One Rein to Rule Them All: A Framework for Datacenter-to-User Congestion Control

Kefan Chen
Tsinghua University, Beijing National Research Center for Information Science and Technology (BNRist)

Danfeng Shan
Xi'an Jiaotong University

Xiaohui Luo
Tsinghua University, Beijing National Research Center for Information Science and Technology (BNRist)

Tong Zhang
Nanjing University of Aeronautics and Astronautics

Yajun Yang
Tencent Technology Shenzhen Company

Fengyuan Ren
Tsinghua University, Beijing National Research Center for Information Science and Technology (BNRist)

ABSTRACT

Today, considerable Internet traffic is sent from datacenter and heads for users. The network characteristics of connections served by servers in datacenters are usually diverse. As a result, a specific congestion control algorithm hardly accommodates the heterogeneity and performs well in various scenarios. In this work, we present Rein — a novel framework for Internet congestion control. With Rein, diverse congestion control algorithms can be assigned purposely to connections in one server to adapt to heterogeneity. We design and implement Rein in Linux, and the experiments validate that Rein is capable of smoothly switching among various candidate algorithms on the fly to achieve potential performance gain. Meanwhile, the overheads introduced by Rein are moderate and acceptable.

CCS CONCEPTS

• **Networks** → **Transport protocols**.

KEYWORDS

Congestion Control; Datacenter; Internet

ACM Reference Format:
Kefan Chen, Danfeng Shan, Xiaohui Luo, Tong Zhang, Yajun Yang, and Fengyuan Ren. 2020. One Rein to Rule Them All: A Framework for Datacenter-to-User Congestion Control. In *4th Asia-Pacific Workshop on Networking (APNet '20), August 3–4, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3411029.3411036

1 INTRODUCTION

In today's Internet, a considerable amount of traffic originates from datacenter and heads for users. In such datacenter-to-user transmissions, a single server in datacenter can face various users with diverse network characteristics. The congestion control algorithms (CCAs) deployed in servers play a vital role in the users' quality of experience, especially for large content transfers which dominate Internet traffic [28] [1].

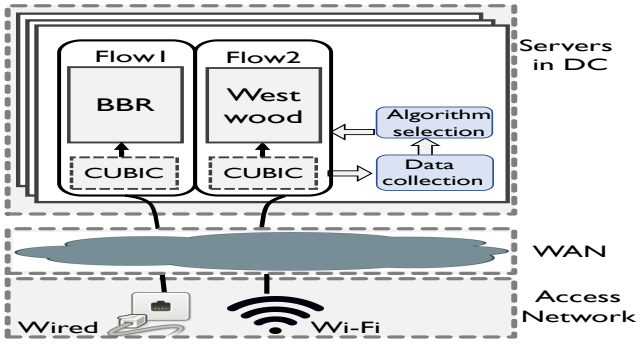

Figure 1: Basic Idea

Although dozens of CCAs are proposed [9], in practice, a single algorithm is usually configured for all connections in a server [25], despite the fact that one specific CCA cannot excel in diverse scenarios (§2.1). To drive congestion control to adapt to heterogeneous network environments, there has been a surge of efforts towards revolutionizing the methodology of congestion control by introducing machine learning techniques [11, 18, 19, 30, 31, 33]. They follow the creed that the handcrafted approaches fail to cope with heterogeneity and react ineffectively to dynamic reality, resulting in less ideal performance. These approaches performs well in certain conditions but have practical issues, resulting in deployment difficulty and performance degradation (§2.2).

Unlike existing work, we take another road to improve the performance of congestion control in heterogeneous environments: selecting the suited CCA for each connection according to the observed network characteristics of this connection. This is inspired by two observations. (1) CCAs in typical network environments (e.g., wired, cellular) have been well studied and continuously proposed. They usually perform expectedly well in their domain (§2.2), and applying matched CCA can attain potential performance gain for heterogeneous networks (§5.2). (2) Selecting appropriate CCA

APNet '20, August 3–4, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-8876-4/20/08…$15.00
https://doi.org/10.1145/3411029.3411036

[1] Transport performance of short connection is largely determined by initial sending window or sending rate.

