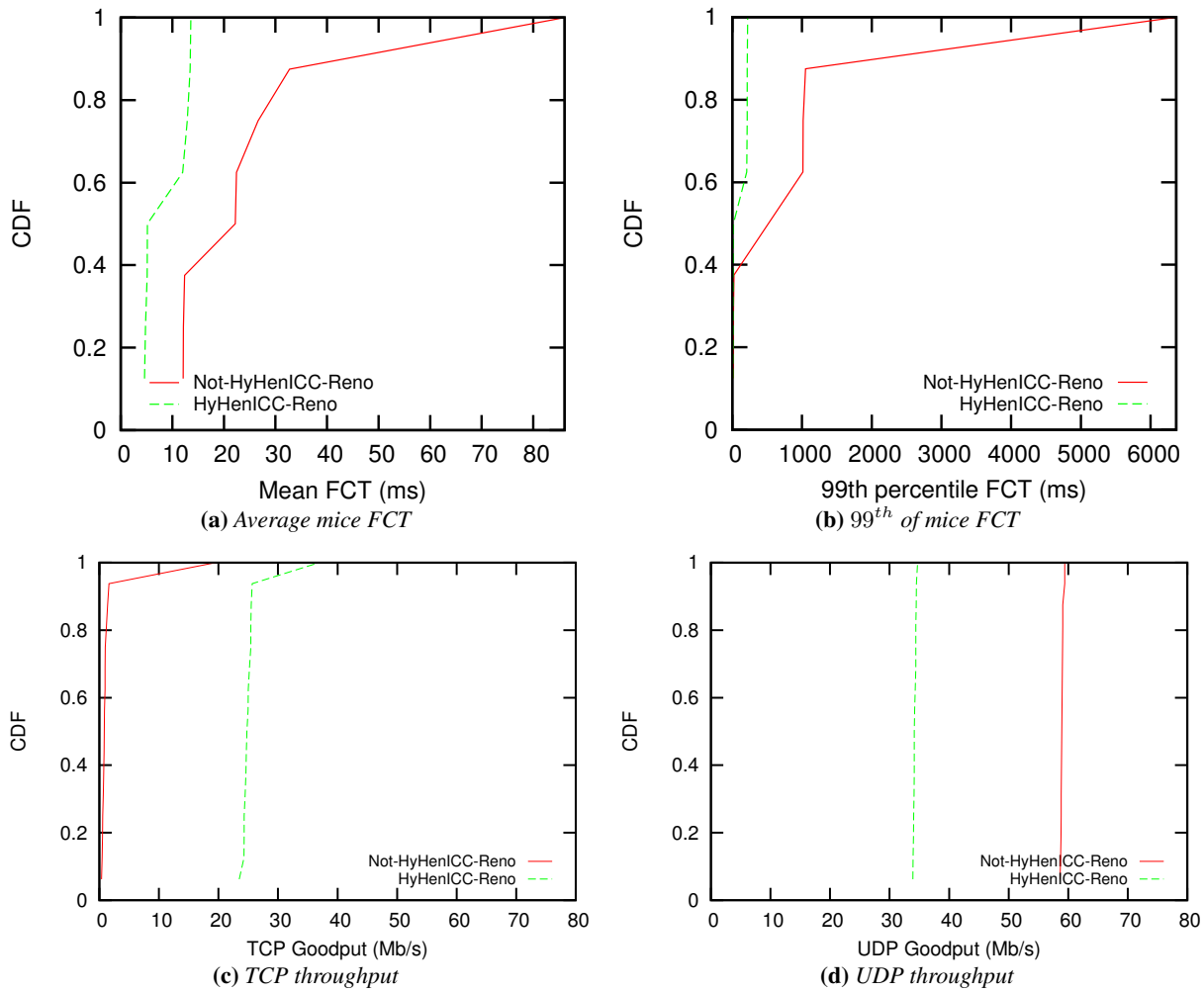
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 6: *Comparison of mice FCT and elephant goodput for HyGenICC in 8 sender scenario*



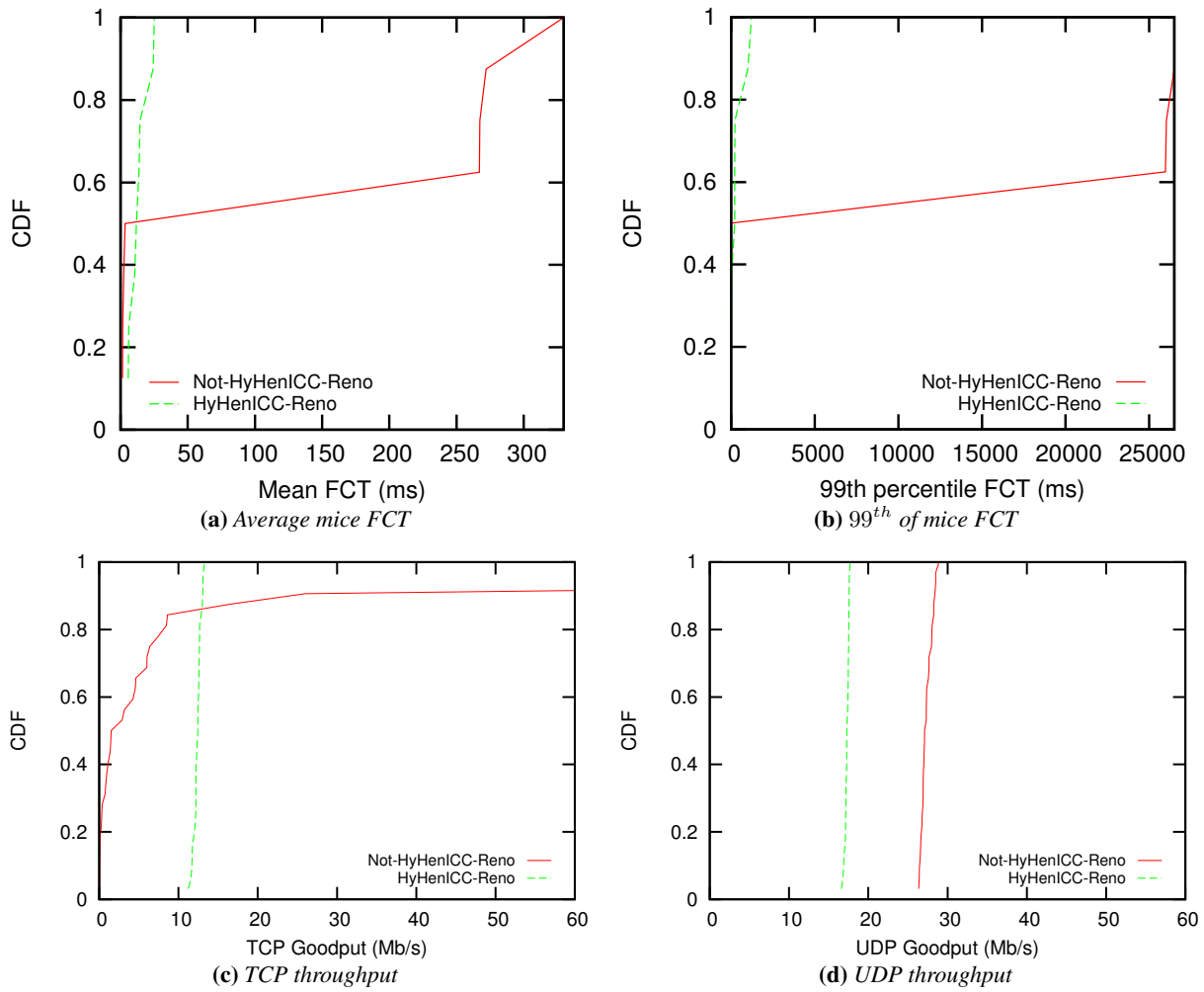
