A Fine-grained and Transparent Congestion Control Enforcement Scheme*

Yuxiang Zhang
Jinan University

Lin Cui
Jinan University

Fung Po Tso
Loughborough University

Quanlong Guan
Jinan University

Weijia Jia
University of Macau

Jipeng Zhou
Jinan University

ABSTRACT

In practice, a single TCP congestion control 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 algorithm may not always be the most suitable one. Adjusting congestion control usually to meet different networking scenarios requires modification of servers' TCP stacks. This is difficult, if not impossible, due to various operating systems and different configurations on the servers. In this paper, we propose *Mystique*, a light-weight and flexible scheme that allows administrators (or operators) to deploy any congestion control schemes transparently without changing existing TCP stacks on servers. We have implemented *Mystique* in Open vSwitch (OVS) and conducted extensive test-bed experiments in public cloud environments. We have extensively evaluated *Mystique* and the results have demonstrated that it is able to effectively adapt to varying network conditions, and can always employ the most suitable congestion control for each TCP connection. *Mystique* can significantly reduce latency by up to 37.8% in comparison with other congestion controls.

1 INTRODUCTION

Recent years have seen many Web applications moved into cloud datacenters to take advantage of the economy of scale. It is well known that Web latency inversely correlates with revenue and profit [20]. Reducing latency is of profound importance for providers [7].

*Corresponding author: Dr. Lin Cui (tcuilin@jnu.edu.cn)

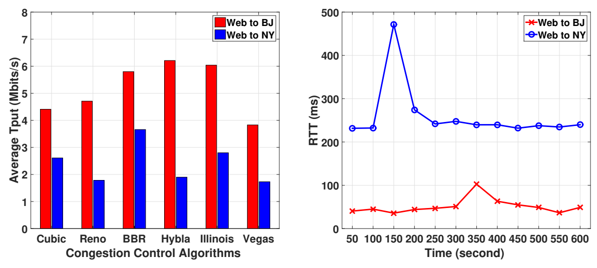
ANRW '18, July 16, 2018, Montreal, QC, Canada
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5585-8/18/07…$15.00
https://doi.org/10.1145/3232755.3232778

In response, administrators (or operators) opt to use network appliances to reduce network latency. For example, TCP proxies and WAN optimizers are used for such optimization [5, 9]. However, these appliances have fixed capacity and thus do not scale well with rapidly increasing traffic volume [5]. Besides, many turned to optimizing TCP congestion control for improving network latency. As a result, plenty of TCP congestion controls have been proposed, e.g., Reno [13], Cubic [10] and BBR [4]. However, our extensive evaluations have shown that none of these proposals can constantly outperform one another. In fact, they only reach peak performance when some specific packet loss and network delay conditions are met, and starts to degrade dramatically when these change. The degradation of performance caused by inappropriate congestion control algorithms can lead to decreasing throughput and increasing latency.

In order to better understand such performance diversity, we performed several experiments by triggering 50MB file transfer from a Web server (in Guangzhou, China) to two clients in Beijing (BJ) and New York (NY) , respectively. The throughput results in Figure 1(a) show that performance of different congestion controls are varied under different network conditions. For example, Reno has better performance than Cubic for the transferring from Web to BJ, while performs worse when transmitting from Web to NY. The main reason is that network conditions are strikingly different for the two connections, e.g., RTT. Figure 1(b) shows the measured RTT during experiments for both connections. Even though both clients connect to the same Web server, their RTTs are significantly different. RTT of Web$\rightarrow$NY is about 10 times longer than that of Web$\rightarrow$BJ. Particularly, RTTs are changed dynamically for both connection. This implies that *the best congestion control may change over time, even for a single TCP connection.* However, congestion controls are usually determined by TCP stack of servers or changed on per-socket basis implemented inside application's source code.

Furthermore, many Web servers[1] in cloud datacenters have different operating systems and configurations, e.g., Linux or

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



(a) Performance dynamics **(b) RTT dynamics**

Figure 1: The dynamics of congestion control's performance

Windows with different kernel versions and congestion control algorithms. Considering such diversity and vast number of Web servers, adjusting congestion controls (e.g., deploying new advanced algorithms) is a difficult, if not impossible, task for administrators[6, 12]. Yet, on the other hand, administrators sometimes cannot modify servers' TCP stack directly because of the security constraints which makes it more difficult for administrators to meet agreed-upon SLAs (e.g., latency performance).