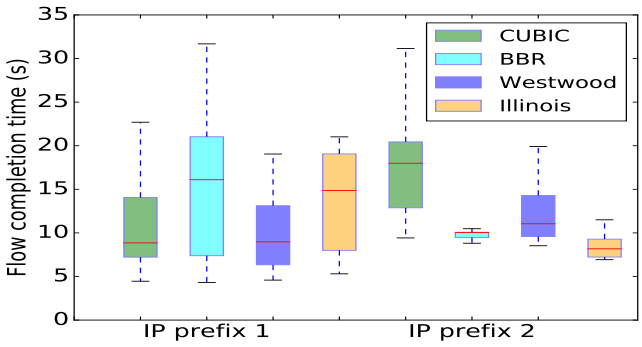

Figure 2: FLT of downloading same size file

is feasible in datacenter-to-user paradigm. Unlike traditional end-to-end congestion control which makes rate control decision solely based on run-time network feedback data of a single connection, servers in datacenter possess more information to make congestion control decisions since they have a global view across connections on two dimensions. *Spatial dimension.* Connections can come from same local network and share common network path. E.g., users of a specific CDN (content delivery network) server are usually in the same network location. *Temporal dimension.* Hosts can access the same datacenter over time. E.g., many users retrieve contents from CDN server periodically. This two-dimension property causes connections share similar network properties (e.g., bottleneck bandwidth, loss patterns) in servers, which is validated by our initial analysis on dataset collected from production CDN servers (§3.2). Selecting CCA for new incoming connections can thus benefits from consulting CCAs' historical performance on previous similar connections. Therefore, we do not intend to design a new omnipotent algorithm to excel in all scenarios. We pin the performance degradation on the mismatch between the rigid single-CCA policy and the diversity of networks, and we seek to bridge the gap by identifying the connection's network characteristics and selecting suited CCA properly.

We introduce Rein — a framework for Internet congestion control, which has two main functions: collecting data for characterizing connection and switching CCA to adapt to heterogeneity. We design and implement the framework in Linux, and briefly describe key ideas of designing CCA selection policy (§3.2). Preliminary evaluations show that Rein provides following benefits: (1) Rein ensures smooth and live transition among various CCA; (2) Rein performs better than one single CCA; (3) Rein introduces modest overheads.

2 MOTIVATION AND BASIC IDEA

2.1 Need for Considering Heterogeneity

The servers in datacenters face clients whose connections have diverse network characteristics. The heterogeneity is largely attributed to the variety of access network (e.g., Wi-Fi, cellular network, optical fibers), carrier network (e.g., different traffic engineering policies) and run-time environment (e.g., multiplexing degree and protocol aggressiveness of competing flows). However, nowadays servers usually adopt a single CCA for all connections

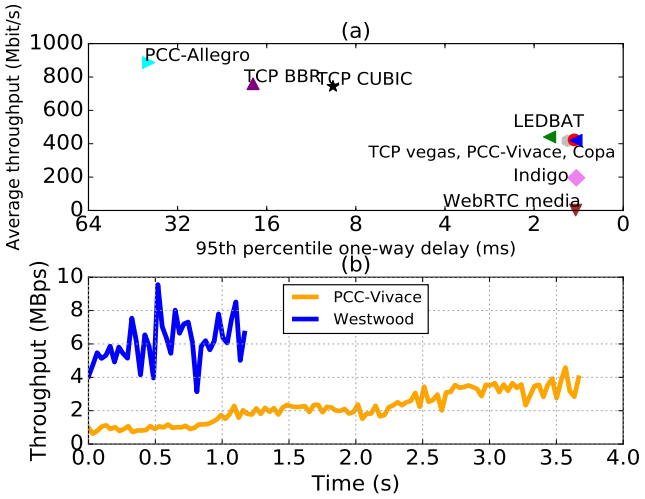

Figure 3: (a). Normal wired network; (b). Throughput of downloading same fie in WiFi

[25]. To understand the landscape of network heterogeneity and consequent impact on CCA performance, we collect dataset from production CDN nodes servers covering over thirty cities across country. Specifically, we instrumented web servers which deliver application software to personal computer clients across cities through CDN, and the last mile of clients' network is wired or wireless. The TCP stack of servers is instrumented to collect transport layer data, including real-time bandwidth, RTT (round-trip time), retransmission ratio, SACK (Selective Acknowledgment), etc. Metrics of part of the dataset is shown next.

One CCA cannot excel in diverse scenarios due to its design assumption can be violated by complex reality. We set the congestion control policy to assign CCA for incoming connection in a round robin manner to attain performance of various CCA in CDN servers. Figure 2 shows CCA performance differentiates on two groups of connections with same IP prefix sampled in our CDN servers, CUBIC [20] performs well for IP prefix 1 but less ideally for IP prefix 2. We further analyze the reasons of performance differentiation. Traditional end-to-end CCA makes assumptions about network bottleneck, and maintains local observables (e.g., packet delivery rate, delay and loss pattern, etc.) from network feedback data, and uses theses observables as signal to infer the bottleneck state and makes rational rate control decisions. For example, CUBIC interprets packet loss as the signal of congestion and decreases sending rate when loss is detected, but this is not always the case. E.g., CUBIC exhibits low throughput when bottleneck buffers are small, due to misinterpretation of loss [13].

2.2 Why Selecting CCA

Existing works which aim to adapt congestion control to heterogeneity have practical issues. To overcome the limitation of classical CCA's dependence of assumptions, machine learning have been applied in congestion control [18, 19, 30, 31, 33]. These approaches has their practical issues and can be categorized in terms of design rationale.

Model training approaches. They [30, 31, 33] generate an optimized rate control policy for given artificial networks (specified by parameterized simulation or real world trace) by training a control model offline. **Practical issue**: it is hard to predefine clients' network parameters or get their network trace in advance (end host is not fixed), and the effectiveness of training algorithm heavily depends on training dataset [32]; besides, run-time overheads of them are remarkably higher than classical CCA [30].