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 7: *Comparison of mice FCT and elephant goodput for HyGenICC, , 16 sender*



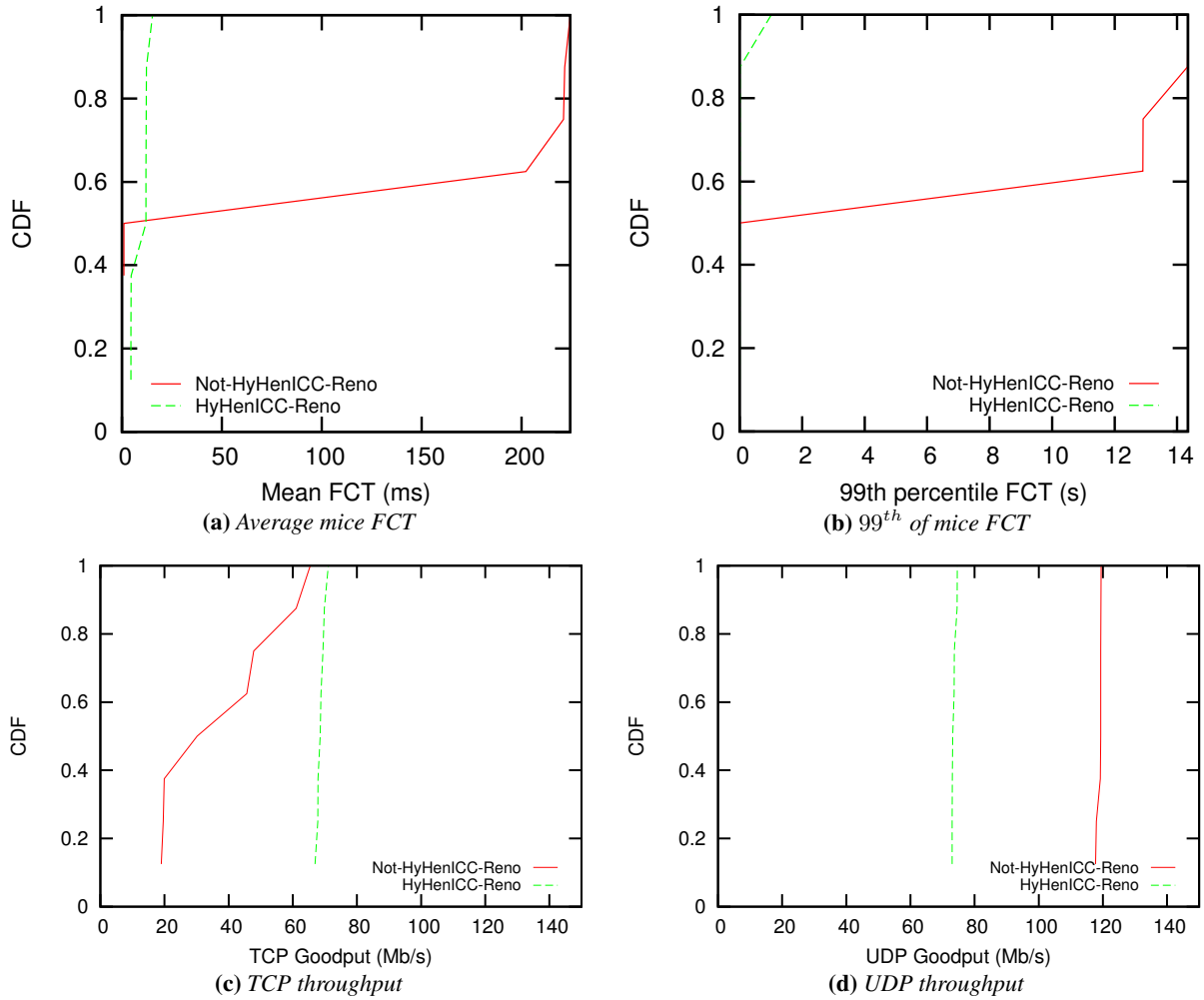
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 8: *Comparison of mice