Mystique: a fine-grained and transparent congestion control enforcement scheme

Mystique: A Fine-grained and Transparent Congestion Control Enforcement Scheme

Yuxiang Zhang, Lin Cui, *Member, IEEE,* Fung Po Tso, *Member, IEEE,*
Quanlong Guan, Weijia Jia, *Senior Member, IEEE,* and Jipeng Zhou

***Abstract*—TCP congestion control is a vital component for the latency of Web services. In practice, a single congestion control mechanism is often used to handle all TCP connections on a Web server, e.g., Cubic for Linux by default. Considering complex and ever-changing networking environment, the default congestion control may not always be the most suitable one. Adjusting congestion control to meet different networking scenarios usually requires modification of TCP stacks on a server. This is difficult, if not impossible, due to various operating system and application configurations on production servers. In this paper, we propose *Mystique*, a light-weight, flexible, and dynamic congestion control switching scheme that allows network or server administrators to deploy any congestion control schemes transparently without modifying existing TCP stacks on servers. We have implemented *Mystique* in Open vSwitch (OVS) and conducted extensive test-bed experiments in both public and private cloud environments. Experiment results have demonstrated that *Mystique* is able to effectively adapt to varying network conditions, and can always employ the most suitable congestion control for each TCP connection. More specifically, *Mystique* can significantly reduce latency by 18.13% on average when compared with individual congestion controls.**

***Index Terms*—Web Service, TCP, Congestion Control, Transfer Completion Time**

I. Introduction

Recent years have seen many Web applications moved into cloud datacenters to take advantage of the economy of scale. As the Web applications are becoming more interactive, service providers and users have become far more sensitive to network performance. This is because any increase in network latency always hurt experience and hence providers' revenue. For example, Google quantifies that an additional 400ms latency in searches leads to 0.7% fewer searches per user [1] and Amazon estimates that every 100ms increase in latency cuts profit by 1% [2]. Hence, reducing latency, especially the latency between datacenters and users, is of profound importance for providers.

To reduce network latency, administrators (or operators) opt to use network appliances such as TCP proxies and WAN optimizers [3] [4]. However, these appliances usually have fixed capacity and thus are challenged with scalability issues when faced with increasing traffic volume [3] [5]. For example, TCP proxies split a TCP connection into several sub-connections, breaking the TCP end-to-end semantics. This behavior potentially violates the sequential processing pipelines which applications probably receive the acknowledgement for still transmitting packets [6] [7]. Similarly, WAN optimizers introduce additional data transfer complexity. They compress data at senders for faster transmission. This means additional decompression appliances are required in ISPs or other corresponding places [6].

With these limitations, many researchers chose to intrinsically tackle network latency from TCP congestion control for improving service quality since almost all Web services rely on TCP for service delivery. As a result, a number of congestion control (CC) algorithms have been proposed, including Reno [8], Cubic [9] and BBR [10]. Nevertheless, our extensive evaluations have shown that none of the algorithms can constantly outperform one another in all scenarios (see Section II). In fact, they only reach their peak performance when some specific loss ratio and network delay[1] conditions are met, and degrade dramatically when these change. The degradation of performance caused by inappropriate CCs can lead to decreasing throughput and increasing latency.

On the other hand, many Web servers[2] in cloud datacenters have different operating systems and configurations, e.g., Linux or Windows server with different kernel versions and various default CCs. Considering such vast diversity and quantities of Web servers, adjusting congestion controls (e.g., deploying new advanced CCs) is a difficult, if not impossible, task for administrators [7] [11]. Worse still, administrators sometimes are simply not allowed to modify servers' network stacks directly due to security and SLA constraints.

Motivated by above challenges, we ask: *Can we design a transparent congestion control switching scheme that can always employ the most suitable congestion control for each TCP connection, adapting to network diversities and dynamics, without modifying TCP stacks of Web servers?*

In this paper, inspired by the works in [7] and [11], we propose *Mystique*, a resilient congestion control enforcement

Yuxiang Zhang and Lin Cui are with the Department of Computer Science, Jinan University, Guangzhou, China and the State Key Laboratory of Internet of Things for Smart City, FST, University of Macau, Macau SAR, China, Email: samuelzyx0924@gmail.com, tcuilin@jnu.edu.cn

Fung Po Tso is with the Department of Computer Science, Loughborough University. Email: p.tso@lboro.ac.uk.

Quanlong Guan and Jipeng Zhou are with the Department of Computer Science, Jinan University, Guangzhou, China. Email: gql@jnu.edu.cn, tjpzhou@jnu.edu.cn.

Weijia Jia is with the State Key Laboratory of Internet of Things for Smart City, FST, University of Macau, Macau SAR, China. Email: jiawj@umac.mo.

Corresponding authors: Lin Cui and Weijia Jia

[1]In this paper, latency represents the period of time starting from when service is requested lasting until responses are completely received, while delay means the time taken for a packet to be transmitted across a network from source to destination and RTT indicates the length of time from the packet sent to network to the corresponding acknowledgement arrived.

[2]Those Web servers can be either physical servers or VMs in cloud datacenters. For consistency, we use "Web server" to refer both cases.

without modifying TCP stacks on servers. Advanced TCP congestion controls can be easily implemented by using APIs provided by *Mystique*. *Mystique* can effectively adapt to network conditions and dynamically employ the most suitable congestion control for each connection according to rules specified by administrators.

In summary, the main contributions of this paper are three-fold [12]:

1) We conduct extensive studies to show that no single congestion control suits all network conditions. We also provide a recommendation of the most suitable congestion control for certain network conditions.
2) We propose *Mystique* and present its detailed design working examples. *Mystique* is comprised of a State Module that monitors flow states and a Enforce Module that transparently force Web servers to comply with its decision.
3) We present the prototype implementation of *Mystique* on top of Open vSwitch (OVS). Extensive experiment results show that *Mystique* works effectively, reducing latency by 18.13% on average compared to other schemes.

The remainder of this paper is organized as follows. We present our motivations in Section II. Then we describe the design of *Mystique* in Section III, followed by implementation details in Section IV. The evaluation of *Mystique* in production datacenter environments and Mininet [13] based simulation is presented in Section V. Related works are surveyed in Section VI. Finally we conclude the paper in Section VII.

II. Background and Motivations

A. Background

Latency for Web service is closely linked to revenue and profit. Large latency degrades quality of service (QoS), resulting in poor customer experience and hence revenue loss [2] [14]. In light of this, it is always in service providers' primary interest to minimize their network latency. Many service providers use network functions such as TCP proxies and WAN optimizers for reducing latency [3] [4]. However, their scalability is of a great challenge, while TCP proxies go against TCP end-to-end semantics and WAN optimizers add additional compression and decompression complexity.

On the other hand, congestion control is known to have significant impact on network performance. As a result, many congestion control algorithms have been proposed and studied [10] [15] [16] [17] [18]. Some of these schemes are delay-based, e.g., Vegas and Hybla, which measure the difference between expected and actual RTT, and recognize the difference exceeding a threshold as the congestion signal. Whereas, some CCs, e.g., Reno and Cubic, are loss-based proposals, which consider packet loss as the congestion signal. Furthermore, Illinois is a compound scheme which treats both loss and RTT difference as the congestion signal. Although above schemes use varied congestion signals, they all follow the AIMD (Additive Increase Multiplicative Decrease) rule. On the contrary, BBR constantly estimates BDP (bandwidth delay product) and adjusts its sending rate based on the estimation.

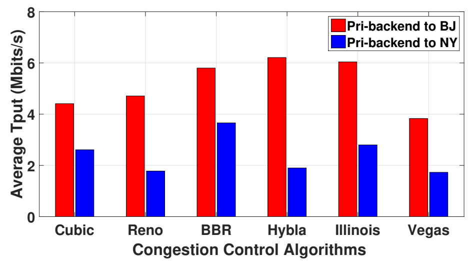


Fig. 1. Performance comparison with different congestion controls

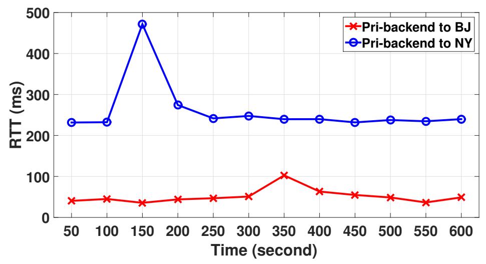


Fig. 2. RTT measured from Pri-backend to BJ and NY respectively

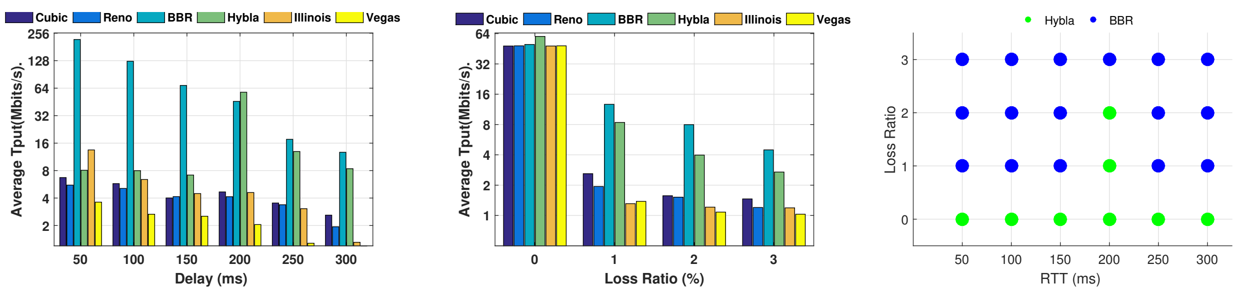
However, recent investigations reveal that the performance of CCs depends on instantaneous network conditions [18] [19] [20]. In the meantime, CCs are usually determined by TCP stack on a server or changed on per-socket basis determined by source code implementation of applications[3]. Furthermore, service providers usually deploy a range of different Web servers for fault tolerance and security. These servers run different operating systems (e.g., Linux and Windows) and are configured with different congestion controls. On the other hand, in multi-tenants cloud datacenters, network operators may be prohibited from upgrading TCP stacks on particular Web servers for security issues. Clearly, manual fine-tuning congestion control on every server for each TCP connection is impractical. It is necessary to provide a mechanism which allows drop-in replacement of TCP stacks for per-flow (as oppose to per-server) level of granularity, giving network administrator sufficient control of network resource whilst improving latency for Web services.