Online learning approaches. They [18, 19] make no assumption about network and adjusts sending rate based on online reward to maximize a utility function of combined performance metrics. **Practical issue**: slowly reaching available resource share due to learning rate control from scratch for each connection. Figure 3(b) shows the throughput of downloading same file in WiFi environment using PCC-Vivace [19] and specialized handcrafted CCA, respectively.

Selecting CCA is another framework which can adapt server-side congestion control to heterogeneity effectively and efficiently for following reasons. (1) Handcrafted CCAs achieve their design goals and perform well in normal scenarios. Typical network environments (e.g., wired, cellular) have been well studied and corresponding CCAs are proposed. Figure 3(a) shows real world CCA test result in a normal wired network [27] which tells handcrafted CCAs achieve their expected performance. For example, CUBIC and BBR [13] achieve high throughput, and Vegas [12] achieves low delay. (2) Meanwhile, machine learning (ML) CCAs cannot magically surpass handcrafted CCAs in all performance metric (e.g., Copa and PCC-Vivace perform similar as Vegas), their advantages lie in: flexibility to balance between different objectives (e.g, high throughput and low delay) by adjusting utility function; online learning CCAs can be effective for rare abnormal environments[29]. (3) Find the mapping of given connection to suited CCA is feasible in servers of datacenters. Unlike traditional end-to-end congestion control which makes rate control decision solely based on run-time network feedback data of a single connection, servers in datacenter possess more information for making congestion control decision since they have a global view across connections on spatial and temporal dimensions (§1). (4) The architecture of decoupling of network environments and suited CCA is effective and efficient. It combines the advantages of traditional CCA and ML techniques, i.e., reusing the human wisdom in traditional CCA to tackle normal network scenarios in a robust and lightweight way, and utilizing pattern recognition power of ML to find Scenarios-CCA mapping. Furthermore, it provides extensibility. New proposed CCAs which tackle emerging environments can be added in our framework, also aforementioned machine learning CCAs can be incorporated. E.g., PCC is incorporated in our framework.

2.3 Basic Idea

To improve overall transport performance, we intend to select suited CCA for each connection so that the CCA match each connection's network characteristic respectively. To achieve this, as shown in Figure 1 , we take following measures for each connection. (1) Setting an initial CCA and collecting network feedback data (e.g., bandwidth, RTT...) from the front packet sequence of the connection; (2) Characterizing the connection by collected data and switching its CCA to the matched algorithm. Correspondingly, Rein is a framework which requires no receiver modification and provides two functions: (1) Exposing network feedback data efficiently in network protocol stack; (2) switching to selected CCA online smoothly according to user-defined selection policy.

2.4 Challenges

Realizing the basic idea requiring efforts on two aspects. (1) System support. The network protocol stack needs to accommodate diverse CCAs and enable online algorithm switch, and collecting data should be lightweight so that normal processing of network protocol stack is uninfluenced. (2) Policy design. The mapping from collected data to matched CCA is needed. Thus, two types of challenges are posed.

System related challenges. (1) Connection states migration is needed when switching to selected CCA from initial CCA, and a smooth migration can be nontrivial. Ideally, the selected CCA should inherit the probed network resource and its internal variables should synchronize with current network snapshot (sending rate, congestion state, etc) just as though the selected CCA has executed from the beginning. But this is hard since different CCAs maintain individualized variables, it is likely that not all variables of the selected CCA can be translated from variables of initial CCA. (2) Characterizing connections online requires analyzing the quickly generated per-ACK (Acknowledgment) feedback data in a real-time manner, but frequent data exchange between network protocol stack and analysis procedure can introduce non-negligible overheads. Experiments (§5.3) show that, for a saturated 10G NIC port, exposing the Linux kernel TCP stack feedback data by a commonly used kernel interface cost more than 10% CPU resource, which is far higher than the original congestion control procedure.

Policy related challenges. The main challenge is how to find the mapping from collected data to matched CCA. In essence, selecting CCA is predicting performance of each candidate CCA on the given connection and adopting the most competent one. This is challenging because the applicable scenarios of CCAs are usually empirical and not specified by detailed network parameters, while collected data are in the form of multivariate time series (MTS, Figure 6) with noise (per-ACK bandwidth, RTT,...) and span a immense range due to the network heterogeneity.

3 DESIGN

Figure 4 shows a high-level schema of Rein, including two key modules: *Selector* and *Agent*, which exchange information through a pair of pipes. *Agent* is responsible for collecting data and switching CCA in network protocol stack, *Selector* maps collected data to suited CCA. *Workflow*: *Agent* collects data and exposes it to *Selector* by upward pipe, *Selector* maps data to suited CCA and send CCA switch notification to *Agent* through downward pipe, then *Agent* parses the notification and launches algorithm switching in network protocol stack.