FCT and elephant goodput for HyGenICC, 32 sender*



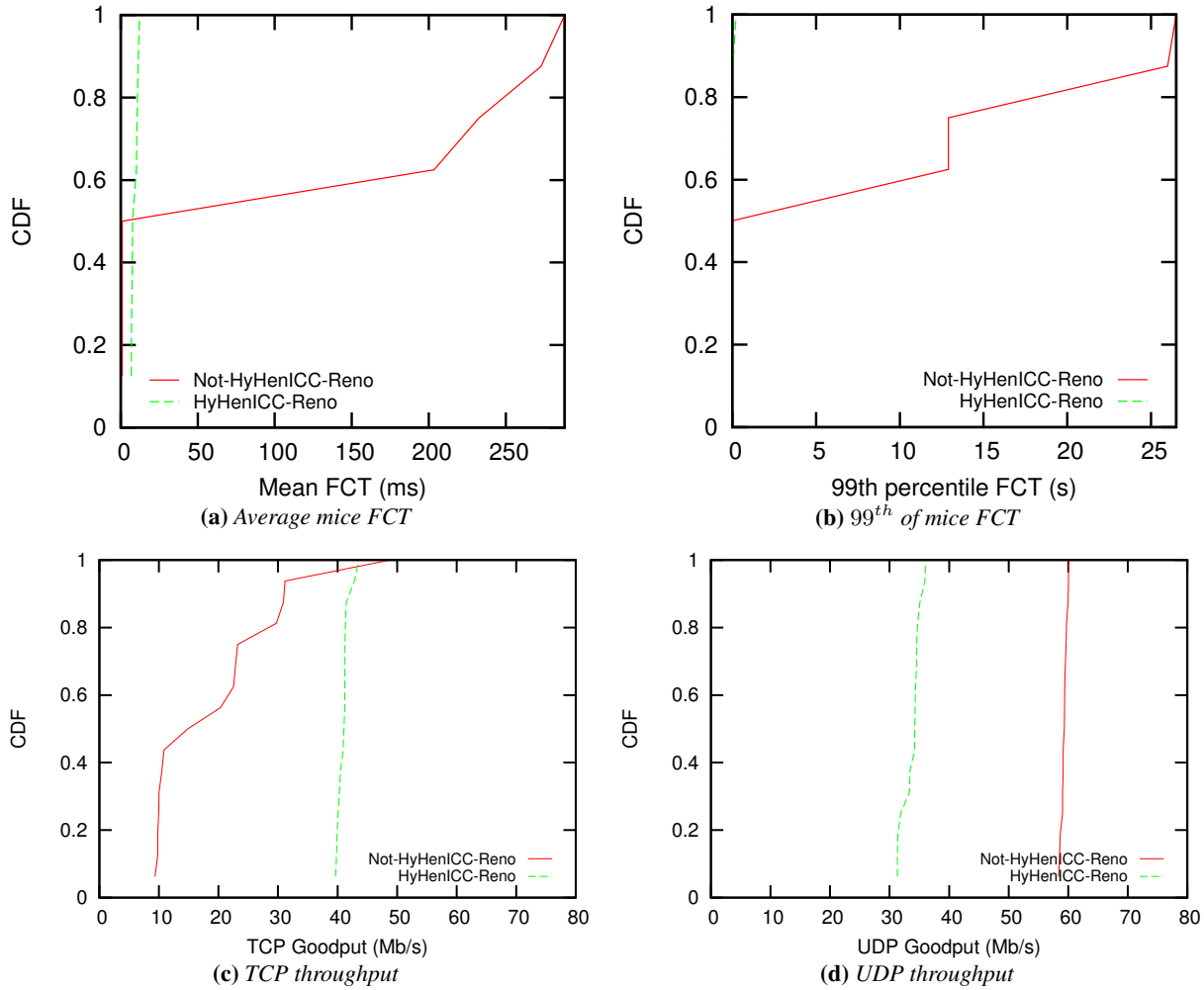
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 9: *Comparison of mice FCT and elephant goodput for HyGenICC in 8 sender scenario where UDP