B. Problem Exploration

To better understand the performance diversity of different congestion controls under varying scenarios, we carried out several testbed and simulation experiments to investigate such performance diversity. First, we triggered 50MB file transfers from a Web server named Pri-backend (in Guangzhou) to two clients in Beijing (BJ) and New York (NY)[4], respectively. We measured the performance of six different CCs that are available in Linux kernel. The throughput results in Figure 1 show that *performance of different congestion controls varies under different network conditions*. For example, Reno performs better than Cubic for the data transfer to BJ, but performs worse in the scenario transmitting to NY. This is because the network conditions are very different for the two connections as reflected in their RTTs. Figure 2 shows even though both

[3]CC algorithms can be changed through system call *setsockopt* in Linux.

[4]Pri-backend is a Web server deployed in our campus datacenter. BJ and NY are two client machines in AWS cloud datacenter. The detailed information and configuration about those machines can are in Table IV and Table V.



(a) Throughput of varied congestion control under the scenarios of loss=1%

(b) Throughput of varied congestion control under the scenarios with delay = 300ms

(c) The most suitable congestion control

Fig. 3. The performance of different congestion controls under varied scenarios

BJ and NY connect to the same server Pri-backend, Pri-backend → NY has almost 10X larger RTT than that of Pri-backend → BJ. Particularly, RTTs change dynamically for both connections. *This demonstrates that the most suitable and efficient CC can be changed with times even for a single TCP connection.*

Next, we conducted Mininet [13] based experiments to quantify the performance variation of CCs under different network conditions. The network contains two servers connected to two switches, respectively, in a line topology. Bandwidth of all links are set to be 1Gbps. Long-lived flows generated by *iPerf* [21] are used to evaluate performance of different CCs.

1) Observation with varying delay: Delay variation is common on the Internet since routes for packets are changing and end-hosts adopt delayed ACK technique. Here, the loss ratio of the link between the pair of servers is set to be 1% with delay ranging from 50*ms* to 300*ms*. Results in Figure 3(a) show that BBR has the best performance in most scenarios, while Hybla [22] outperforms all other schemes at 200*ms*. BBR constantly estimates the network capacity and adjust its sending rate based on the measured BDP [10]. Therefore, packet loss would not affect its sending rate which keeps its substantial performance [23]. Hybla uses relatively large congestion window to counteract the effects of long RTT and packet losses so that is able to assure a satisfactory transmission rate [22]. And the other schemes, e.g., Illinois and Vegas, are slightly affected by the RTT since the feedbacks of these CCs depend on the network delay, which reduces the congestion window growth rate and results in performance degradation [22] [24].

Observation 1. *The performance of TCP congestion control varies under different network delays with constant loss ratio.*

2) Observation with varying loss ratio: In addition to delay variation, loss ratio also varies in a real network. To quantify its impact, we set the delay between the pair of servers to be 300ms with loss ratio ranging from 0% to 3%. Results in Figure 3(b) show that all TCP variants have different performance under scenarios with different loss ratio. Specially, Figure 3(b) shows that Hybla can give a satisfactory performance in an ideal channel since it removes its dependence on RTT [22]. BBR performs the best due to its BDP based mechanism. And other schemes reduce their congestion windows when packet loss is detected, and eventually impair their performance.

Observation 2. *The performance of TCP congestion control varies under different loss ratios with constant delay.*

TABLE I
CONGESTION CONTROLS RECOMMENDATION

CC	Target scenarios	Characteristics
Hybla	No packet loss	Remove the performance dependence on RTT; Effective in counteracting the effects of long RTT and losses
Illinois	RTT $\leq$ 50ms & loss exists	Recognize loss as congestion signal, adjust sending rate based on RTT variance
BBR	RTT $\geq$ 50ms & loss exists	Compute sending rate based on estimated BDP, not sensitive to packet loss

3) Observation on the most suitable congestion control: The best performing congestion control is also obtained (indicated with different colors in Figure 3(c)) for all scenarios with delay ranging from 50ms to 300ms and loss ratio ranging from 0% to 3% respectively. Hybla performs well in idle environment and BBR is good at handling loss albeit not being a loss-based scheme. This further confirms our analysis above. Particularly, results demonstrate that Hybla could achieve satisfactory performance in ideal channel and obtain comparable performance in long RTT and lossy scenario [9] [22]. Further, since BBR focuses on estimating maximal available network capacity rather than the packet loss, it is very suitable for deployment in lossy network [10] [23].

Observation 3. *No single congestion control suits all network conditions.*

Remarks: Based on above observations and analysis of the behavior of each CC, we summarize a recommendation in Table I. Generally, BBR is suitable for the scenario that packet losses exist while Hybla can be chosen for ideal network with no packet loss. However, according to Figure 1, Illinois outperforms BBR when RTT is less than 50ms and loss exists. This is because the decrease factor of Illinois keeps small when the RTT is relatively small. Therefore, Illinois would not degrade its performance in a scenario which has packet loss and short RTT. Hence, we choose Illinois as the most suitable congestion control when RTT is less than 50ms and loss exists.

C. Design Goals

We aim to design a transparent platform allowing network administrators to dynamically deploy and adjust congestion controls in a fine-grained granularity without modifying TCP

Algorithm 1 *Mystique* Overall Algorithm

```
 1: for each incoming packet do
 2:   if belongs to client → server then
 3:     if SYN packet then
 4:       Create flow entry
 5:     end if
 6:   else if FIN packet then
 7:     Remove flow entry
 8:   else if ACK packet then
 9:     Update congestion states # e.g., max_bw, min_rtt
10:     if Loss signal then
11:       Retransmit loss packet
12:     end if
13:     CC ← CC_switching(states)
14:     Compute new cwnd based on CC
15:     Enforce cwnd
16:   else
17:     if DATA packet then    # belongs to server→client
18:       Record packet states  # e.g., una, nxt
19:       Store packet
20:     end if
21:   end if
22: end for
```

stacks of Web servers. This goal can be translated into three properties:

1) **Transparency**. Our scheme should allow network administrators or operators to enforce advanced congestion controls without touching TCP stacks of servers. Enforcement of our scheme should be transparent to both Web servers and clients. This is important in untrusted public cloud environments or in cases where servers cannot be updated due to security issues [11] [25].
2) **Flexibility**. Our scheme should allow different congestion controls to be applied on a per-flow basis and select the most suitable congestion control according to current network status. This is useful since each congestion control has its own deficiency and suitable scenarios. Allowing adjusting CCs on a per-flow basis can enhance flexibility and performance.
3) **Light-weight**. While the entire TCP stack may seem complicated and prone to high overhead, the congestion control is relatively light-weight to implement. Our scheme should consume less resource as possible and be a feasible solution for real world network.

We will show how *Mystique* achieves the first two objectives in the next section. To achieve the last goal, *Mystique* needs to apply specific techniques to handle packets processing, e.g., Read-Copy-Update hash tables and NIC offloading, which are discussed in Section IV.

III. *Mystique* Design

A. Design Overview

Mystique is a resilient congestion control enforcement platform without modification on Web servers. *Mystique* monitors all flows in/out Web servers and obtains corresponding congestion states. With these states, *Mystique* switches and deploys the most suitable congestion control by modifying the receive window.

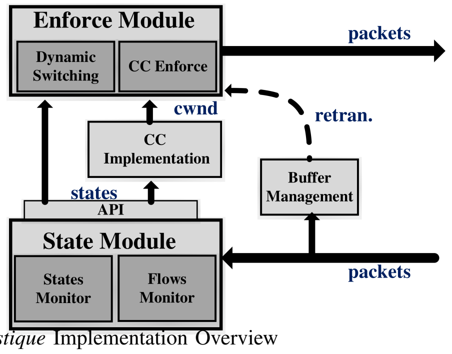


Fig. 4. *Mystique* Implementation Overview

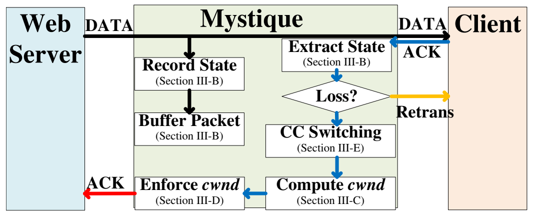

Web Server
DATA
Mystique
DATA
Client
Extract State (Section III-B)
ACK
Record State (Section III-B)
Loss?
Retrans
Buffer Packet (Section III-B)
CC Switching (Section III-E)
ACK
Enforce cwnd (Section III-D)
Compute cwnd (Section III-C)

Fig. 5. *Mystique*'s processes of Data packets and ACK packets

In this section, without loss of generality, we assume that Web servers run loss-based congestion control which is popular in today's operating systems. This is because in Linux kernels 2.6 and above, as well as Windows 10, Windows Server 2016 and later versions, Cubic is the default CC. For Unix OS, Cubic and Reno are usually configured as the default CC [26]. Thus, loss-based CCs, e.g., Cubic and Reno, are supposed to be run on most Web servers since Unix-like (including Linux) and Windows Server are dominant Web server operating systems [27].