3.1 Agent

Agent achieves data collection and algorithm switching by instrumenting and restructuring network protocol stack.

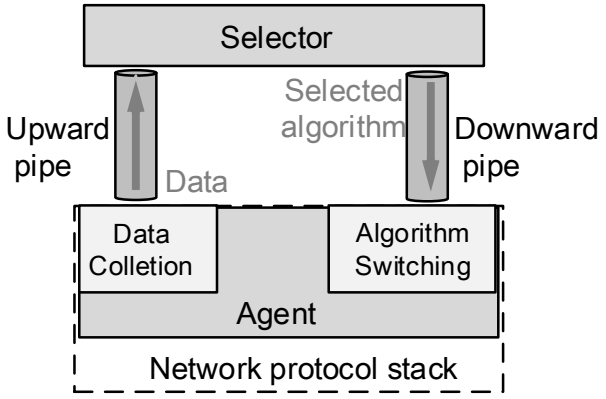

Figure 4: Rein Overview

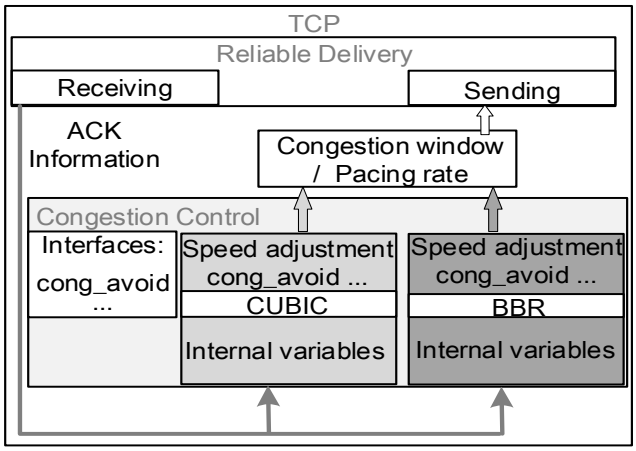

Figure 5: Modularized TCP

Data collection. Data collection function is instrumented in the ACK processing procedure and is invoked when an ACK is received. Firstly, *Agent* extracts the information(e.g., current RTT, loss ratio, bandwidth) and then writes them to the upward pipe where data is read by *Selector* later.

Algorithm switching. Algorithm switching is realized by two measures: replacing CCA on the fly and ensuring the smoothness of CCA transition.

Replacing CCA on the fly. Algorithm replacement is done by replacing the reference to the algorithm [2], with transport protocol implementation is restructured to be modularized and CCA is implemented as pluggable module. We describe this measure based on TCP stack [3] (Figure 5). Specifically, the restructuring including two parts. [10, 16] (1) The congestion control module only computes the sending rate and does not involve in other TCP functions, so that CCA replacing does not effects other TCP components. (2) Pluggable CCA modules share a common congestion state machine (e.g., congestion avoidance, loss recovery) and implement corresponding defined interface of rate adjustment. Taking slow start state for example, all CCAs should implement their own rate control policy for slow start so that the congestion control component can call the interface when a connection enters slow start. In this way, the congestion control component only interacts with the common interface of algorithms and can invoke the related rate adjustment interface in any state. Therefore, algorithm switching can be done by replacing the reference to the algorithm.

Ensuring the smoothness of CCA transition. The state migration for algorithm switching is essentially translating the variables of previous algorithm to the variables of new algorithm, and two kinds of variables are related to smoothness: the sending rate variable (in the form of congestion window or pacing rate) and observed variables (e.g., the minimum observed RTT), because the sending rate directly determines performance, and observed variables are used to adjust sending rate (e.g., BBR computes the sending rate by the maximum observed bandwidth and the minimum observed RTT). We take two measures to decide the initial value of the sending rate and observed variables of the new algorithm. (1) Inheriting the previous algorithm's sending rate to avoid drastic performance degradation. We set the initial congestion window for the new algorithm by inheriting the previous algorithm's evolutional value. As for the algorithms employing the pacing mechanism, such as BBR, we set the initial pacing rate to the value of congestion window divided by recent sampled RTT. And the transition from pacing-rate based algorithm to window-based algorithm is symmetric. (2) Initializing the new algorithm's observed variables to their default value. The reason is that the observed variables are secondary to sending rate variables in metric of performance, and the update frequency of observed variables is high enough to compensate the information loss caused by initializing them to default values. To confirm this point, some typical observed variables of Linux kernel CCAs are listed in Table 1. Most of them are updated per-ACK, and multiple ACKs are received in one RTT, which can generate adequate samples to update observed variables. Besides, initializing observed variables requires no transplant work for algorithms to fit in our framework, because the initialization is done by calling their original initialization function, otherwise, we need to find the transition relation between the two algorithms since observed variables of different algorithms are usually dissimilar. To sum up, the previous algorithm's sending rate is inherited and the observed variables are reset for the new algorithm to ensure the smoothness, and the evaluation result suggests the feasibility of this method in §5.1.