starts at t=5 and ends at t=25*



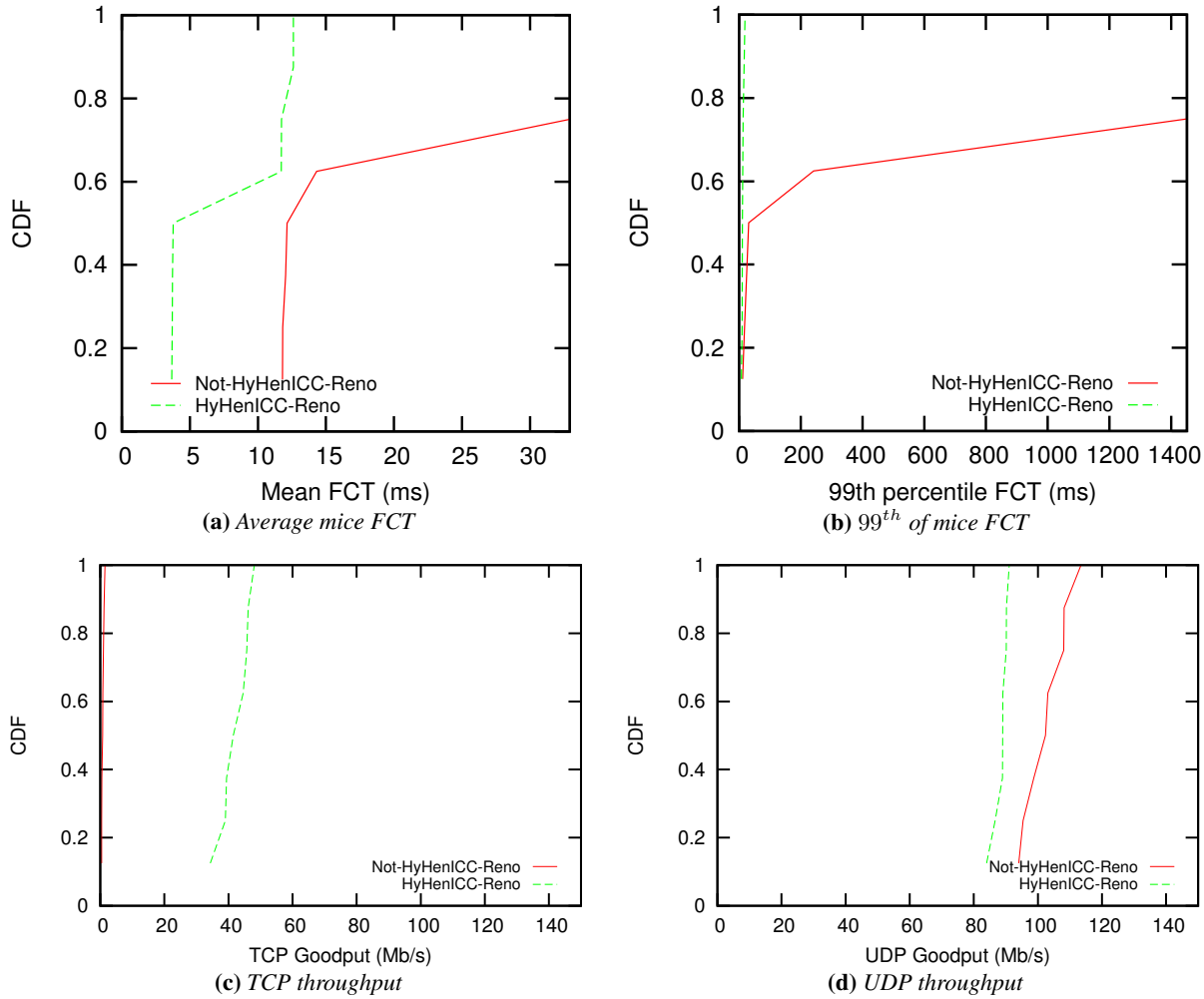
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 10: *Comparison of mice FCT and elephant goodput for HyGenICC in 16 sender scenario where UDP starts at t=5 and ends at t=25*



