Antelope: A Framework for Dynamic Selection of Congestion Control Algorithms

Jianer Zhou*†, Xinyi Qiu†, Zhenyu Li‡†, Gareth Tyson§, Qing Li†, Jingpu Duan*†, Yi Wang*†
*Southern University of Science and Technology, Shenzhen, China †Peng Cheng Laboratory, Shenzhen, China
‡ ICT, CAS, China §Queen Mary University of London
{zhouje, duanjp, wangy37}@sustech.edu.cn, {qiuxy, liq}@pcl.ac.cn, zyli@ict.ac.cn, g.tyson@qmul.ac.uk

***Abstract*—Most congestion control mechanisms are designed for specific network environments. Hence, there is no known algorithm that achieves uniformly good performance in all scenarios for all flows. Rather than devising such a one-size-fits-all algorithm, we propose a system to dynamically switch between the most suitable congestion control mechanisms for specific *flows* in specific *environments*. This raises a number of challenges, which we address through the design and implementation of *Antelope*, a system that can dynamically reconfigure to use the most suitable congestion control mechanism for an individual flow. We build a machine learning approach to learn which algorithm works best for individual conditions and implement kernel-level support for dynamically adjusting congestion control algorithms. We have implemented Antelope in Linux, and evaluated it in both emulated and production networks. We show that in WAN, DCN, and cellular networks, Antelope achieves an average 16% improvement in throughput compared with BBR; compared with Cubic, Antelope achieves an average 19% improvement in throughput and 10% reduction in delay.**

I. Introduction

Since the birth of TCP, many congestion control (CC) mechanisms have been proposed [28] [14] [10] [15] [30] [8]. However, none of these *individual* mechanisms can achieve high network performance across all environments and user requirements. Two reasons account for this. First, each algorithm is generally designed for a particular environment. For example, Sprout [33], C2TCP [3] [2] and Verus [36] are designed for cellular networks; DCTCP [6], pFabric [7] and Swift [18] are designed for datacenter networks; TACK [19], HACK [31] and Westwood [20] are designed for wireless local area networks (WLANs). Second, network environments and application requirements have evolved over decades. For example, when Cubic [14] (the default Linux TCP mechanism) was proposed, improving bandwidth utilization was the most important goal. However, for modern gaming or live streaming applications, latency is much more critical. Our goal is therefore to devise a congestion control framework that can achieve good performance across all environments and requirements, with sufficient flexibility to evolve over time.

In pursuit of this goal, machine learning based CC mechanisms have been proposed. These strive to autonomously learn the optimal CC policy for any given scenario. For example, RemyCC [32] uses the network parameters, user behavior, flow model and target function as an input, then derives an appropriate sending rate as an output. Similarly, PCC-Vivace [12] uses online learning, while DeepCC [4] and Orca [5] use deep reinforcement learning (DRL) to adjust their sending rates based on network feedback. However, deploying such machine learning based CC mechanisms in a production network has proven complicated, as it is necessary to continually learn for each environment. Thus, applying such models in unseen networks decreases their performance [16]. Our goal is to *devise a congestion control framework (based on the CC mechanisms available in Linux kernel) that can achieve good performance across all networks and application requirements, while avoiding the complicated deployment issues introduced by other machine learning mechanisms*. To achieve this target, we propose a simple yet effective framework called **Antelope**.

Antelope adjusts the congestion control algorithm for individual flows according to network and flow state observed. It collects TCP flow information from the kernel data-path and delivers the data to user space, where we can exploit pre-existing machine learning libraries. Using supervised classification, Antelope then predicts which congestion control algorithm could achieve the best performance for that particular flow. Antelope then continues to monitor the flow and changes the CC algorithm dynamically if the network environment or flow state changes. To coordinate this, Antelope uses eBPF (a new kernel function which support more control in kernel from user space) [22] [9] to deliver information between the user space and kernel. Through extensive experiments on emulated and real networks, we demonstrate that Antelope can nearly always choose the most suitable mechanism for each flow. Our key contributions are:

- We design and implement Antelope, an adaptive CC framework which dynamically reconfigures between the most suitable CC algorithm on a per-flow basis. Antelope only needs the change at TCP senders without changing the TCP socket. As such, Antelope can easily be deployed in a production environment and the source code is available for the community.[1]
- As part of Antelope, we build and train a supervised classification algorithm (in user space) that can select suitable CC mechanisms for flows that have similar

Jianer Zhou and Xinyi Qiu are co-first authors.