3.2 Selector

The main function of *Selector* is mapping collected data to matched CCA, which is the focus of future work, we briefly describe key ideas. We take two methods to select matched CCA. (1) We first check whether this connection is facing typical network scenario, if so, corresponding specialized CCA is selected. We adopt this because there are typical cases which can be directly identified by extracting features with explicit physical meaning from collected data. For instance, typical wireless connections can be identified (§4.2.2). For those cases, manually designed rules can be applied

[2] In Linux kernel, the reference to CCA is in the form of function pointer, replacement is done by changing the function pointer to the new algorithm.
[3] Our implementation is based on Linux kernel TCP stack, but our design can be applied to other protocol stacks like QUIC [22]

Table 1: Observed variables of classical CCAs

Algorithm	Typical observed variables and their update frequency
CUBIC	dealy_min (UP); last_max_cwnd (UR)
BBR	min_rtt (UA); max_bw (UA)
Vegas	base_rtt (UA); count_rtt, min_rtt (UP)
Westwood[15]	bandwidth_estimation (UC); cumulated_acked (UA)
Illinois[23]	sum_rtt, count_rtt (UP); max_rtt, base_rtt (UP); alpha, beta (UC)

UA:updated when ACK arrives; UR: updated when retransmission timer expired; UP: updated when packets are ACKed; UC: updated when current sending round ends

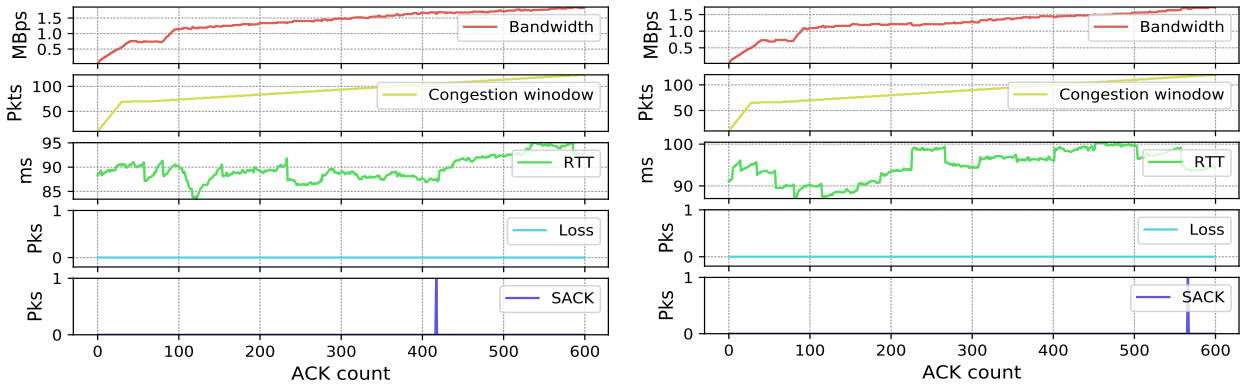

Figure 6: Similar multivariate time series (MTS) data of two connections

to map data to CCA, we call this method rule-based selection [5]. (2) For general cases, we intend to apply data driven method to gain the mapping, based on two initial observations captured in our CDN dataset. (a) For most connections, there exists other connections with similar MTS (Figure 6), due to the network temporal and spatial locality. (b) Connections with similar MTS tend to exhibit similar CCA performance. We find that connections whose MTS match on all attributes (RTT, bandwidth...) tend to face homogenous network environment, and CCA performance difference is relatively stable for these connections, e.g., BBR is better than CUBIC for most cases under this environment. Therefore, new arriving connections' CCA can be selected by consulting CCA performance on past connections with similar MTS. We plan to encode patterns in the MTS data and mine the relation between MTS pattern and CCA performance by data driven method, and we call this method as learning-based selection. We provide a selection policy instance in §4.2.2 to demonstrate *Selector* workflow.

4 IMPLEMENTATION

We implement Rein in Linux by modifying the kernel (4.14.29) source tree, adding a kernel module and writing the associated user-level library. Implementation details of key components of Rein are introduced respectively.

4.1 Kernel Implementation

Rein contains two parts in kernel: Pipe implemented as an external kernel module and *Agent* implemented as modifications to TCP stack.

4.1.1 Pipe. Pipe provides high efficient data exchange between *Agent* and *Selector*, ring buffer is decided as its data structure to support batch processing and lock-free synchronization, with per-core structure. Two kinds of overheads are further minimized: memory allocation/release and data access. Memory pools are pre-allocated for both upward and downward pipe respectively to avoid frequent memory allocation and release. Since the rate of writing data into upward pipe is the same as ACK arriving rate. The data access overheads are decreased by memory mapping. Taking upward pipe for instance, for data is collected in kernel and analyzed by *Selector* in user space, memory mapping avoids the overheads of user/kernel mode switch and system calls for reading and writing data.