Hence, we ask a question: *Can we design a fine-grained congestion control scheme that can always employ the most suitable congestion control algorithm for each TCP connection, adapting to network diversities and dynamics, without modifying TCP stacks of Web servers?*

In this paper, inspired by works done in [6] and [12], we present *Mystique*, a resilient congestion control enforcement without changing TCP stack on servers. Advanced TCP congestion control algorithms can be easily implemented by using APIs provided by *Mystique*. Moreover, *Mystique* can effectively adapt to network conditions and dynamically employ the most suitable algorithm for each connection according to rules specified by administrators.

The main contributions of this paper are as follows:

(1) We designed *Mystique*, which allows network administrators (or operators) dynamically adjusting and deploying congestion control algorithms without modifying TCP stacks of servers.
(2) A prototype of *Mystique* is implemented based on Open vSwitch (OVS). *Mystique* is light-weight, containing only about 1400 lines of code.
(3) Preliminary test-bed experiments are conducted with Web servers located in AWS cloud. Experiment results show that *Mystique* works effectively, reducing latency by up to 37.8% compared to other schemes.

2 BACKGROUND AND MOTIVATIONS

2.1 Background

Latency for Web service is closely linked to revenue and profit [7]. Many service providers use network functions such as TCP proxies and WAN optimizers for reducing latency [5, 9]. However, the scalability is of great challenge. When there is a burst of requests for service, the performance of such network functions can be easily saturated due to the limiting processing capacity.

On the other hand, TCP congestion control is known to have significant impact on network performance. As a result, TCP congestion control has been widely studied and many schemes have been proposed to improve performance [1–4, 8, 10, 13, 16]. These schemes perform well in their target scenarios but have varied performance in other circumstances. However, service providers usually deploy a diversity of Web servers which may run different versions of operating systems (e.g., Linux and Windows) and be configured with different congestion controls. Adjusting TCP stacks for such a large amount of Web servers could be overwhelming for network administrators. Specially, in multi-tenants cloud datacenters, network operators may be prohibited from upgrading TCP stacks on particular Web servers for security reasons. Furthermore, the congestion control algorithm determined by TCP stack will take effect on all TCP connections of current server, which is inconvenient considering network diversities from all clients to the server (see Section 2.2).

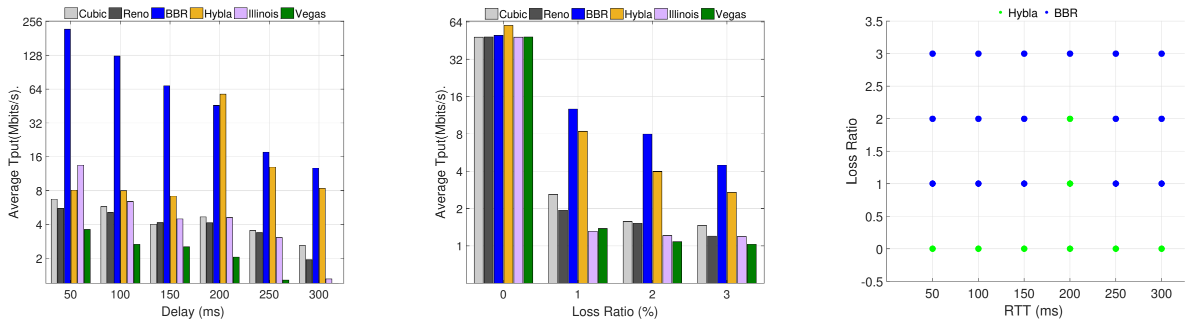
Clearly, it is necessary to provide a mechanism which allows drop-in replacement of TCP stack for per-flow (as oppose to per-server) level of granularity, giving network administrator sufficient control of network resource whilst improving latency for Web services.

2.2 Problem Exploration

We have seen in Figure 1 that the same congestion control algorithm may have different performance under different network environment. To better understand the impact of network dynamics on congestion control, we conducted a Mininet [11] based experiment to quantify the performance variation of TCP congestion controls under different network conditions The network contains two servers connected to two switches, respectively, in a line topology.