Figure 4 shows the high-level overview of *Mystique*'s implementation based on Open vSwitch (OVS) [28]. The overall operations of *Mystique* and processing on TCP data and acknowledgement packets are shown in Algorithm 1 and Figure 5 respectively. When a data packet from a Web server is pushed down to network stacks, it is intercepted by *Mystique*.[5] The sequence number and sending timestamp are recorded. The packet is sent to the client immediately when a copy is buffered by the Buffer Management module. When ACKs from clients reach *Mystique*, *State Monitor* extracts congestion states from each ACK and releases corresponding buffer simultaneously (Section III-B). Upon receiving ACKs, *Enforce Module* pulls congestion states from *State Module* via corresponding APIs and selects a CC which is used to compute *cwnd* (Section III-E)[6]. *Enforce Module* computes *cwnd* by calling the CCs implemented in *Mystique* and modifies *rwnd* if needed for enforcing *cwnd* (Section III-C & Section III-D). Further, when congestion signals are detected, *Mystique* re-computes the *cwnd*. In the meantime, *Mystique* retransmits lost packets in order to mitigate the adverse effect on both Web server's congestion window and throughput (Section III-D).

[5]In current implementation of *Mystique*, packets are intercepted by OVS's *ovs_dp_process_packet* module.

[6]In this section, *cwnd* stands for congestion window, while *rwnd* represents receive window in acknowledge packet.

TABLE II
SOME STATES MONITORED BY *Mystique*

States	Descriptions
una	The first packet's sequence number not acked
nxt	The sequence number of next packet
c_rtt	Current value of rtt
c_bw	Current value of bandwidth
min_rtt	Minimum value of rtt
max_bw	Maximum value of bandwidth
t_step	The value of *min_rtt* adjustment
bw_step	The value of *max_bw* adjustment

TABLE III
SOME APIS PROVIDED BY *Mystique*

Methods	Descriptions
getCRTT()	Obtain state c_rtt's value
getCBW()	Get state c_bw's value
getMinRTT()	Get state min_rtt's value
getMaxBW()	Get state max_bw's value
setPeriod()	Set parameter *period*'s value
setTstep()	Set parameter t_step's value
setBWstep()	Set parameter bw_step's value
isLoss()	Offer loss feedback
setCwnd()	Set new congestion window

Detailed designs are elaborated in the following sections. Please note, congestion states (Table II) can be delivered to canonical TCP congestion controls through APIs provided by *Mystique* (Table III), and then used by varied congestion controls to compute appropriate congestion window *cwnd*. The initial *cwnd* is set to a default value of 10 MSS [11].

B. Monitoring Congestion Control States

One fundamental operation of *Mystique* is to obtain packet-level congestion states, which are then used as inputs for each congestion control. We note that *Mystique* should be implemented in network locations where all traffic would pass through so that flows/packets/congestion states can be retrieved (see Section III-F).

TCP sequence number can be obtained directly from packets. With the sequence number states, *Mystique* can monitor flows' packet loss, sending rate and so on. Similar to [11], *Mystique* uses variable *una* to record the sequence number of the first packet (denoted as *seq*) which has been sent, but not yet acknowledged. Variable *nxt* is used to record the *seq* of the next packet to be sent (but not yet received from the Web server). Packets between *una* and *nxt* are being transmitted (inflight packets). Each ACK contains an acknowledgement number (denoted as *acknum*) in TCP header field. The *acknum* represents the packets whose *seq* is less than or equal to *acknum* have been confirmed received by receiver. Thus, variable *una* would be updated when the *acknum* is larger than current *una* value, since relative packets (bytes) have been acknowledged. When a packet is received from Web servers, *nxt* would be updated if the new packet's *seq* is larger than current *nxt* value, because *Mystique* has received the expected packet.

With the valuables *una* and *nxt*, detecting packet loss is relatively simple. There are two possible packet loss signals: three duplicate ACKs and timeout [7] [11]. For the first signal, *Mystique* adopts a local duplicate ACK counter *dupack* to sense it. When an ACK packet arrives, if the *acknum* is less than or equal to the value of *una*, it means this ACK is acknowledging stale data packets, and the local *dupack* counter would be updated. When *dupack* counts to 3, i.e., three duplicate ACKs have been received, a packet loss has been detected [29]. For the second signal, timeouts can be inferred when *una* is less than *nxt* and an inactivity timer fires.

As shown in Table II, *Mystique* monitors some congestion states, such as maximal available bandwidth for current connection *max_bw*, minimal round trip time for current connection *min_rtt*, round trip time measured by current received ACK *c_rtt* and measured bandwidth obtain by current received ACK *c_bw*. When a new TCP connection is detected, states *min_rtt* and *max_bw* are initialized to ∞ and 0 respectively. When ACK arrives, *c_rtt* can be updated by computing the difference between ACK and corresponding data packets arriving timestamps. In the meantime, the size of acknowledged bytes (for the ease of description, we denoted this as *acked*) can be obtained by *acknum* - *una*, which represents the acknowledged bytes by using the acknowledgement number to subtract the first *seq* that has not been received. Hence, current available bandwidth *c_bw* could be computed as *acked*/*c_rtt*. Further, if *c_rtt* is less than *min_rtt* which means current observed round trip time is smaller than the historically minimal RTT *min_rtt*, the *min_rtt* would be renewed as *c_rtt*. Similarly, *max_bw* is updated when *c_bw* is larger than *max_bw*, this means current observed available bandwidth *c_bw* is larger than the maximal bandwidth *max_bw*.

C. Implementing Congestion Control

Next, we use the implementation of BBR as an example to elaborate how to implement congestion controls based on *Mystique*. Other congestion controls can be implemented in the similar way.

First, we briefly introduce BBR congestion control mechanism [10]. A TCP connection has exactly one slowest link or bottleneck in each direction and such bottlenecks affect TCP performance. The bottleneck determines the connection's maximum sending rate and it is where persistent queues form. Therefore, there are two physical constraints, round trip propagation time (denoted as *RTprop*) and bottleneck bandwidth (denoted as *BtlBw*), these bound transport performance. A connection can have the highest throughput and lowest delay when the total data in flight is equal to the BDP (= *BtlBw* × *RTprop*). This guarantees that the bottleneck can run at 100 percent utilization and there is enough data to prevent bottleneck starvation but not overfill the pipe. Therefore, in BBR mechanism, sender needs to continuously measure the *BtlBw* and *RTprop* and control the total inflight data equal to the BDP [10] [23]. Since the bottleneck bandwidth can be approximately equal to the maximum available bandwidth, *Mystique* recognizes the *max_bw* as the *BtlBw*. Similarly, *min_rtt* can be used as the *RTprop*.

Therefore, if the estimations of *BtlBw* and *RTprop* are obtained, enforcing BBR's mechanism is straightforward. Congestion control information is extracted from data and ACK packets. Connection tracking variables, *max_bw* and *min_rtt*, are updated based on the ACKs (details refer to Section III-B). Hence, *cwnd* can be computed based on *min_rtt* (obtained by *getMinRTT()*) and *max_bw* (obtained by *getMaxBW()*).

In Linux kernel's implementation, BBR uses special modes to calibrate the estimation of network capacity, e.g., probe_bw and probe_rtt. Since *Mystique locates outside the host, it is impossible for Mystique to set the pacing_gain directly (a parameter for probing more bandwidth). Mystique* can not directly control flows' sending rate, especially can not force server to increase its *cwnd* to probe for more bandwidth. Therefore, we adopt a trade-off to approximately estimate the network capacity and to adapt to the network dynamics.

Mystique adopts a time unit variable *period*, which can be set via *setPeriod()*, to update *min_rtt* and *max_bw* cyclically. In every *period*, states *min_rtt* and *max_bw* would be updated as follow: *min_rtt = min_rtt + t_step, max_bw = max_bw - bw_step*. Here, *t_step* and *bw_step* are adjustment constant for recalibrating *min_rtt* and *max_bw* respectively. If the new *min_rtt* is larger than the RTT under current network condition, *min_rtt* would be updated when the next ACK arrives. For example, we assume the *min_rtt* at time T_k is 2 seconds, while the actual network round trip time has been increased to 2.5 seconds at time $T_k + period$. If the above trade-off is not adopted, the change of RTT could not be sensed by *Mystique*. On the other hand, if above trade-off is adopted and the *t_step* is preset as 1 second, *Mystique* can detect that the *min_rtt* has reached 2.5 seconds as follows: (1) *Mystique* updates the *min_rtt* to 3 seconds at time $T_k + period$. (2) Upon receiving ACKs, *Mystique* knows that the *c_rtt* reaches 2.5 seconds. (3) Since *c_rtt*(2.5 seconds) is less than *min_rtt*(3 seconds), the *min_rtt* would be updated as 2.5 seconds. Hence, *Mystique* can approach to the actual minimal RTT with such operations step by step.

Also variable *max_bw* can be updated based on similar trade-off mechanism. In current *Mystique*'s implementation, *period* is preset to be 5 seconds, *t_step* and *bw_step* are configured as *min_rtt*/10 and 1 MSS (via methods *setTstep()* and *setBwstep()*) respectively [30].