Kernel Module. Specifically, pipe is implemented as a special device file: /dev/rein. *Selector* calls `open` system call to open the file to create a suite of pipes, which is also accessible from *Agent*. A handle is returned to *Selector* for manipulating pipes. Besides, the pre-allocated memory pool is allocated by the driver of /dev/rein in the `open` system call. In addition, the driver implements the `ioctl` system call to realize the synchronization between *Selector* and *Agent*, and the memory mapping is achieved by implementing the `mmap` interface of the device. Batch processing is implemented by *Selector* polling the pipes, that is, *Agent* continuously writes collected data to the shared area, where *Selector* periodically reads a batch of data by timer-triggering.

4.1.2 Agent. *Agent*'s functions include data collection and algorithm switching, realized by minor modifications made to Linux

kernel TCP stack. **Data collection**. We instrument the ACK processing procedure to generate raw data, and extract per-ACK information, e.g., RTT, loss rate, delivery rate, ECE (ECN-Echo) mark, etc. More information can be elicited if required. **Algorithm switching**. Dozens of CCAs are implemented in Linux kernel in the form of loadable modules. Changing the CCAs of connections can be realized by replacing function pointers. The switching starts by *Selector* writing algorithm decision of a group of connections in the downward pipe and calling `ioctl`, which notifies *Agent* to set algorithm switching flags for the related connections' TCP control blocks. The flag will directly trigger an algorithm replacement when kernel congestion control component is invoked. The state migration for the replacement is in accordance with §3.1.

4.2 User Space Implementation

4.2.1 User Interface. The user library of Rein is essentially a simple wrapper of the kernel module, which includes `rein_open` and `rein_close` to open/close the device file, and `ioctl` to synchronize *Agent* and *Selector*. The pseudocode in §4.2.2 shows an example of using the interface.

4.2.2 Selector. Selector currently only implements rule-based selection, we provide one instance of differentiating WiFi connections from wired connections to demonstrate the workflow, classification accuracy and more abundant rules will be further explored in future work. The rules is summarized by analyzing connection trace, and find that the jitter of RTT is drastic in WiFi and we quantize it by the coefficient of variation (COV, standard deviation divided by mean) and normalized range ((maximum - minimum)/minimum) of sampled RTTs. The rule is described as following: if the COV and the normalized range of sampled RTTs for a connection exceed certain thresholds (CTH and RTH in the pseudocode) in the first N sampled RTTs, we speculate the connection is through WiFi and set TCP Westwood for this connection.

```
/* Selector */
/* get a handle to manipulate Rein */
handle = rein_open();
/* main loop of selection */
while(true) {
 /* update upgoing pipe for new data */
 ioctl(handle->fd, UPSYNC,NULL);
 /* whether Agent has writen data in the upward pipe */
 while(!pipe_empty(handle->up_pipe)) {
  data = get_data(handle->up_pipe)
  /* get the state of the flow who own the data */
  flow_state = hash_find(data.flow_id);

  /* the rule to classify WiFi in the seciton 4.2.2 */
  if(flow_state.rtt_cnt < N &&
   flow_state.newcc == 0) {
   update_state(flow_state, data);
   if(flow_state.rtt_cov > CTH &&
    flow_state.rtt_range > RTH) {
    flow_state.newcc = westwood;
    /* write switching message to downward pipe */
    put_data(handle->down_pipe, flow_state));
   }
  }
 }
 /* signal Agent to switch algorithm, if any */
 ioctl(handle->fd, DOWNSYNC, NULL);
 /* sleep to achieve batch processing */
 sleep(2ms);
}
```

Listing 1: Pseudocode for WiFi classification

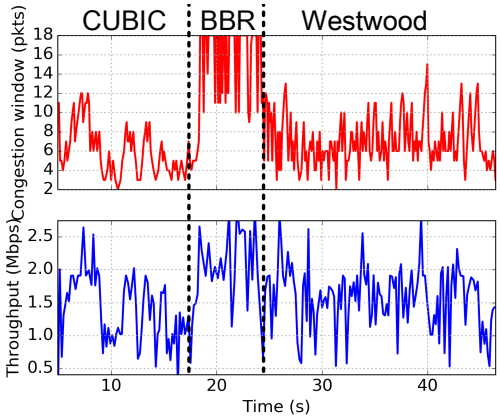

Figure 7: State migration

5 EVALUATION

We answer three questions in this section. Whether smooth and online algorithm switching is feasible? Whether potential performance gain can be achieved by Rein? Whether the overheads are moderate?

5.1 Algorithm Switching

To observe the dynamic behavior of Rein during algorithm switching, we firstly utilize the Linux Traffic Control (TC) [6] to regulate the link characteristics to: 2 Mbps bandwidth, 30ms RTT and 4% loss ratio. Driven by Rein,the CCA is switched from CUBIC to BBR, then to Westwood, which covers the dominated algorithm types, i.e., rate-based/window-based, delay-based/loss-based. TCP probe [1] is then used to trace the following metrics: sender congestion window and throughput. The results are shown in Figure 7. Obviously, the smooth and online algorithm switching can be conducted by Rein.