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

[1] https://github.com/antelopeproject/antelope

patterns with the training data, but also on novel flows that have not appeared before. We show that eBPF, as part of Antelope, is an effective choice to manage CC algorithms in the kernel, even after a TCP flow has been established.
- Extensive experiments in WAN, DCN, and cellular network show that Antelope achieves an average 16% improvement in throughput compared with BBR; compared with Cubic, Antelope improves the throughput by 19% on average, and reduces delay by 10%. Further, Antelope shows better performance than the state-of-the-art ML-based mechanisms (Orca and PCC-Vivace).

II. Motivation

A. Why switch CC mechanisms?

Network environments impact TCP flows. Servers that perform data transfer services (*e.g.* web servers) will usually deal with TCP flows from diverse network environments. This may be due to a diversity of clients or because a server has multiple responsibilities. For example, a front-end server may receive client requests (*e.g.* from a 4G network), yet retrieve content from a back-end server situated in the same data center (*e.g.* via Ethernet). Whereas the Ethernet path will support high bandwidth and low delay delivery, the 4G path will likely suffer from much higher levels of delay and bandwidth fluctuations. Using a single network stack with a shared CC algorithm therefore forces administrators to select which environment to optimize for.

To highlight this, Figure 1 shows a toy example of the TCP throughput for different CC mechanisms over datacenter, cellular and wide area (wired) networks. This is done using the Mahimahi emulator [25], parameterized as follows. The cellular network is configured using the public trace data from [5]; the WAN is setup with an RTT, packet loss and bandwidth of 100ms, 2% and 2MB/s, respectively; the DCN is setup with 1ms, 0.1% and 1GB/s, respectively. We see that for the DCN network, both the short and long TCP flows have the highest throughput when using BBR. For the cellular network, when using C2TCP, the long flows' throughput is the best; in contrast, for short flows, Westwood is the best. For long flows over the WAN, using Cubic is the best, but for short flows BBR has the highest throughput. Despite this, most front-end servers use Cubic or BBR to serve all TCP flows [10]. In other words, there is no one-size-fits-all algorithm.

Network environments are dynamic. Complicating matters further, network environments may change on the fly. For example, in the public cloud, it is common for flows to change paths at ten-second intervals or even faster [29]. Alternatively, when more cellular users pair with a base station, the buffer provided to one user becomes smaller. This will impact performance, *e.g.* BBR obtains higher throughput with small buffers [10]. Alternatively, ISPs may adjust their network paths (*e.g.* via MPLS or SDN), changing existing flows' RTTs and buffer sizes. Under such conditions, switching the TCP flows' CC may improve performance.

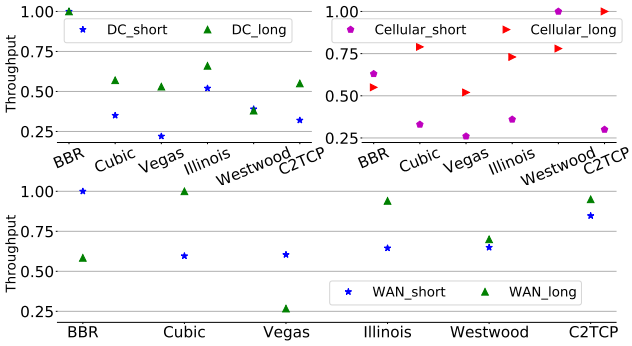


Fig. 1: Performance of different CC mechanisms over 3 different networks.

Machine learning CC mechanisms are limited. Rather than adjusting the congestion window or pacing rate using ML, we build a model to select the CC algorithms on a per-flow basis. We do this for two reasons. First, as pointed out by both Orca [5] and Rein [11], learning-based approaches (*e.g.* Indigo [35], Aurora [16]) suffer from performance degradation and slow convergence when used in unseen conditions. In contrast, hand-written classic CC algorithms do not have these two issues. Second, classic CC algorithms that have been widely used in practice, often achieve very good performance in the network environments for which they are designed (*e.g.* Westwood [20] for wireless networks).