D. Enforcing Congestion Control

Once the *cwnd* is ready, the next step is to ensure that the Web server's sending rate can adhere to it. TCP provides built-in functionality that can be exploited for *Mystique*. Specifically, TCP's flow control allows a receiver to advertise the amount of data it is willing to process via a receive window *rwnd* [11] [29]. *Mystique* will overwrite the *rwnd* with its computed *cwnd* (done by *setCwnd()*) for restricting amount of inflight packets. In order to preserve TCP semantics, this value is overwritten only when it is smaller than the packets' original *rwnd*, i.e., *rwnd* = min(*cwnd*, *rwnd*). Such scheme restricts amount of packets sent from server to clients while preserving TCP semantics.

Ensuring a Web server's flow adheres to receive window is relatively simple. Web servers with unaltered TCP stacks will naturally follow our enforcement scheme because the stacks will simply follow the standard. The Web server's flow then uses *min(cwnd, rwnd)* to limit how many packets it can send since Web server would constrain single connection's inflight packets as the smaller value of congestion window and receive window. Further, in order to be compatible with TCP receive window scaling, *Mystique* monitors handshakes to obtain this value and *cwnd* are adjusted accordingly.

When it comes to enforcing *cwnd*, there are two possible situations for *Mystique*. For the ease of description, we use general example to illustrate the conditions rather than specified congestion controls, in order to give a general idea of this challenge. (a) When *cwnd* in *Mystique* is smaller than the congestion window in Web server, modifying *rwnd* can limit the sending rate effectively. For example, if the *cwnd* in *Mystique* is 20 and the congestion window of Web server is 30, modifying *rwnd* can throttle the connection's sending rate since the *rwnd* is less than the congestion window in Web server and sending rate equals to *min(cwnd, rwnd)*. In this condition, *Mystique* takes the control of server's congestion control which achieves *Mystique*'s goal. (b) Whereas if *Mystique*'s *cwnd* is the larger one, modifying *rwnd* may not be an effective method. For example, if the *cwnd* in *Mystique* is 20 and the *cwnd* of Web server is 10, throttling the connection's inflight packets by modifying *rwnd* would be unavailing since the *rwnd* is larger than *cwnd* of Web server and this *cwnd* is in effect to constrain the sending rate. Therefore, Web server's congestion window have to stay at a high value to allow *Mystique* enforcing its congestion window.

Since we have assumed that Web server runs on loss based congestion control which is sensitive to packet loss. Here, we take Cubic as an example [9]. Cubic follows the AIMD rule. If a loss event occurs, Cubic performs a multiplicative decrease of congestion window by a factor of β where β is a decrease constant and the window just before the reduction is set to the parameter W_{max}. Otherwise, Cubic increases the window by using a window growth function which is set to have its plateau at W_{max}. The window growth function of Cubic uses the following function: $W(t) = C(t - K)^3 + W_{max}$, Where C is a Cubic parameter, t is the elapsed time form the last window reduction, and K is the time period that the above function takes to increase W to W_{max} when there is no further loss event and is calculated by using the following equation: $\sqrt[3]{\frac{W_{max}\beta}{C}}$. If *Mystique* wants to prevent condition (b) occurring, avoiding Web server receives loss signals which lead to window shrinkage is of importance. Therefore, *Mystique* prevents any congestion signals (e.g., ECN feedback and three duplicated ACKs) are transmitted to Web server in order to prevent decreasing the *cwnd* of Web server. As Web servers would not receive these congestion signals, their congestion window will not be affected. Besides, in addition to above methods, *Mystique* adopts packets buffering and retransmits the corresponding packet when the loss signals are detected. With continuous data transmission, Web server's congestion window would arrive at a high level gradually.

If Web servers do not run loss-based congestion control, e.g., running on delay-based CC, we can adopt Fake-ACK (FACK) mechanism [7] [11] to get the control of the connections. *Mystique* can send dedicated FACK packet to Web servers in order to keep the RTT states measured by the server relatively low, so that the Web server would keep its sending rate at a high level.

E. Dynamic Congestion Control Switching

Mystique always tries to assign suitable congestion controls on a per-flow basis according to networking conditions. The most suitable congestion control that would be employed can be either/both determined by current network congestion states or administrator defined switching logics. Algorithm 2 shows a simple example of congestion control switching logic. In the following, we recognize BBR, Hybla and Illinois can perform well under most network environment according to Table I.

Since *Mystique* dynamically employs the most suitable congestion control according to connection conditions, CC switching can happen even for a single TCP connection. It is essential to ensure smoothness when performing CC switching. Some congestion controls (e.g., BBR & Hybla) can compute their congestion window based on the measured states. Others (e.g., Illinois) may need other parameters to compute congestion window. Though these parameters can become useless after switching to other congestion controls, *Mystique* updates all these parameters continuously to prevent any performance degradation after switching back to corresponding mechanisms later.

Enforcement Example: Next is a step-by-step example to show how *Mystique* works with Algorithm 2. Particularly, we will show how to operate a TCP Cubic Web server via *Mystique* with the BBR congestion control scheme. The network scenario is assumed to have RTT larger than 50ms and packet loss exists (see Table I). For the ease of presentation, the value of *cwnd* on Web server is assumed to have arrived at a high level, otherwise it would be increased to such high value gradually (see Section III-D). The enforcement steps are as follows:

1) *Mystique* monitors all data and ACK packets. When an ACK packet arrives, *Mysitque* extracts and updates the congestion states, e.g., *min_rtt* and *max_bw*.
2) According to Algorithm 2, *Mystique* employs BBR as the most suitable congestion control for this connection.
3) *Mystique* will call procedure of BBR to caculate *cwnd* with all required states. In this case, $cwnd = min\_rtt \times max\_bw$. If *cwnd* is smaller than the original *rwnd* of this ACK, *Mystique* will overwrite the *rwnd* with *cwnd*.
4) Since the congestion window of Web server has a high value, the sending rate of Web server is restricted by the *rwnd* of this ACK.

Hence, Mystique can take the control of Web server's congestion control.

Finally, based on *Mystique*, administrators can define more complex switching logic using more metrics, e.g., loss ratio, variation of RTT. Besides, a more uniform schedule can also be used to mitigate side effects of some CCs [11]. Due to space limitation, other switching schemes are not elaborated here.

F. Available Deployment Locations

Mystique can be easily deployed in three possible locations in cloud datacenters:

- *VMs*: Deploying *Mystique* in VMs allows network administrators to setup new *Mystique* servers or release old ones dynamically for load-balancing. However, such scheme requires routers/switches redirecting desired traffic to *Mystique* servers, which is not difficult specially for SDN-enabled environment.
- *Hypervisors*: Since *Mystique* currently can be implemented in OVS which is compatible with most hypervisors, hypervisors of physical servers would be a good choice of employing *Mystique*. Such scheme allows *Mystique* to be easily scaled with numbers of servers in datacenters. It also minimizes the latency between *Mystique* and Web servers, i.e., VMs. Furthermore, no route redirection is required in this case. However, the flexibility and scalability are limited considering migrations of VMs or situation that VMs on a server are heavy loaded.
- *Routers/Switches*: Routers/switches can inherently monitoring all incoming traffic, making *Mystique* can easily enforce congestion control without route redirection. However, traffic sent through a router/switch is determined by the routing algorithm of datacenters, and it is difficult to perform load balancing. And heavy traffic may also overwhelm capacity of routers/switches.

Each deployment choice suits for different requirements and scenarios. In practice, combination of these three deployment choices above can be considered.

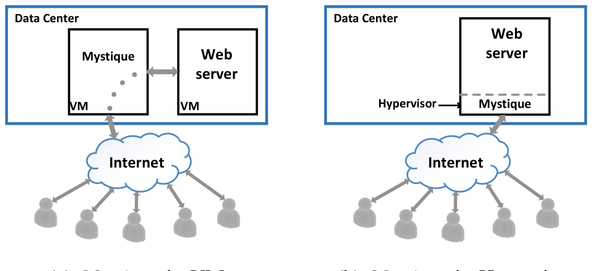
Algorithm 2 Congestion control switching logic example

```
Input: Congestion states, e.g, loss, RTT
1: if no loss then
2:     Return Hybla
3: else if RTT < $50ms$ then
4:     Retrun Illinois
5: else
6:     Return BBR
7: end if
```

IV. Implementation

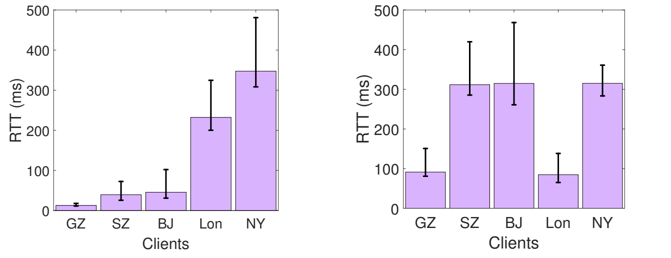
We have implemented a prototype of *Mystique* on Open vSwitch (OVS) v2.7.0 [31]. About 1400 lines of code are added to implement *Mystique*'s basic functions, including tracking congestion states, managing buffer and switching logic. As shown in Figure 4, *Mystique* is mainly comprised of State Modules, Enforce Modules and Buffer Management. The only function of Buffer Management is to buffer all packets from Web server to clients and to retransmit them if needed. *skb_clone*() is used for packet buffering to prevent deep-copy of data. Besides, the State Module is responsible for monitoring congestion states while the Enforce Module is used to implement and enforce both congestion controls and administrator-defined switching logics.