5.2 Performance Gain

We show that Rein averagely outperforms CUBIC by 7.43%, BBR by 57.0% and Westwood by 17.16% in the heterogeneous network by creating three network scenarios: clients with WiFi access and clients with wired access(small and large bottleneck buffer). The server runs four Nginx [8] web servers with Rein, BBR, CUBIC and Westwood, respectively, three client machines send requests to all four web servers to download 30MB files. One client machine accesses a public WiFi, the other two clients travel through two emulation networks configured by Linux traffic control tool [6] in a middle box as in work [14]. The parameters of two emulation network are: propagation delay 40 / 40 ms, bandwidth 3 / 3 MBps, buffer size 1BDP / 4BDP (bandwidth delay product), and background traffic is running CUBIC as competing flows. The initial algorithm of Rein is CUBIC, and the rule developed in §4.2.2 is used to differentiate WiFi connections from wired connections, small and large buffer scenarios are separated by setting a threshold on the 95th percentile RTT sampled in data collection phase. Figure 8 shows the cumulative distribution function of flow completion time of Rein, CUBIC, BBR and Westwood. In the wired-small buffer

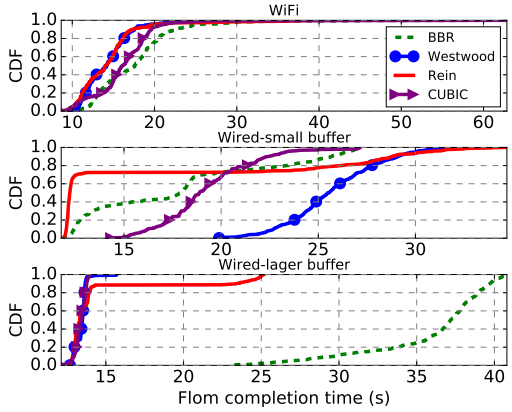

Figure 8: CDF of flow completion time

Table 2: CPU cost percentages comparison of Rein and netlink

	User space (%)		Kernel (%)	
Rein	*Selector* 0.54	Nx 29.24	*Agent* 0.08	`tcp_ack` 0.38
netlink	n_user 11.83	Nx 21.13	n_kernel 0.54	`tcp_ack` 0.71

environment, Rein runs CUBIC first and switch to BBR later, and surpass other schemes due to BBR is advantages versus loss-based CCA in small buffer scenario [13, 21]. In the wired-large buffer scenario, Rein runs CUBIC and is superior to BBR due to BBR compete inferiorly against CUBIC when bottleneck buffer is large [21]. In WiFi environment, Rein switch to Westwood after data collection and perform better than CUBIC and BBR since Westwood matches wireless characteristics. Rein performance is not completely same as the Westwood due to scarce cases of algorithm misselection (selection policy failed due to randomness in collected data). The average flow completion time across three environments of Rein, CUBIC, BBR, Westwood are 15.21s, 16.34s, 23.88s, 17.82s, respectively.

5.3 Overheads

In this section, we test the overheads of Rein in high-load server. To validate our system optimizations, we compare Rein with netlink socket [4], which is designed and commonly used for transferring miscellaneous networking information between the kernel and user space processes. We set up one server machine and one client machine, which are both equipped with a 12-core CPU (Intel Xeon E52620 @ 2.4GHz), 64GB RAM and one dual-port Intel 82599 10G NIC, respectively. The 10G port of 82599 NIC in the server is directly connected to client machines' NIC port. We stress the CPU by combining the multi-queue of 10G NIC into a single queue. Then Rein is applied to Nginx (Nx) [8] and ensure Rein, Nginx, kernel network stack all run in the same core. To identify the overheads caused by the framework itself, *Selector* only pre-processes data by computing average RTT, loss rate, and throughput. The client machine firstly regulates the RTT to 20 ms, then 250 concurrent persistent connection are launched in the client using an HTTP benchmark tool (wrk) [7] to download 1MB files, which nearly saturates the 10G NIC port. We replace Rein's pipes and related enhancements with netlink socket as the comparison experiment. We then profile the CPU cycles cost proportions by performance profiling tool [2] to evaluate the overheads of Rein and netlink socket. As listed in Table 2, Rein contains two parts, *Selector* in user space and *Agent* in kernel. To make the CPU cost proportion value more concrete, we compare Rein's overheads with Nginx and `tcp_ack` procedure. [4] The netlink socket also includes two parts, n_user, which reads data from kernel, and n_kernel, which is instrumented into `tcp_ack` to extract data. The result shows that the overheads introduced by Rein are modest while netlink's cost is nonnegligible.

6 RELATED WORK

ML related congestion control. Some works [30, 31, 33] adopt model training approaches and generate an optimized rate control policy for a given artificial network (specified by parameterized simulation or real world trace) by training a control model offline. Others [11, 18, 19] take online learning approaches by make less assumption about network and adjusts sending rate based on online reward to maximize a utility function of performance metrics. Their control unit is more subtle (e.g., sending rate), while Rein's control unit is CCA. While work [26] explores the idea of selecting CCA through reinforcement Learning, its model is based on averaged network feedback data from emulation, which is prone to noise and variability in the wild network [32]. Rein aims to utilize real-world multivariate time series network feedback data.