B. Challenges

Selection of CC algorithm. Antelope must design an appropriate reward function to select the optimal CC algorithm for a given scenario. However, the TCP parameters alone (*e.g.* RTT, CWND, in_flight and lost packets) are inherently limited in their ability to predict throughput, fairness, delay etc. Solely relying on these parameters to decide the optimal CC algorithm is therefore not wise. Furthermore, manually selecting CC algorithms, even with machine learning support, is difficult for network operators and domain specialists [21]. This is exacerbated by dynamic network conditions, which may invalidate historical data used to make such decisions.

Short flows. If a machine learning approach is taken, as the duration of many flows is short, they may finish before it is possible to learn which CC algorithm would have been most suitable. Antelope must be able to rapidly select the most suitable CC algorithm.

Kernel vs. user space. The kernel lacks machine learning libraries. Thus, we argue it is necessary for Antelope to implement any machine learning technology in user space, and enable flexible interaction between user space and the kernel. Limiting the overhead and delay for such communications is challenging.

III. Antelope Overview

Overview. The duration of a TCP flow can be divided into three phases: connection setup, data transmission and con-

nection closure. Different actions will be performed in these three phases by Antelope. After the connection setup, the Information Collection component (in the kernel) will collect TCP flow information and deliver it to the Mechanism Match component (in user space). Then during the data transmission phase, the Mechanism Match component (periodically) selects the most suitable CC mechanism according to the flow's characterisics. The most suitable CC mechanism will then be passed to the Mechanism Switch component (in the kernel) which will switch to that CC mechanism in the network stack. When a connection closes, both the Mechanism Match and Mechanism Switch components will delete this flow's records. The overall architecture is shown in Figure 2.

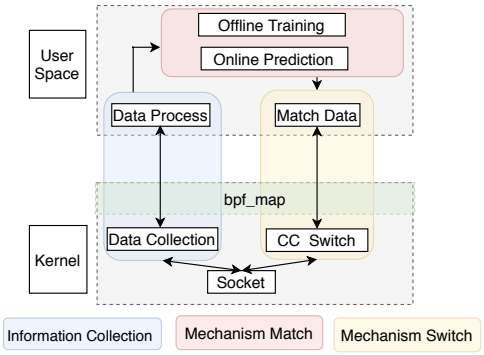


Fig. 2: High-level components of Antelope.

Information Collection. The Information Collection component consists of two sub-modules: the Data Collection module and the Data Process module. The Data Collection module runs in the kernel. It collects all TCP flow information and then delivers it via eBPF to the Data Process module, which is in user space. The Data Process module aggregates and formats the data before passing it to the Mechanism Match component. In Section V we will show how we collect the information.

Mechanism Match. The Mechanism Match component consists of two sub-modules: Online Prediction and Offline Training modules, both of which are implemented in user space. When TCP information is delivered to the Mechanism Match component, it will dynamically select the most appropriate CC algorithm to use. This will then be recorded to the bpf_map structure and made accessible in the kernel. The Online Prediction module relies on several trained models for selecting different mechanisms, and will return the most suitable one according to the scores generated by each models. To inform this process, the Offline Training module will train the matching model using a reward function. Specifically, we build a decision-tree model using XGBoost. The details of this component are shown in Section IV.

Mechanism Switch. When the Mechanism Match component selects the most suitable mechanism, it will record the flow identifier (by IP and port) and the corresponding CC mechanism. Using eBPF, the information is delivered to the kernel. Then the Mechanism Switch component (in the kernel) will switch to the selected CC algorithm. This process is hooked into three Linux kernel functions: tcp_setup, tcp_sendmsg and tcp_close. In the tcp_setup and tcp_sendmsg functions, the hook monitors the bpf_map and will switch the CC mechanism if instructed. In tcp_close, the hook function just sends flow closing signals to the Mechanism Match and Mechanism Switch components.