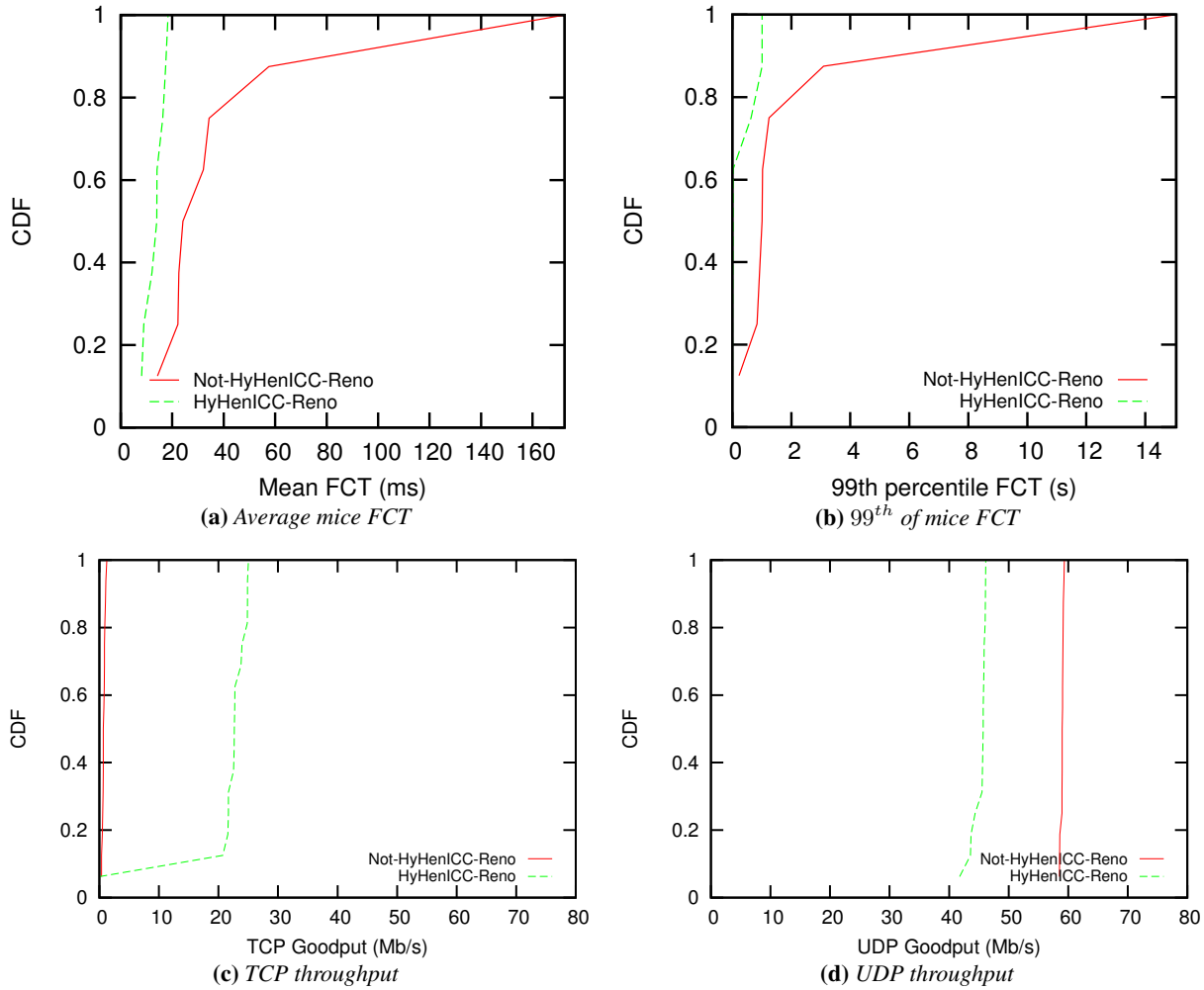
(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 11: *Comparison of mice FCT and elephant goodput for HyGenICC in 8 sender scenario where TCP starts at t=5 and ends at t=25*



(a) *Average mice FCT*

(b) 99^{th} *of mice FCT*

(c) *TCP throughput*

(d) *UDP throughput*

Figure 12: *Comparison of mice FCT and elephant goodput for HyGenICC in 16 sender scenario where TCP starts at t=5 and ends at t=25*