Based on the results shown in Figure 2, we have three important observations that highlight the dynamics of congestion control's performance

Observation 1: *The performance of TCP congestion control varies under different network delays with constant loss ratio.* Results in Figure 2(a) shows that BBR has the best performance in most scenarios, while Hybla [3] outperforms all other algorithms at 200*ms*. Interestingly, BBR's performance



(a) Performance dynamics **(b) RTT dynamics** **(c) The most suitable CC**

Figure 2: Problem exploration

degrade gradually when network delay is increased, whereas Hybla's performance is improved.

Observation 2: *The performance of TCP congestion control varies under different loss ratios with constant delay.* Results in Figure 2(b) shows that all TCP variants have different performance under scenarios with different loss ratio. Specially, Hybla can have the best performance compared to other schemes when there is no loss, while BBR has the best performance with the presence of packet loss. And Illinois can achieve comparable performance to Cubic when loss ratio equals to 0 and 1% but get opposite performance under loss ratio equals to 2% and 3%.

Observation 3: *No single congestion control suits all network conditions.* Figure 2(c) shows the best performing congestion control 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.

2.3 Design Principles

The goal of *Mystique* is to provide a transparent platform allowing network administrators to dynamically adjusting congestion control in a fine-grained granularity without modifying TCP stacks of web servers. Thus, *transparency* is the most important characteristics of *Mystique*. *Mystique* should allow network administrators or operators to enforce advanced congestion controls without touching TCP stacks of servers. Deployment and operations of *Mystique* should be transparent to both web servers and clients [12, 15, 20]. Besides, *Mystique* should be able to apply different congestion controls on a per-flow basis and switch congestion control according to current network status, and consume as less resource as possible.

3 *MYSTIQUE* DESIGN

3.1 Obtaining Congestion Control States

One of *Mystique*'s tasks is to obtain congestion control states (e.g., *min_rtt*, *max_bw* and loss ratio) on packet level. These states are then used as inputs of each congestion control algorithm implemented above *Mystique*.

Since *Mystique* is implemented in the *datapath* module of OVS, all traffic passing through can be monitored. TCP sequence number can be obtained directly from packets. Similar to [12], *Mystique* adopts *una* to record the first packet's sequence number which has been sent, but not yet ACKed. *nxt* is used to record the sequence number of the next packet to be sent (but not yet received it). Packets between *una* and *nxt* are being transmitted. Each ACK contains an acknowledgement number (*acknum*), and *una* is updated when *acknum* > *una*. When a packet is received from Web servers, *nxt* is updated if its sequence number is larger than or equal to current value of *nxt*. With *una* and *nxt*, detecting packet loss is easy. When receiving a ACK packet, if *acknum* ≤ *una*, the a local *dupack* counter is updated. When *dupack* counts to 3, it means a packet loss happened [14].

When a new TCP connection is detected, minimal RTT *min_rtt* and maximal sending rate *max_bw* are initialized to be ∞ and 0 respectively. Flos's current RTT *c_rtt* and current sending rate *c_bw* are used for record current RTT and Bandwidth. When ACK arrives, *c_rtt* can be updated by computing the difference between ACK and corresponding arriving timestamps. In the meantime, the size of acknowledged bytes *acked* can be obtained by *acknum* - *sna*. Hence, *c_bw* would be equal to (*acked* / *c_rtt*). If *c_rtt* < *min_rtt*, the minimal RTT *min_rtt* would be updated. Similarly, *max_bw* is updated when *c_bw* is larger than *max_bw*. And *max_rtt* could be measured similar to *min_rtt* and *max_bw*.

3.2 Implementing Congestion Control

All states above can be obtained by canonical TCP congestion controls through APIs provided by *Mystique* (summarized in Table 1), and then used to compute appropriate congestion window *cwnd*. Next, we will use our implementation of BBR as an example to elaborate how congestion controls are implemented based on *Mystique*. Other congestion controls can be implemented in similar way.