IV. Prediction and Training

A. Prediction Module

Overview. The Online Prediction module is the heart of matching process. It selects a suitable CC mechanism based on the TCP flow features. Figure 3 shows the overview of the Online Prediction module. It consists of three main modules, which we describe below: the Statistics Module, Reward Module and Selection Module. As an input, Antelope takes

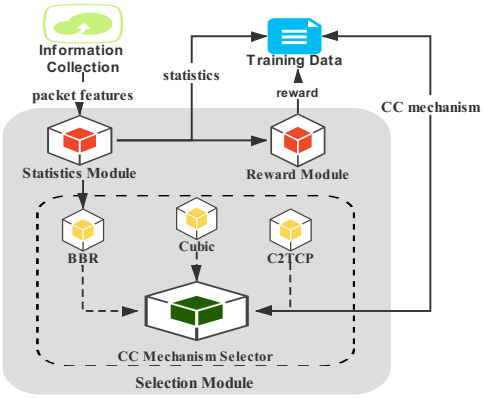


Fig. 3: Online Prediction module overview.

a set of N contiguous ACK packets (in the order that the ACK packets arrive). We refer to this set of packets as a *data unit*. The CC mechanism selection is then performed on the granularity of each data unit. Once N packets are recorded, the information is passed to the Selection Module and Reward Module. The Selection Module is composed of multiple prediction models for different CC mechanisms. By comparing the reward predictions made by each model, the best CC mechanism is selected. When the next data unit is generated, the Reward Module analyzes its statistics to evaluate the effect of the last switching CC mechanism (later used for re-training).

Statistics Module. The Statistics Module is responsible for gathering flow information. It does this by reading packets' information from the Information Collection component. Let d_t denote the t^{th} data unit in a stream, and s_t refer to the statistics of d_t. For every data unit (every 20 packets by default) we calculate the statistics based on the features that the ACK packet carried. We set the default parameters via experimental experience. We set the data unit size as a tradeoff between computational overhead and effectiveness. The statistics are shown in Table I.

TABLE I: Statistics generated by the Statistics Module

Category	Meaning
sRTT_avg	The average smoothed RTT.
number	The number of ACK packets.
lost	The number of lost packets.
time	The time to construct data block.
pacing_rate_max	The maximum pacing rate so far.
throughput	The average sending rate.
delay_min	The minimum packet delay so far.

The Statistics Module continuously calculates the statistics for each data unit and stores them in memory. When the Selection Module receives the statistics of d_t, it predicts the CC mechanism that d_{t+1} needs to use. The reward calculated by the Reward Module is then used to provide feedback on the effect of d_t's prediction. In this paper, we define r_t as the reward calculated using the statistics of d_t. So, the final state (*i.e.* the training data for the prediction model) at step t becomes the vector $train_t = (s_t, r_{t+1})$.

Reward Module. This module is responsible for calculating the effectiveness of a given CC algorithm, and returning a predicted reward. As previously mentioned, this is stored in the Statistics Module and later used by the Selection Module to choose the CC algorithm for the next period.

In order to quantify the performance of each CC mechanism, we define the normalized reward function as Eq 1:

$$\hat{R} = R/R_{max} = (\frac{throughput - \eta * loss}{delay'})/(\frac{pacing\_rate_{max}}{delay_{min}}) \quad (1)$$

Giessler [13] showed that the effectiveness of a CC mechanism can be measured by a metric called Power, defined as Power = $\frac{throughput}{delay}$. It has been shown that when the power reaches the maximum value, not only the network but also the individual flows are in their best state. Our reward function (as shown in Eq 1) is therefore based on the definition of Power. We also incorporate loss as a parameter to adjust the reward function, in order to minimize the packet loss. When computing the reward function, we set the unit of throughput as Kbps, the delay as ms, and the loss as number of lost packets (in one data block interval). η is a parameter that determines the weight of packet loss to reward function. In our current implementation, we empirically set it as 1.