State Module: State Module monitors every incoming or outgoing flows and obtains congestion states. Flows are hashed on a 5-tuple (IP addresses, ports and protocol) to obtain a flow's entry for maintaining the congestion control states mentioned in Section III-B. SYN packets are used to create flow entries while FIN packets are used to remove flows entries. Other



(a) *Mystique* in VM (b) *Mystique* in Hypervisor

Fig. 6. Test-bed Topology.



(a) The RTT between clients and servers in private cloud (b) The RTT between clients and servers in public cloud

Fig. 8. The RTT(ms) between clients to private/public Cloud.

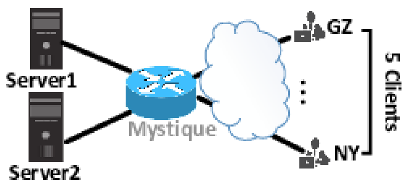


Fig. 7. Simulation Topology

TCP packets, such as data and ACKs, trigger updates of flow entries. Since there are many table lookup and update operations, *Read-Copy-Update* (RCU) hash tables are used to enable efficient lookups. Additionally, *spinlocks* [32] are used on each flow entry in order to allow for multiple flow entries to be updated simultaneously. Multi-threading technique is used for updating congestion states and parameters. When an ACK arrives, a new thread is created to free up according acknowledging bytes and update congestion states and variables. When a data packet arrives, new thread is used for recording the state variables of this packet. *Mystique* provides an event driven programming model and the congestion states could be obtained by calling the corresponding APIs.

Enforce Module: Enforce Module enforces administrator-defined switching logics and corresponding congestion controls. Submodule *Dynamic Switching* and *CC Implementations* would use the according APIs provided by State Module to pull current TCP connection states for dynamically employing the most suitable CCs or computing *cwnd* adhered to according mechanisms. Every time a new ACK from client arrives, *cwnd* is computed with the congestion states obtained from State Module. If *cwnd* is smaller than ACK's original *rwnd*, *Mystique* rewrites *rwnd* with computed *cwnd*. With the help of NIC offloading features such as TSO (TCP Segment Offloading), some necessary operations can be offloaded to NIC in order to reduce computational overhead, e.g., TCP checksum calculation [11].

V. Evaluation

This section quantifies the effects of *Mystique* and evaluates the performance of *Mystique* through extensive experiments.

A. Experiment Setup

Testbed Setup: The test-bed consists of 14 servers from both production private and public clouds and 5 clients are used for obtaining in-depth understanding. The setup of these machines are summarized in Table IV and Table V. Particularly, names of Web servers reflect their locations (e.g., private or public cloud) and evaluated congestion control schemes. For example, Web server Pri-Cubic represents a server locates in private cloud and runs Cubic as its default CC. Pub-Reno demonstrates that this Web server located at public cloud (i.e., AWS) and Reno is configured as its CC. Further, there are two backend servers (Pri-backend & Pub-backend) which are used for evaluating *Mystique*, e.g., Pri-backend is a backend server in private cloud aiming at evaluating *Mystique* in private cloud environment. Both backend servers run Cubic as their default congestion control.

The private cloud is a campus datacenter located in Jinan University, Guangzhou, China. It contains over 400 Web servers (VMs) running on 293 physical servers. The bandwidth of the datacenter to the Internet is 20Gbps, shared by all servers in the datacenter. And the data rate of NICs on each physical server is 1Gbps, shared by all VMs on the same server. The AWS (public cloud) [33] is also used for deployment and experiments of evaluating the performance of *Mystique*. Our experiments involve clients from five locations described in Table V. These clients experienced different RTT and loss ratio when connecting to both private and public cloud Web servers. Figure 8 depicts the average Round Trip Time of each pair of clients and Web servers with variation. Since all private cloud or public cloud Web servers locate in the same datacenter, we can assume their RTTs to the same client are similar. All of these Web servers (including two Mystique servers) and clients are connected through the Internet. They all run Ubuntu 16.04 with Linux kernel 4.9 and Apache 2.0 as well. Besides, in all Web servers, *tcp_no_metrics_save*, *tcp_sack* and *tcp_low_latency* are set to 1 and the MTU size is configured as 1.5KB [7] [11].

Simulation Setup: Mininet based simulations are used to evaluate *Mystique* implemented on switches/routers. The topology used in simulation is a star topology similar to [7], shown in Figure 7. In order to simulate intra-datacenters background traffic, a long-lived flows from server 1 to server 2 is created. Five clients are used to represent the test-bed clients for consistency. Links are configured according to Figure 8, mirroring the configuration of our testbed.

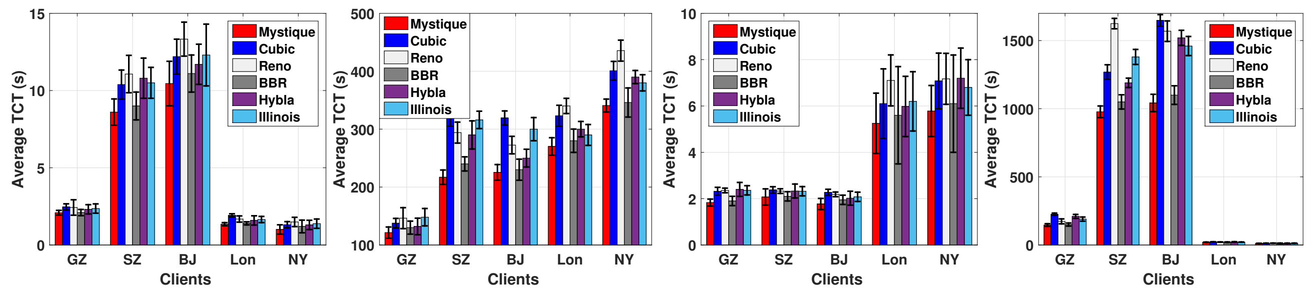
***Mystique* Setup:** Each *Mystique* server is equipped with 2.30 GHz CPU and 4 GB memory. Besides, the *Mystique* server in Private cloud has 1Gbps bandwidth while the one in public cloud has 5Gbps bandwidth. Current version of *Mystique* is implemented in Open vSwitch 2.7.0, which enforces the CC switching logic presented in Algorithm 2. Further, we evaluated *Mystique* by triggering clients to connect to Pri-

TABLE IV
WEB SERVERS INFORMATION IN TEST-BED EXPERIMENT

Machine(s)	Location	Role	CPU	Mem.	Band.	CC
Pri-Cubic	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Cubic
Pri-Reno	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Reno
Pri-BBR	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	BBR
Pri-Hybla	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Hybla
Pri-Illinois	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Illinois
Pri-backend	Guangzhou, China	Private Cloud Server	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Cubic
Pub-Cubic	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Cubic
Pub-Reno	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Reno
Pub-BBR	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	BBR
Pub-Hybla	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Hybla
Pub-Illinois	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Illinois
Pub-backend	Singapore	AWS Server	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Cubic
*Mystique*1	Guangzhou, China	Private Cloud Deployment	Intel(R) E5-2670 @ 2.30GHz	4GB	1Gbps	Alg. 2
*Mystique*2	Singapore	AWS Deployment	Intel(R) E5-2670 @ 2.30GHz	4GB	5Gbps	Alg. 2

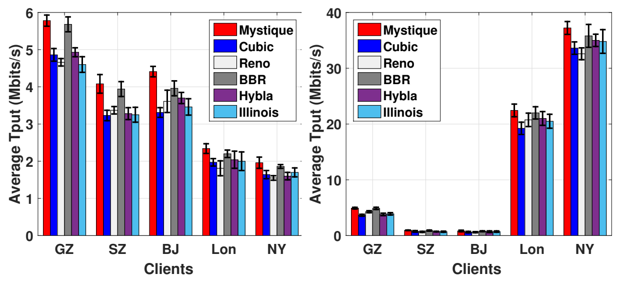
TABLE V
CLIENTS INFORMATION IN TEST-BED EXPERIMENT

Machine	Location	CPU	Mem.	Band.
GZ	Guangzhou, China	Intel i7-4790 @ 3.6GHz	16GB	1Gbps
BJ	Beijing, China	Intel(R) E5-2686 @ 2.30GHz	2GB	5Mbps
SZ	Shenzhen, China	Intel(R) E5-2699 @ 2.40GHz	2GB	5Mbps
NY	New York, US	Intel(R) E5-2670 @ 2.30GHz	4GB	5Mbps
Lon	London, UK	Intel(R) E5-2670 @ 2.30GHz	4GB	5Mbps



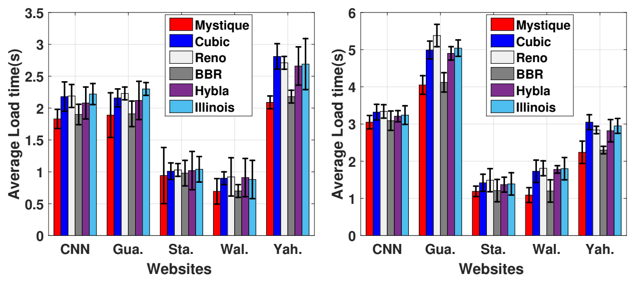
(a) Small file in private cloud (b) Large file in private cloud (c) Small file in public cloud (d) Large file in public cloud

Fig. 9. The average transfer completion time (TCT) for both small file and large file in private cloud and public cloud, when *Mystique* is deployed in VM.



(a) In private cloud (b) In public cloud

Fig. 10. Average throughput (Tput) for both private and public cloud with *Mystique* deployed in VM.