In BBR, sender needs to continuously estimate the bottleneck bandwidth (BtlBw) and round-trip propagation time (RTprop) and let the total data in flight be equal to the BDP (= BtlBw $\times$ RTprop) [4]. By adjusting the *cwnd* based on BDP, BBR guarantees that the bottleneck can run at 100 percent utilization and preventing bottleneck starvation but not overfilling. *min_rtt* (obtained by *getMinRTT()*) and *max_bw* (obtained by *getMaxBW()*) are tracked continuously for each TCP connection. *Mystique* recognizes them as RTprop and BtlBw to compute according *cwnd*.

Besides, the original BBR implementation uses 4 modes to decide how fast to send, i.e., startup, drain, probe_bw and probe_rtt, which are used to increase sending rate quickly and estimate whether the pipe's bandwidth has been fully utilized. However, it is a obstacle for *Mystique* to enforcing such 4 modes. Because *Mystique* is not a host based scheme. Thus *Mystique* cannot modify any end-host's TCP stack directly which leads to a obstacle of enforcing such 4 modes, especially forcing server to rapidly increase its sending rate when server's congestion window is smaller than *Mystique*'s *cwnd*. Therefore, we seek a tradeoff between such measurement and *cwnd* computing.

Mystique defines time unit *period* (set its value via *setPeriod()*). And every *period* some states (i.e., *min_rtt*, *max_bw*) would be updated as follow: *min_rtt* = *min_rtt* + *t_step, max_bw* = *max_bw* - *bw_step*. If the new *min_rtt* is larger than the RTT under current network condition, *min_rtt* would be updated when the next ACK arrives. Otherwise, network condition has been changed and *min_rtt* can be updated to actual minimal RTT with such operation step by step. Also *max_rtt* and *max_bw* are updated similarly. In current *Mystique*'s implementation, *period* is set to 5 second, *t_step* and *bw_step* are configured as *min_rtt*/10 and MSS (via methods *setTstep()* and *setBwstep()*) respectively [19]. Note that in this section we only provide general idea for the implementation of congestion control and leave optimal parameter tuning as an important future work.

3.3 Enforcing Congestion Control

Once the *cwnd* is ready, the next step is to ensure that a Web server's sending rate can adhere to it. TCP provides built-in functionality that can be reprovisioned for *Mystique*. Specifically, TCP's flow control allows a receiver to advertise the amount of data that it is willing to process via a receive window (*rwnd*) [12, 14]. *Mystique* will overwrite *rwnd* with its computed window size *cwnd* (done by setCwnd()) for restricting amount of packets sent from clients to Web servers. Of course, 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*). Web servers with unaltered TCP stacks will naturally follow our enforcement scheme because the stacks will simply follow the standard.

Table 1: Some APIs provided by *Mystique*

Methods	Descriptions
getCRTT()	Get 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

More specific, there are two conditions when *Mystique* enforces its *cwnd*. (a) When *cwnd* in *Mystique* is smaller than the congestion window in Web server, modifying *rwnd* can limit the sending rate effectively. *Mystique* takes the control of server's congestion control which achieves *Mystique*'s goal. (b) When *cwnd* in *Mystique* is larger than the congestion window in Web server, modifying *rwnd* may not be an effective method. Hence, Web server's congestion window must be kept at a high level to allow *Mystique* enforcing its *cwnd*. In response, *Mystique* prevents any congestion signals (e.g., ECN feedback and three duplicated ACKs) are sent to Web server to avoid decreasing of Web server's congestion window. Moreover, *Mystique* adopts packets buffering to handle packet loss and retransmits them if necessary. But, the limitation of *Mystique* is that it cannot force Web server increasing sending rate fast. However, with continuous data transmission, Web server's congestion window would arrive at a high level gradually.

3.4 Dynamic Congestion Control Switching

Mystique always tries to assign suitable congestion controls on a per-flow basis based on each connection conditions. The most suitable congestion control to be employed can be either/both determined by current network congestion states or administrator defined switching logics.