Fine-grained configurable network protocol stack. There has been a surge on making network protocol stack customizable to adapt to clients' characteristics by configuring protocol parameters. Configtron[25], PQUIC[17], CCP[24] and Linux eBPF[3]. Rein focus on smooth algorithm switch to adapt to heterogeneity.

7 CONCLUSION

We argue that, instead of applying a single congestion control algorithm for connections with heterogeneous network environments, preferable algorithm should be selected according to the characteristics of each connection. We design and implement the prototype of Rein to support our argument, it aims to improve performance by exploiting the heterogeneity of clients' network. Preliminary evaluations confirm the feasibility of Rein. Future work will focus on designing CCA selection policy and deploying Rein in our CDN server.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by the National Key Research and Development Program of China (No.2018YFB1700203), and by National Natural Science Foundation of China (NSFC) under Grant 61872208 and Grant 61902307.

REFERENCES

[1] 2015. Linux TCP Probe. https://wiki.linuxfoundation.org/networking/tcpprobe.
[2] 2015. Perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
[3] 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[4] Nginx runs in user space as application while `tcp_ack` is the original kernel procedure we add *Agent* in.

[4] 2018. Netlink. https://en.wikipedia.org/wiki/Netlink.

[5] 2018. Wiki: Rule-based system. https://en.wikipedia.org/wiki/Rule-based_system.

[6] 2019. Linux Advanced Routing and Traffic Control. https://en.wikipedia.org/wiki/Tc_(Linux).

[7] 2019. Modern HTTP benchmarking tool. https://github.com/wg/wrk.

[8] 2019. Nginx. http://nginx.org/.

[9] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock. 2010. Host-to-Host Congestion Control for TCP. *IEEE Communications Surveys Tutorials* 12, 3 (Third 2010), 304–342. https://doi.org/10.1109/SURV.2010.042710.00114

[10] Somaya Arianfar. 2012. TCP's congestion control implementation in linux kernel. In *Proceedings of Seminar on Network Protocols in Operating Systems*. 16.

[11] 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.

[12] Lawrence S. Brakmo and Larry L. Peterson. 1995. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.

[13] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *Queue* 14, 5 (2016), 50.

[14] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. 2016. Analysis and Design of the Google Congestion Control for Web Real-Time Communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2910017.2910605

[15] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, and Ren Wang. 2002. TCP Westwood: End-to-end Congestion Control for Wired/Wireless Networks. *Wirel. Netw.* 8, 5 (Sept. 2002), 467–479. https://doi.org/10.1023/A:1016590112381

[16] Yuchung Cheng and Neal Cardwell. 2016. Making Linux TCP Fast. In *Netdev Conference*.

[17] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 59–74. https://doi.org/10.1145/3341302.3342078

[18] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 395–408.

[19] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[20] 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.

[21] M. Hock, R. Bless, and M. Zitterbart. 2017. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. 1–10. https://doi.org/10.1109/ICNP.2017.8117540

[22] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842

[23] Shao Liu, Tamer Başar, and R. Srikant. 2006. TCP-Illinois: A Loss and Delay-based Congestion Control Algorithm for High-speed Networks. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools (valuetools '06)*. ACM, New York, NY, USA, Article 55. https://doi.org/10.1145/1190095.1190166

[24] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring Endpoint Congestion Control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 30–43. https://doi.org/10.1145/3230543.3230553

[25] Usama Naseer and Theophilus Benson. 2017. Configtron: Tackling network diversity with heterogeneous configurations.. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)*.

[26] X. Nie, Y. Zhao, Z. Li, G. Chen, K. Sui, J. Zhang, Z. Ye, and D. Pei. 2019. Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning. *IEEE Journal on Selected Areas in Communications* 37, 6 (2019), 1231–1247.

[27] Pantheon. 2019. 201809 Stanford to AWS-California. https://pantheon.stanford.edu/measurements/node/.

[28] Sandvine. 2018. The Global Internet Phenomena Report 2018. https://www.sandvine.com/hubfs/downloads/phenomena/2018-phenomena-report.pdf.

[29] Michael Schapira. 2018. Network-Model-Based vs. Network-Model-Free Approaches to Internet Congestion Control. In *Proceedings of IEEE International Conference on High Performance Switching and Routing (IEEE HPSR '18)*.

[30] Wei Wang, Yiyang Shao, and Kai Zheng. 2018. Muses: Enabling Lightweight and Diversity for Learning Based Congestion Control. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos (SIGCOMM '18)*. ACM, New York, NY, USA, 141–143. https://doi.org/10.1145/3234200.3234202

[31] Keith Winstein and Hari Balakrishnan. 2013. TCP ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 123–134.

[32] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 495–511. https://www.usenix.org/conference/nsdi20/presentation/yan

[33] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/yan-francis