Although Power captures the ultimate goal of the congestion control algorithm (maximizing throughput while minimizing the delay), in practice it is hard to obtain the maximum throughput and the minimum delay at the same time. Furthermore, the sensitivity of streams of different sizes to throughput and delay varies greatly. For example, large flows are usually throughput sensitive, but small flows are more concerned about delay. To address this, we add the coefficient $\delta(\geq 1)$ into Eq 2 (which defines $delay'$ used in Eq. 1):

$$delay' = \begin{cases} delay_{min} & (delay_{min} \leq delay \leq \delta \times delay_{min}) \\ delay & o.w. \end{cases} \quad (2)$$

After the TCP connection setup completes, δ will be initialized to 2. As packets are received by the Information Collection component, δ will increase exponentially with the number of data units. For example, δ is 2 for the first data unit, 4 after the second data unit etc. This means that the reward function will change from delay sensitive to throughput sensitive when more packets are sent in the flow. Orca [5] uses a similar approach.

Selection Module. This module is responsible for retrieving the reward predictions across the set of available CC mechanisms (for a given flow) and then selecting the optimal one. However, short TCP flows may finish before it is possible for the Reward Module to calculate the prediction. Thus, we use two types of predictions: (1) A *stream-level* prediction which predicts the most suitable CC mechanism for this flow by analyzing realtime information (suitable for long flows); and (2) An *IP-level* prediction which uses historical information about prior stream from that IP address (suitable for short flows). We describe these below.

Algorithm 1 Stream-Level Prediction Algorithm

```
1:  function STREAMPREDICT(statistics)
2:      maxReward ← 0
3:      predict_cc ← NULL
4:      // cc_model_map stores prediction models of each CC
5:      for cc in cc_model_map.keys do
6:          predict_model ← cc_model_map[cc]
7:          reward < −predict_model.predict(statistics)
8:          if reward > maxReward then
9:              maxReward ← reward
10:             predict_cc ← cc
11:         end if
12:     end for
13:     return predict_cc
14: end function
```

Stream-level prediction. The stream-level prediction's pseudocode is shown in Algorithm 1. In Antelope, we train a model for each algorithm independently so that we can easily extend the system to new CC algorithms. At each step t, the Selection Module observes the statistics (s_t), and then selects the CC mechanism with the highest predicted reward. The calculation of predictions is described in Section IV-B, where we rely on XGBoost decision trees. Figure 4 shows the architecture of the decision tree. The number of layers in the decision tree depends on the complexity of the training data. Put simply, when, for example, we want to predict BBR's reward for one stream, we input the flow information to the BBR prediction model (which contains some regression trees). For each regression tree, we get the predicted reward, and then we add up all the rewards to get the final result. The reward

can be obtained both after a CC algorithm is deployed and prior using offline training.

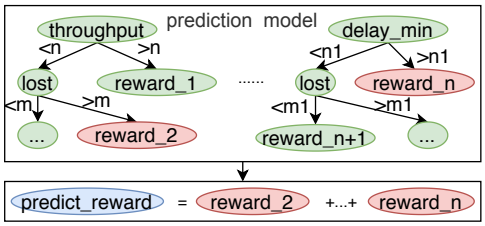


Fig. 4: Architecture of the decision tree for prediction.

.

IP-level prediction. In IP-level prediction, Antelope selects the CC mechanism based on the historical results of the streams belonging to the same IP or a /24 segment. This allows Antelope to select an appropriate CC mechanism before a flow has been initiated. Algorithm 2 presents the IP-level prediction pseudo code. For each IP range, Antelope records the number of times that each CC mechanism has been chosen in the flows to that IP space. In order to adapt to changes in the network, each time a new stream level prediction is obtained, the IP prediction result will be merged with the current prediction results. The historical data is multiplied by the coefficient $\alpha(0 < \alpha < 1)$. Finally, the CC mechanism that has been chosen most frequently with the highest reward is selected.

Algorithm 2 IP-Level Prediction Algorithm

```
function IP PREDICT(ip, cc_mechanism)
    cc_count_map ← ip_CCs_map[ip]
    //Reduce the weight of all historical data.
    for cc in cc_count_map.keys do
        cc_count_map[cc] ← cc_count_map[cc] * α
    end for;
    cc_count_map[cc_mechanism] += 1
    //Choose the cc mechanism with the largest count.
    cc_predict ← getMaxCountCC(cc_count_map)
    //Update the IP and cc mechanism in bpf_map.
    updateBpfMap(ip, cc_predict))
end function
```

B. Training Module

The above relies on a trained model that can predict the reward for a given flow using each CC algorithm available. For training the XGBoost model, we perform both offline and online training.