(a) In private cloud (b) In public cloud

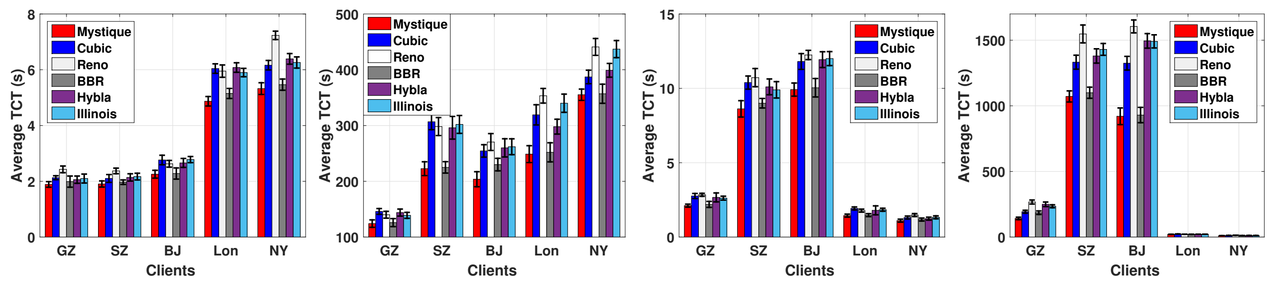
Fig. 11. Average load time for both private and public cloud Websites with *Mystique* deployed in VM.

backend/Pub-backend through *Mystique*. Though such evaluation would add additional one-hop latency, the results show that *Mystique* can obtain impressive performance constantly.

Schemes Compared: To understand *Mystique* performance, we compare *Mystique* with other five congestion controls. Clients connect to these Web servers directly in order to evaluate the performance of these schemes, and the detailed machine information and configuration is in Table IV.

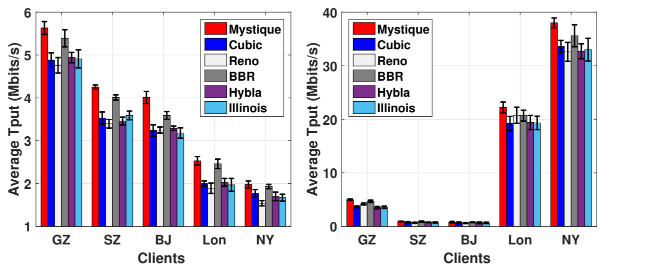
Metrics: We use Transfer Completion Time (TCT) as the primary performance metric. For all Web servers except Mystique1 and Mystique2, we uploaded two files: small file (OpenFlow Switch Specification v1.5.1.pdf [34], 1.2MB) and large file (Linux kernel 4.13 source code.xz [35], 95.9MB). Besides, average throughput and load time are considered as primary metrics for efficiency of data transfers. We also snapshot the homepages of CNN, Guardian (Gua.), Stack Overflow (Sta.), Walmart (Wal.) and Yahoo (Yah.)[7] for evaluat-

[7]The whole homepages are downloaded from their original Website to avoid effects of any dynamic contents in the Webpages.



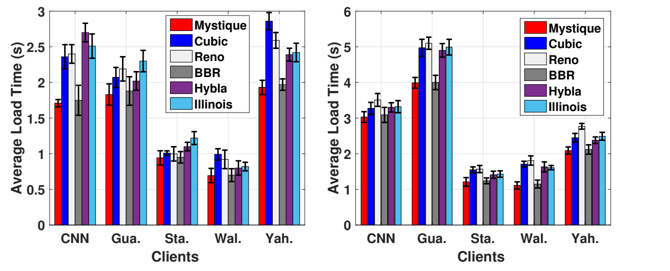
(a) Small file in private cloud (b) Large file in private cloud (c) Small file in public cloud (d) Large file in public cloud

Fig. 12. The average transfer completion time for both small and large files in private and public cloud, when *Mystique* is deployed in hypervisor.



(a) In private cloud (b) In public cloud

Fig. 13. The average throughput (Tput) in private cloud and public cloud, when *Mystique* is deployed in hypervisor.



(a) In private cloud (b) In public cloud

Fig. 14. The average load time in private cloud and public cloud, when *Mystique* is deployed in hypervisor.

ing the performance of *Mystique* on Webpages with elements including html files, js, css, images and so on. Curl [36] is used to measure and record the TCT and throughput and Chrome Devtools [37] on client GZ is used to obtain the load time of the mirrored Websites. Besides, we use *sar* [38] to measure CPU and memory usage. Confidence interval and confidence level are used too. Confidence interval contains a range of potential values of the unknown population parameter and confidence level represents the frequency of confidence intervals that contain the true value of the unknown population parameter [39] [40].

For each scenario, we have conducted a 48-hour data transfer experiment by instructing all clients to pull data from private/public cloud servers simultaneously for every 2 hours. We report results averaged unless mentioned otherwise.

B. Deployment in VMs

We first evaluate *Mystique*'s performance when it is implemented in a VM, as shown in Figure 6(a). Figure 9∼11 show the performance of *Mystique* and other schemes, including TCT, throughput and load time.

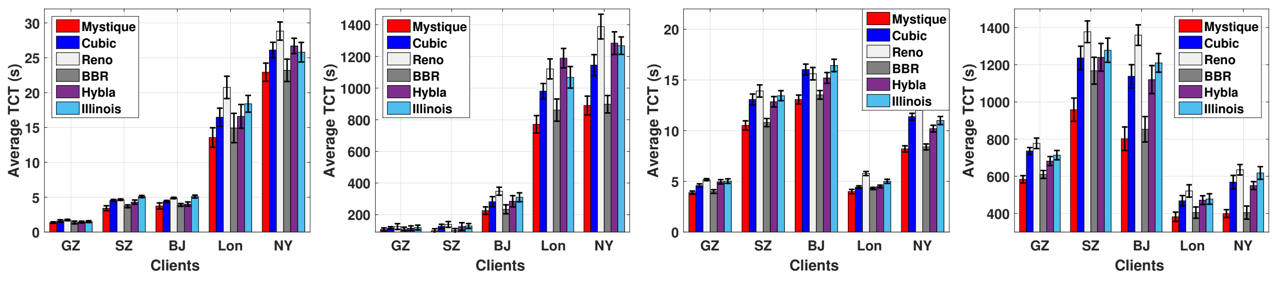
Private Cloud: In comparison with Cubic, *Mystique* performs better (8.73%∼26.18%) for both transfer of small file and large file in private cloud. Comparing with Reno, *Mystique* achieves up to 36.9% improvement among all clients. Figure10 shows that *Mystique* has about 29.28% and 31.03% throughput improvement over Cubic and Reno, respectively. On the other hand, *Mystique* outperforms Hybla and Illinois by 21.43% and 25.92% on average, respectively. Specially, *Mystique* achieves up to 31.03% higher throughput compared to other five schemes and these results lie in the confidence interval with confidence level 90%. This convincingly demonstrate that *Mystique* can improve transfer performance. Meanwhile, for SZ and BJ, the performance of Cubic is better than Reno. On the contrary, Reno performs better in clients Lon and NY. These results confirm the observations we discussed in Section I and Section II. Moreover, *Mystique* also has better performance than other schemes for Websites loading, up to 25.62% performance enhancement which shows that *Mystique* is able to reduce service latency for Websites.

Public Cloud: *Mystique* also achieves good performance in public cloud deployment. Evidently, *Mystique* reduces average completion time for both small file and large file by up to 14.29% and 32.5% respectively over Cubic and 11.33%∼35.14% for both over Reno. For throughput, *Mystique* achieves up to 9.74% and 14.2% better performance compared to Cubic and Reno respectively. In contrast to BBR, *Mystique* can reduce TCT by up to 5.81% and 5.05% for small file and large file respectively. And *Mystique* outperforms Hybla and Illinois by up to 20.45% and 34.1% respectively. Particularly, clients Lon and NY are instances hosted on Amazon AWS. Thus the transfer between public cloud Web servers and Lon and NY are completed quickly, due to AWS inter-DC acceleration. Besides, *Mystique* reduces load time of accessing Websites by about 17.13% on average, when compared with other schemes.

Clearly, the evaluation results above have confirmed our observations in Section II. In addition, the results demonstrate that *Mystique* can achieve the best performance with up to 36.9% improvement on TCT. This improvement also demonstrates that *Mystique* reduces the transfer latency effectively.

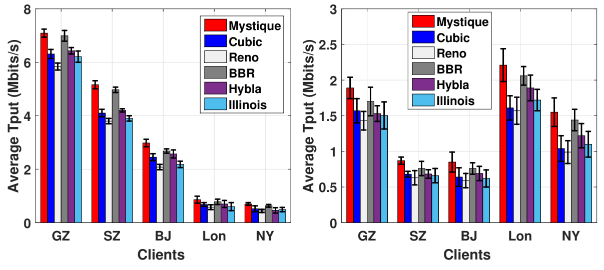
C. Deployment in Hypervisors

We next evaluate *Mystique*'s performance when it runs inside server (like a hypervisor), as shown in Figure 6(b). Figure 12∼14 depict the performance of all compared schemes, including TCT, throughput and load time.



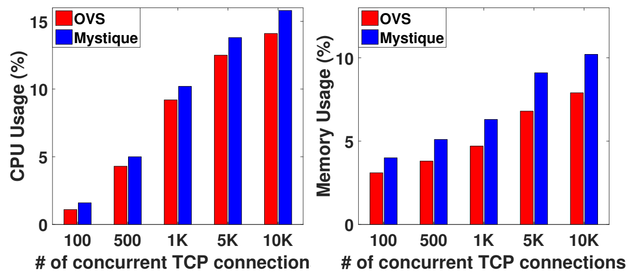
(a) Small file in private cloud (b) Large file in private cloud (c) Small file in public cloud (d) Large file in public cloud

Fig. 15. The average transfer completion time for both small and large files in private and public cloud, when *Mystique* is deployed in switch.