Algorithm 1 shows a simple example of such congestion control algorithm switching logic. In this example, we recognize BBR, Hybla and Illinois can perform well under most network environment. BBR outperforms others with the presence of packet loss. However, the tradeoff mentioned in Section 3.2 can lead to lots of bandwidth occupied by BBR. Thus, Illinois, which outperforms others included BBR when delay

Algorithm 1 An example of congestion control switching logic

```
1:  for each incoming TCP ACK do
2:      if no Loss then
3:          $cwnd$ = TCP_Hybla()
4:      else if RTT < $50ms$ then
5:          $cwnd$ = TCP_Illinois()
6:      else
7:          $cwnd$ = TCP_BBR()
8:      end if
9:      setCwnd(cwnd)
10: end for
```

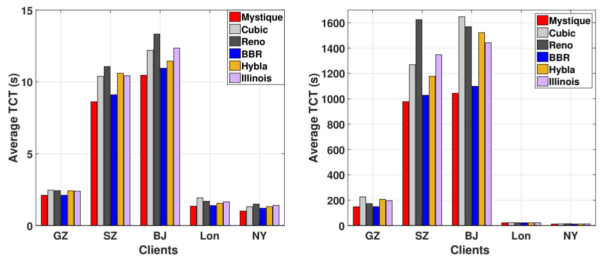
is less than 50ms in Figure 1 (Web→BJ), is adopted to cope with the situation with packet loss in such scenarios. Otherwise, BBR is used to be the most suitable algorithm. When no loss is detected, Hybla is convinced to be the most suitable one.

Since *Mystique* dynamically employ the most suitable congestion control according to connection conditions, algorithm switching would happen for a single TCP connection. In this case, *Mystique* needs to decide whether some parameters continue to be updated or re-initialized. In our current implementation, BBR and Hybla can compute their *cwnd* based on the network states obtained by *Mystique*. However, Illinois needs assistant parameters α and β to compute *cwnd* with additive increasing and multiplicative decreasing. Though the value of α and β would become useless after switching from Illinois to others, α and β are updated continuously even *Mystique* switches to other algorithms in order to prevent degrading the performance if switching back to Illinois later.

Finally, based on *Mystique*, administrators can define more complex switching logic using more metrics, e.g., loss ratio, variation of RTT. Due to space limitation, more other logic are not elaborated here.

3.5 Deployment Locations

Since *Mystique* is implemented on OVS, it can be easily deployed in three possible locations in cloud datacenters: VMs, Hypervisors and Routers/Switches. Deploying *Mystique* in VMs allows network administrators to setup new *Mystique* servers or release old ones dynamically for load-balancing. While deploying *Mystique* in Hypervisors 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. Routers/switches can inherently monitoring all incoming traffic, making *Mystique* can easily enforce congestion control without route redirection. Each deployment choice is suitable for different requirements and scenarios. In practice, combination of these three deployment choices above can be considered.



(a) Small file transfer **(b) Large file transfer**

Figure 3: The average transfer completion time for both small file and large file when *Mystique* is deployed in VM

4 PRELIMINARY EXPERIMENTS

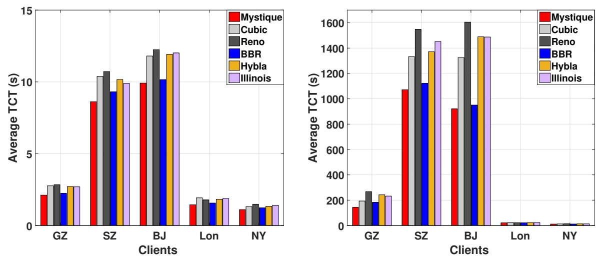
4.1 Prototype Implementation

We have implemented a prototype of *Mystique* on Open vSwitch (OVS) v2.7.0. About 1400 lines of code are added to implement *Mystique*'s basic functions, e.g., tracking congestion control states, managing buffer and switching logic. Flows are hashed on a 5-tuple (IP addresses, ports and protocol) to obtain a flow's state for maintaining the congestion control state. SYN packets are used to create flow entries while FIN packets are used to remove flows entries. Other TCP packets, such as data and ACKs, trigger updates of flow entries. Read-Copy-Update (RCU) hash tables are used to enable efficient lookups while *spinlocks* [17] are used on each flow entry in order to allow for multiple flow entries to be updated simultaneously. Moreover, *skb_clone*() is used for packet buffering to prevent deep-copy of data and multi-threading technique is used for releasing memory space and updating congestion control states and parameters