Offline training. We initiate training in an offline fashion, where we trigger clients to connect to the server, which then randomly selects different CC algorithms to use. This can be done in an emulated environment, as we show in Section VI. The servers collect statistical information (s) and the corresponding ground-truth reward (r). The reward result (r_{t+1}) represents the reward of the mechanism for the $t+1$ data unit. This provides the training instance for data unit t in a tuple (s_t, r_{t+1}). We then use this to train a XGBoost model to predict the correct reward based on the observed statistical information in the previous data unit.

Online training. The previous step creates a pre-trained model for each CC algorithm. We then continue the training in an online fashion by continually computing the real reward to measure the accuracy of the predictions in-the-wild. The reward result and the TCP stats (s_t, r_{t+1}) for the chosen CC mechanism are appended to the training data and are used for periodic re-training. The above training is per-CC not per client-server pair. That said, the trained models are independent to clients and servers and can be reused by other servers.

V. Implementation

We have implemented Antelope in both user space and the Linux kernel (CentOS 8 with kernel version 4.18). We collect TCP flow information from the kernel and then share it with user space (via eBPF), where Antelope uses it to select the most suitable CC mechanism. The suitable CC mechanism for this flow is then delivered back to the kernel using bpf_map. Antelope then switches the CC algorithm in the kernel. An overview of the implementation is shown in Figure 5.

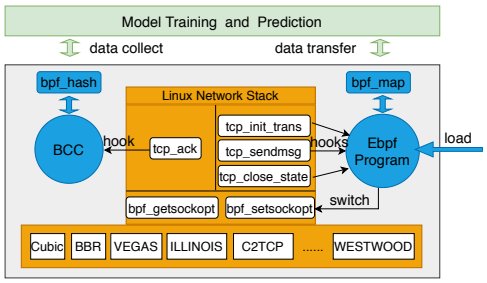


Fig. 5: Overview of the Antelope implementation.

A. Collecting flow information

We use the BPF Compiler Collection (BCC) probe function to get the TCP flow information [1]. We extract the information from struct sock in the kernel. BCC sets different hook functions in the Linux network stack, which means we can get information from different hook points. In our system, we set a hook in the tcp_ack function.

The basic unit we collect is the TCP flow and we distinguish different flows by the saddr, daddr, lport and dport. In every flow, we collect srtt, mdev, min_rtt, packets_out, lost, total_retrans, pacing_rate and TCP state, which are all recorded in the struct sock for this flow. For every ACK that arrives, the hook will be triggered and the information will be delivered to user space via eBPF.

B. Exchanging Information by ebpf_map

To transmit flow information from the kernel to user space, we use the ebpf_hash. To deliver the suitable congestion mechanism from user space to the CC Switch module in the kernel, we use the bpf_map. The suitable congestion

mechanism delivered by ebpf_map is formatted as a key-value pair: IP+flow ID $\rightarrow$ CC mechanism. As at the beginning of a flow, there is not enough information to predict the best mechanism, we select the default mechanism or the one based on the historical information associated with that IP.

C. Switching TCP in the kernel

We use eBPF to switch TCP mechanisms in the kernel. To run Antelope, the compiled eBPF program is loaded into the kernel first. In the eBPF program we use the bpf_getsockopt and bpf_setsockopt in the tcp_ebpf library to switch to the corresponding mechanism [9]. We set three hook points in the kernel to trigger the switching process: tcp_init_transfer, tcp_sendmsg, tcp_close_state. For the tcp_init_transfer hook, the eBPF program will set the new mechanism according the flow's IP or the default one as we explained in Section V-B.

For the tcp_sendmsg hook, we set the new congestion control mechanism according to the prediction. At the end of the flow, the hook point in tcp_close_state will delete the key-value item for this flow. Since we use an eBPF program, when we run Antelope and add a new ability to the kernel, it is unnecessary to rebuild the kernel or to reboot the system.