(a) In private cloud (b) In public cloud

Fig. 16. The average throughput (Tput) in private cloud and public cloud, when *Mystique* is deployed in switch.



(a) CPU usage (b) Memory usage

Fig. 17. The CPU and memory usage of *Mystique* under different scale of concurrent TCP connections

Private Cloud: First, we observe that the performance of *Mystique* is better than Cubic with latency reduction by up to 18.44% for both the transfer of small file and large file. Compared to Reno, *Mystique* reduces 14.23%~26.39% TCT among all clients. By measuring throughput, *Mystique* can improve Web server's average throughput up to about 28.57% and 33.86% against Cubic and Reno, respectively. On the other hand, *Mystique* outperforms Hybla and Illinois by up to 32.73% and 29.93%, respectively. Particularly, *Mystique* improves 6.81%~33.86% throughput when compared to other schemes. The throughput results lie in the confidence interval with confidence level 90% which demonstrates that *Mystique* outperforms other schemes with convincing evidence. When it comes to Websites' load time, *Mystique* achieves up to 28.75% latency reducing, compared to other congestion controls.

Public Cloud: *Mystique* also achieves good performance in Public Cloud deployment. Compared to Cubic, *Mystique* reduces the average TCT for both small file and large file by up to 23.48% and 25.63% respectively. Moreover, *Mystique* outperforms Reno (10.72%~30.76%) among all clients. Besides, compared to BBR, *Mystique* can reduce TCT by up to 7.99% and 4.66% for small file and large file respectively. And *Mystique* achieves 21.8% and 24.68% better performance on average, comparing with Hybla and Illinois respectively. In addition, *Mystique* achieves up to 16.66% higher throughput compared to other schemes. And, *Mystique* reduces load time of accessing Websites by about 17.22% on average, compared to other schemes.

Additionally, by comparing the testbed results of both deployment in VM and deployment in Hypervisor, we conclude that *Mystique* on VM achieves comparable performance as it on Hypervisor, even with additional one-hop delay. Since *Mystique* is installed in datacenter where latency is relatively low, such one-hop delay is negligible.

D. Deployment in Routers/Switches

Performance of *Mystique* deployed in routers/switches is evaluated in a Mininet simulation environment. Since links are configured according to Figure 8, for the ease of description and consistency, client 1 to client 5 are named as the machine name in Table V according their link-setting. Besides, both private and public cloud are simulated in experiments and unmodified OVS is used when evaluating all schemes except *Mystique*. Results are the average of 10 runs which are shown in Figure 15 and Figure 16.

Private Cloud: Compared to Cubic, *Mystique* can reduce TCT by up to 24.6% and 31.25% for the transfer of small file and large file respectively. Furthermore, *Mystique* outperforms Reno (17.9%~35.9%) on all clients. As for throughput, compared to Cubic and Reno, *Mystique* can increase Web server's throughput for about 23.91% and 27.11% on average, respectively. Comparing with BBR, *Mystique* can reduce TCT by up to 5.86% and 4.59% for the transfer of small file and large file respectively. In the meantime, *Mystique* obtains 13.15% higher throughput than BBR's. When it comes to the comparison with Hybla and Illinois, *Mystique* improve the performance by 19.56% and 25.34% on average, respectively. Further, *Mystique* obtains 5.46%~27.11% throughput improvement compared to other five schemes. with confidence level 90%, the throughput results lie in the confidence interval.

Public Cloud: *Mystique* reduces the average TCT for both small file and large file by up to 25.88% and 29.74% respectively. Compared to Reno, *Mystique* can reduce TCT by 16.31%~37.08% among all clients. In addition, *Mystique*

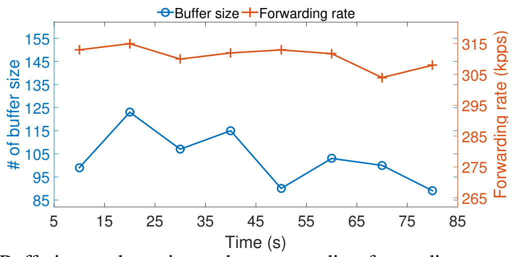


Fig. 18. Buffering packets size and corresponding forwarding rate.

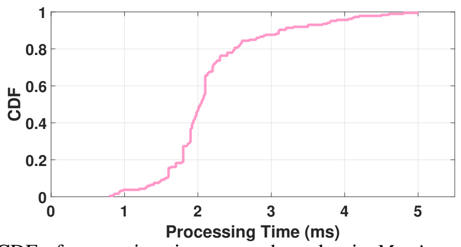


Fig. 19. CDF of processing time on each packet in *Mystique*.

achieves up to 20.38% and 30.76% higher throughput compared to Cubic and Reno respectively. Comparing with BBR, *Mystique* can reduce TCT by up to 10.86% and 18.36% for the transfer of small file and large file respectively. Meanwhile, *Mystique* obtains 13.8% higher throughput than BBR's. Further, *Mystique* outperforms Hybla and Illinois by 20.34% and 25.68% on average, respectively.

With above simulation experiments, we can conclude that *Mystique* outperforms other five schemes when *Mystique* is deployed in Router/Switches. Particularly, *Mystique* achieves up to 37.08% lower TCT when compared to other five schemes. These results demonstrate the effectiveness of *Mystique* again.

E. Overhead

The overhead of *Mystique* is also evaluated in test-bed experiments. Both CPU usage and memory usage are measured by using *sar* [38] with simulating concurrent connections. Multiple simultaneous TCP flows are started from GZ to Pri-backend via *Mystique1* (similar to Figure 6(a)) by using Web Bench [41]. Other services on the *Mystique* server are shut down during the experiment.

CPU Usage: The system-wide CPU overhead of *Mystique* compared to that of original OVS is shown in Figure 17(a). Though *Mystique* increases CPU usage in all cases, the increase is acceptable. The largest difference is less than 2 percentage points: the OVS and *Mystique* have 14.1% and 15.8% utilization, respectively for 10K connections were generated.

Memory Usage: The system-wide memory overhead of *Mystique* compared to that of original OVS is shown in Figure 17(b). Similar to CPU usage, *Mystique* increases memory usage in all cases. In the worst case with 10K connections, *Mystique* just uses 3% memory more. We believe the usage increase is acceptable due to memory size for switching chips has grown five times over the past four years [42].

Buffer Size: The buffer size of *Mystique* is measured by triggering both small and large file transfers from all five clients simultaneously. Figure 18 depicts the result of the number of buffering packet of the worst case (actually the connection from NY) from time 10 second to time 80 second. From the figure, we can tell that the largest buffer size is 125 which is affordable. Besides, the forwarding rate is also measured, which is slightly above 300K packet per second.

Packet Processing Time: Packet processing time of *Mystique* is also measured. Results in Figure 19 show that, in most cases, *Mystique* can process a packet within 2.5ms, which is negligible compared the latency of Web service.

VI. Related Works

With new network environments emerging, new congestion controls are in great need to enhance the TCP performance at such varied scenarios. Dozens of CCs have been proposed successively [10] [18] [17] [15] [43] [44] and most of them have their target scenarios. For example, Sprout [17] was proposed to cope with wireless network, while DCTCP [44] and DCQCN [43] were applied for optimizing TCP performance in DCNs. However, the performance for the environment except target scenarios remains uncertainty. Hence, only using one CC to handle all traffic is not enough, especially facing more complicated environment.

Rather than proposing a new congestion control, some works investigate if congestion controls can be implemented in a overlay manner. AC/DC [11] and vCC [7] are frontiers which convert default CC into operator-defined datacenter TCP congestion controls. AC/DC suggests that datacenter administrators take control of the CC of all the VMs. In particular, it demonstrates this approach by AC/DC implemented a vSwitch-based DCTCP congestion control. And vCC adopts a translation layer between different CCs. The evaluation of these two schemes has demonstrated their excellent performance in translating CC between VMs and actual network.

Specifically, *Mystique* was inspired by these two schemes, with a focus on Web service, allowing administrators to perform fine-grained and dynamic congestion controls.

Recently, NetKernel [45] provides a vision of network stack as a service in public cloud which decouples network stack from OS kernel. And CCP [46] [47] implements different CCs in a separate agent outside the datapath which provide better abstractions for implementation of CC. Both NetKernel and CCP share some objects of *Mystique*, such as flexibility of deploying new protocols. However, these two proposals need to modify server's stacks while *Mystique* requires no modification of servers.

VII. Conclusions

Each congestion control has its own suitable role to play in various networking circumstance while each web server may service clients from varied network environment. In this paper, we proposed *Mystique*, a resilient transparent congestion control enforcement scheme, which aims to enforce more appropriate congestion controls for corresponding network environment with the purpose of reducing web service latency. Our extensive test-bed results have demonstrated the effectiveness of *Mystique* with affordable overhead. The future direction of this work will be to investigate and implement more newly proposed congestion controls and more effective switching logics to attain more effective transmission.