4.2 Testbed Setups

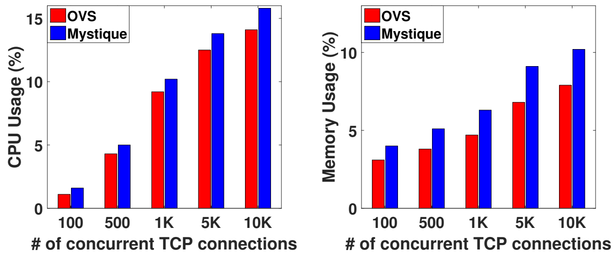
The test-bed consists of 7 servers from AWS clouds. We deploy *Mystique* and 6 Web servers in Singapore AWS datacenters. In order to obtain an in-depth understanding of *Mystique*, our experiments involve 5 clients from all over the world. They locate at Guangzhou(GZ), Shenzhen(SZ), Beijing(BJ), New York(NY) and London(Lon). These clients experienced different RTT and loss ratio when connecting Web servers. All of these Web servers and clients are connected through the Internet and equipped with Intel E5-2686 @ 2.30GHz and 4GB memory.

To understand *Mystique* performance, we compare *Mystique* with Cubic, Reno, BBR, Hybla and Illinois. We use Transfer Completion Time (TCT) as the primary performance metric. For all Web servers, we uploaded two files: small file (OpenFlow Switch Specification v1.5.1.pdf, 1.2MB) and large file (Linux kernel 4.13 source code.xz, 95.9MB).



(a) Small file transfer **(b) Large file transfer**

Figure 4: The average transfer completion time for both small file and large file when *Mystique* is deployed in hypervisors



(a) CPU usage **(b) Memory usage**

Figure 5: The CPU and memory usage of *Mystique*

4.3 Evaluation Results

Deployment in VMs: *Mystique* achieves best performance among all schemes. Compared to Cubic, Mystique reduces the average completion time for both small file and large file by up to 14.29% and 32.5% respectively. Meanwhile *Mystique* outperforms Reno (11.33%~35.14%) among all clients for both small file and large file. Besides, compared to BBR, *Mystique* could 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.

Deployment in Hypervisors: *Mystique* achieves best performance, too. Compared to Cubic, *Mystique* reduces the average TCT by up to 25.63%. Meanwhile, *Mystique* outperform Reno (10.72%~30.76%) among all clients. Besides, compared to BBR, *Mystique* could reduce TCT by up to 7.99% and 4.66% for small file and large file respectively. And *Mystique* outperforms Hybla and Illinois by up to 27.9% and 31.48% respectively. Additionally, by comparing the testbed results of both deployment in VM and deployment in Hypervisor, we observe 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.

Overhead: We have also evaluated the overhead of *Mystique* using test-bed experiments. Both CPU usage and memory usage are measured by using *sar* with simulating concurrent connection. The system-wide CPU overhead of *Mystique* is shown in Figure 5(a). While *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. The system-wide memory overhead of *Mystique* is shown in Figure 5(b). Similar to CPU usage, *Mystique* increases memory usage in all cases. In the worst case with 10K that 10K connections, *Mystique* just uses 3% memory more.

5 RELATED WORKS

AC/DC [12] and vCC [6] are frontiers which converts default congestion control into operator-defined datacenter TCP congestion control. AC/DC suggests that datacenter administrators could take control of the TCP congestion control of all the VMs via implementing congestion control on vSwitch. In the meantime, vCC shares some AC/DC's goals and design details. And vCC adopts a translation layer between different congestion control algorithms. The evaluation of these two schemes has demonstrated their excellent performance in translating congestion control between VMs and actual network. These two schemes rely on DCTCP's effectiveness on loss limitation thus they do not adopt buffer for retransmission. However, it may work inside datacenters while degrade performance due to massive packet loss in WAN environment. Specifically, *Mystique* was inspired by these two schemes, with a focuses on Internet services like Web, allowing administrators performing fine-grained and dynamic congestion control. Recently, NetKernel [18] provides a vision of network stack as a service in public cloud which decouples network stack from OS kernel. NetKernel shares some goals of *Mystique*, such as flexibility of deploying new protocols. However, NetKernel needs to update server's kernel which would damage Web service's functioning. On the contrary, *Mystique* prefers unmodifying server's configurations and impacting less on Web service.