Acknowledgment

The authors would like to thank anonymous reviewers for providing valuable comments to improve this work. This work is partially supported by National Natural Science Foundation of China (NSFC) Project No. 61772235 and 61602210; NSFC Key Project No. 61532013 and No. 61872239; 0007/2018/A1, 0060/2019/A1, DCT-MoST Joint-project No. 025/2015/AMJ of Science and Technology Development Fund, Macao S.A.R (FDCT); University of Macau Grant Nos: MYRG2018-00237-RTO, CPG2019-00004-FST and SRG2018-00111-FST; the Fundamental Research Funds for the Central Universities (21617409, 21617408 and 21619404); the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; the Science and Technology Planning Project of Guangdong (2015A030401043, 2017A040405029); the Science and Technology Planning Project of Guangzhou (201902010041); the Educational Commission of Guangdong Province (2018KTSCX016).

References

[1] I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla, "Why is the internet so slow?!" in *International Conference on Passive and Active Network Measurement*. Springer, 2017, pp. 173–187.

[2] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 159–170, 2013.

[3] X. Chen, H. Zhai, J. Wang, and Y. Fang, "A survey on improving TCP performance over wireless networks," *Resource management in wireless networking*, pp. 657–695, 2005.

[4] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 350–361.

[5] C. Li and L. Cui, "A Novel NFV Schedule Optimization Approach with Sensitivity to Packets Dropping Positions," in *Proceedings of the 2018 Workshop on Theory and Practice for Integrated Cloud, Fog and Edge Computing Paradigms*, ser. TOPIC '18. New York, NY, USA: ACM, 2018, pp. 23–28.

[6] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, J. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2014.

[7] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. Mckeown, I. Abraham, and I. Keslassy, "Virtualized Congestion Control," in *ACM SIGCOMM 2016*, 2016, pp. 230–243.

[8] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno modification to TCP's fast recovery algorithm," Tech. Rep., 2012.

[9] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *Acm Sigops Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[10] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: congestion-based congestion control," *Queue*, vol. 60, no. 2, pp. 58–66, 2017.

[11] K. He, E. Rozner, K. Agarwal, Y. J. Gu, W. Felter, J. Carter, and A. Akella, "AC/DC TCP: Virtual congestion control enforcement for datacenter networks," in *ACM SIGCOMM 2016*. ACM, 2016, pp. 244–257.

[12] Y. Zhang, L. Cui, F. P. Tso, Q. Guan, W. Jia, and J. Zhou, "A Fine-grained and Transparent Congestion Control Enforcement Scheme," in *Proceedings of the Applied Networking Research Workshop*. ACM, 2018, pp. 26–32.

[13] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments Using Container-based Emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '12. New York, NY, USA: ACM, 2012, pp. 253–264.

[14] A. Singla, B. Chandrasekaran, P. Godfrey, and B. Maggs, "The internet at the speed of light," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014, p. 1.

[15] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance." in *NSDI*, vol. 1, no. 2.3, 2015, p. 2.

[16] V. Arun and H. Balakrishnan, "Copa: Practical Delay-Based Congestion Control for the Internet," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[17] K. Winstein, A. Sivaraman, H. Balakrishnan *et al.*, "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks." in *NSDI*, vol. 1, no. 1, 2013, pp. 2–3.

[18] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "PCC Vivace: Online-Learning Congestion Control," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). USENIX Association*, 2018.

[19] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for internet congestion-control research," in *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018, pp. 731–743.

[20] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 479–490.

[21] iPerf, "The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/iperf-doc.php, [Online; accessed 20-June-2019].

[22] C. Caini and R. Firrincieli, "TCP Hybla: a TCP enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.

[23] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of BBR congestion control," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 2017, pp. 1–10.

[24] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose, "Modeling tcp reno performance: a simple model and its empirical validation," *IEEE/ACM transactions on Networking*, vol. 8, no. 2, pp. 133–145, 2000.

[25] G. Judd, "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter." in *12nd USENIX NSDI*, 2015, pp. 145–157.

[26] W3Techs, "Historical trends in the usage of operating systems for websites," https://w3techs.com/technologies/history_overview/operating_system, [Online; accessed 20-June-2019].

[27] L. Xu, A. Zimmermann, L. Eggert, I. Rhee, R. Scheffenegger, and S. Ha, "Cubic for fast long-distance networks," 2018.

[28] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The Design and Implementation of Open vSwitch." in *NSDI*, 2015, pp. 117–130.

[29] V. Jacobson, R. Braden, and D. Borman, *TCP Extensions for High Performance*. RFC Editor, 1992.

[30] J. Postel, "The TCP maximum segment size and related topics," 1983.

[31] "OpenvSwitch 2.7.0," http://openvswitch.org/releases/openvswitch-2.7.0.tar.gz.

[32] R. Love, *Linux Kernel Development (Novell Press)*. Novell Press, 2005.

[33] "AWS Cloud Computing Service," https://aws.amazon.com.

[34] "Openflow specification 1.5.1," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[35] "Linux kernel 4.13 source code," https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.13.tar.xz.

[36] curl, "A command line tool and library for transferring data with URL syntax," https://github.com/curl/curl, [Online; accessed 20-June-2019].

[37] "Chrome DevTools," https://github.com/ChromeDevTools/awesome-chrome-devtools.

[38] "SYSSTAT," http://sebastien.godard.pagesperso-orange.fr/.

[39] J. Neyman, "X-outline of a theory of statistical estimation based on the classical theory of probability," *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 236, no. 767, pp. 333–380, 1937.

[40] M. Kendall and A. Stuart, "The advanced theory of statistics: Inference and relationship, vol. 2," *Griffin, London*, 1961.

[41] "WebBench 1.5," http://home.tiscali.cz/~cz210552/webbench.html.

[42] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017*. ACM, 2017, pp. 15–28.

[43] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control

for large-scale RDMA deployments," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 523–536.

[44] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

[45] Z. Niu, H. Xu, D. Han, P. Cheng, Y. Xiong, G. Chen, and K. Winstein, "Network Stack as a Service in the Cloud," in *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets 17)*. ACM, 2017.

[46] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan, "Restructuring endpoint congestion control," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018*. ACM, 2018, pp. 30–43.

[47] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan, "The case for moving congestion control out of the datapath," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, (HotNets 16)*. ACM, 2017, pp. 101–107.

Yuxiang Zhang received his the BEng, and MEng degree from Jinan University, China, in 2015 and 2018 respectively. He is currently a research assistant at University of Macau. He has broad interests in networking systems, with focuses on the area of datacenter networking and network optimization.

Lin Cui is currently with the Department of Computer Science at Jinan University, Guangzhou, China. He received the Ph.D. degree from City University of Hong Kong in 2013. He has broad interests in networking systems, with focuses on the following topics: cloud data center resource management, data center networking, software defined networking (SDN), virtualization and so on.

Fung Po Tso received his BEng, MPhil and PhD degrees from City University of Hong Kong in 2006, 2007 and 2011 respectively. He is currently lecturer in the Department of Computer Science at the Loughborough University. Prior to joining Loughborough, he worked as SICSA Next Generation Internet Fellow at the School of Computing Science, University of Glasgow during 2011-2014 and lecturer in Liverpool John Moores University during 2014-2017. He has published more than 20 research articles in top venues and outlets. His research interests include: network policy management, network measurement and optimisation, cloud data centre resource management, data centre networking, software defined networking (SDN), distributed systems as well as mobile computing and system.

Quanlong Guan received the MS and PhD degrees from Jinan University, China, in 2006 and 2014. He is currently a professor of engineering in Jinan University. He is directing the Guangdong R&D Institute for the big data of service and application on education. His research interests include network security, big data protection and processing. Current and prior work includes analytics for network log and security, big data application and mobile security. His research has been funded by National Natural Science Foundation of China, Guangdong Key Technologies R&D Program of China.

Weijia Jia is currently a Chair Professor, Deputy Director of State Kay Laboratory of Internet of Things for Smart City, Head of Center of Data Science at the University of Macau. He has been Zhiyuan Chair Prof at Shanghai Jiaotong University, China (where he received 2013 China 1000 Talent Award). He received BSc/MSc from Center South University, China in 82/84 and Master of Applied Sci./PhD from Polytechnic Faculty of Mons, Belgium in 92/93, respectively, all in computer science. For 93-95, he joined German National Research Center for Information Science (GMD) in Bonn (St. Augustine) as research fellow. From 95-13, he worked in City University of Hong Kong as a full professor in Computer Science Dept. His research interests include smart city; next generation IoT, knowledge graph constructions; multicast and anycast QoS routing protocols, wireless sensor networks and distributed systems. In these fields, he has over 500 publications in the prestige international journals/conferences and research books and book chapters. His contributions can be summaried from the aspects of vertex cover and efficient anycast for optimal placement and routing of severs/sensors in many applications of IoT/sensor/wireless networks and the Internet. He has received Best Product Awards from the Internatonal Science&Tech. Expos (Shenzhen) in 2011/2012 and 1st- Prize of Scientific Research Awards from Ministry of Education of PR China in 2017 (list 2). He has served as area editor for various prestige international journals, chair and PC member/keynote speaker for many top international conferences. He is the Senior Member of IEEE and the Member of ACM.

Jipeng Zhou u received B.Sc. degree and M.Sc. degree from Northwest University Xian, China, in 1983 and 1988, and the Ph.D. degree from the University of Hong Kong in 2000. From July 1983 to March 1997, he was a lecturer and an associate professor in Northwest University Xian, China. From Dec. 2000 to Feb. 2002, he was a Postdoctoral fellowship in Nanyang Technology University. He joined Jinan University in 2002, he is currently a professor. His research areas include parallel and distribution computing, routing protocol, location service, channel and bandwidth assignment and energy problems to wireless networks. He has published over 90 papers. He is a senior member of CCF and a member of ACM.