6 CONCLUSIONS

Each congestion control mechanism has its own suitable role to play in various network environments while each Web server may service clients from varied network environment under single congestion control. In this paper, we presented *Mystique*, a resilient transparent congestion control enforcement scheme, which aims to enforce more appropriate congestion control for corresponding network environment with the purpose of reducing Web service latency. Our preliminary test-bed results have demonstrated the effectiveness of *Mystique* with affordable overhead.

ACKNOWLEDGMENTS

This work has been partially supportedby Chinese National Research Fund (NSFC) No. 61772235, 61402200 and 61602210; the Fundamental Research Funds for the Central Universities (21617409 and 21617408); the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/2 and EP/P004024/1; FDCT 0007/2018/A1, DCT-MoST Joint-project No. (025/2015/AMJ) of SAR Macau; University of Macau Funds No. CPG2018-00032-FST & SRG2018-00111-FST; NSFC Key Project No. 61532013; National China 973 Project No. 2015CB352401; Shanghai Scientific Innovation Act of STCSM No.15JC1402400 and 985 Project of Shanghai Jiao Tong University: WF220103001; Science and Technology Planning Project of Guangdong Province (China): 2014A040401027, 2015A030401043, 2017A040405029 and 2017B030306016.

REFERENCES

[1] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 329–342. https://www.usenix.org/conference/nsdi18/presentation/arun

[2] Andrea Baiocchi, Angelo P Castellani, and Francesco Vacirca. 2007. YeAH-TCP: yet another highspeed TCP. In *Proc. PFLDnet*, Vol. 7. 37–42.

[3] Carlo Caini and Rosario Firrincieli. 2004. TCP Hybla: a TCP enhancement for heterogeneous networks. *International journal of satellite communications and networking* 22, 5 (2004), 547–566.

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Queue* 60, 2 (2017), 58–66.

[5] Xiang Chen, Hongqiang Zhai, Jianfeng Wang, and Yuguang Fang. 2005. A survey on improving TCP performance over wireless networks. *Resource management in wireless networking* (2005), 657–695.

[6] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick Mckeown, Ittai Abraham, and Isaac Keslassy. 2016. Virtualized Congestion Control. In *ACM SIGCOMM 2016*. 230–243.

[7] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing web latency: the virtue of gentle aggression. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 159–170.

[8] Cheng Peng Fu and Soung C Liew. 2003. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications* 21, 2 (2003), 216–228.

[9] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, Vol. 41. ACM, 350–361.

[10] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *Acm Sigops Operating Systems Review* 42, 5 (2008), 64–74.

[11] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*. ACM, New York, NY, USA, 253–264.

[12] Keqiang He, Eric Rozner, Kanak Agarwal, Yu Jason Gu, Wes Felter, John Carter, and Aditya Akella. 2016. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *ACM SIGCOMM 2016*. ACM, 244–257.

[13] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. 2012. *The NewReno modification to TCP's fast recovery algorithm*. Technical Report.

[14] V Jacobson, R Braden, and D Borman. 1992. *TCP Extensions for High Performance*. RFC Editor. 190–222 pages.

[15] Glenn Judd. 2015. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter.. In *12nd USENIX NSDI*. 145–157.

[16] Shao Liu, Tamer Başar, and Ravi Srikant. 2008. TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* 65, 6 (2008), 417–440.

[17] Robert Love. 2005. *Linux Kernel Development (Novell Press)*. Novell Press.

[18] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. 2017. Network Stack as a Service in the Cloud. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets 17)*. ACM.

[19] Jon Postel. 1983. The TCP maximum segment size and related topics. (1983).

[20] Yuxiang Zhang, Lin Cui, Fung Po Tso, Quanlong Guan, and Weijia Jia. 2017. TCon: A Transparent Congestion Control Deployment Platform for Optimizing WAN Transfers. In *IFIP International Conference on Network and Parallel Computing*. Springer, 49–61.