In the Online Prediction module, once N ACK packets are received, the prediction process is triggered (by default, $N = 20$). If the prediction process finds another suitable mechanism for this flow, it updates the ebpf_map, adding the IP+flow ID $\rightarrow$ congestion mechanism item in the map. If the new mechanism is the same as the old one, the item will be set as empty. At the tcp_sendmsg hook point, the eBPF program will check the map. If it gets the name of a new congestion mechanism in the map, the eBFP program will set this flow's congestion control algorithm to the new one. To avoid switching the mechanism too frequently, we only switch upon seeing M (default 2) consecutive recommended changes.

Antelope can switch between CC mechanisms that are implemented in mainstream Linux kernel, currently including BBR, Cubic, C2TCP, Vegas, Illinois and Westwood. Regardless of whether competing TCP flows use Antelope, individual flows may use different CCs. Thus, Antelope inherits the fairness of the chosen CC algorithm(s). For example, if Antelope chooses BBR, then it will take a larger share of the bottleneck bandwidth than Cubic in shallow-buffered network.

VI. Training and Experimentation

In this section we describe the training process of Antelope and then show the effectiveness of Antelope. Training and evaluation are based on both an emulated environment and production networks.

A. Testbeds

For both training and evaluation, we rely on a network emulator and a real world deployment. We first describe their setups here and delineate the specifics later when presenting the results.

Emulated testbed. We use Mahimahi, a network emulation tool which can evaluate different network environments either (1) by configuring the delay, bandwidth and queue parameters; or (2) by replaying packet behaviour from a real network [25].

We setup two client processes connected to two servers, and direct all of their flows via Mahimahi. One client sends requests to one server and then the server sends files back. To produce background traffic, the other client sends requests to the other server. All of the requests use TCP and go through the same Mahimahi network. The file sizes are randomly chosen (see later). We change the size of request to emulate different background traffic effects.

Real network testbed. To test Antelope in a more realistic context, we also run it over a production network. We install Antelope on a public cloud (at several locations). We place server instances in Asia, North America, Europe and the Middle East. Each instance runs the same file server software used in the emulated testbed. We then issue requests from our campus in Shenzhen, China.

B. Training

To evaluate Antelope, we must first train its prediction model. The training data obtained through the emulated environment helps us construct the initial prediction model, and then the feedback from the real-world experiments supports the optimization of the model.

1) Emulated Training: We test more than 30 network environments using Mahimahi (their characteristics are shown in Table II). We emulate a WAN with low bandwidth and a large RTT; and a DCN using high bandwidth and a small RTT. We also use 6 cellular LTE traces provided in Mahimahi to test cellular network environments.

We use BDP (Bandwidth*RTT) to describe the size of the queue buffer. In our emulated network environment, we set the 5*BDP in WAN and 0.1*BDP in DCN, following the setting in [7]. In the cellular network we do not set its BDP as it is emulated by the traces [25].

We generate request flows of different sizes (flow size between 1KB and 50MB). The training procedures for each parameter combination are repeated 3 times. As this is for training purposes, we test all CC algorithms for each setup. Specifically, for every network environment, we set the sender to a fixed CC mechanism then randomly switch to other mechanisms to observe their performance. We then use this data to train Antelope's initial XGBoost model. It should be noted that the environment we use to train is different from the environment for the performance evaluation (in Section VI-C).

TABLE II: Range of evaluated environments during the training.

BW (MB/s)	RTT(ms)	BDP	Back. traffic
1.4-1000	1-100	0.1-6	1KB-50MB

2) Real World Training: After the initial training performed within the emulated environment, we further train Antelope in our real network testbed. We envisage this to be the de facto approach: each server will start with a generic pre-trained model, and then iteratively improve it in an online fashion.

We train using both inter- and intra-continental scenarios by locating clients and servers in two continents and in the same continent, respectively. Clients (located in our campus) use a wired network to access these servers by default. RTT and bandwidth for intra-continental setups are approximately 30ms, 800kB/s; and for inter-continental cases, are approximately 200ms, 800kB/s respectively. In order to measure the effectiveness of the CC mechanisms in different time periods, every 6 hours, clients send requests to servers (at 09:00, 15:00 and 21:00). We first randomly pick any CC mechanisms and then select the mechanism using Antelope. For each request, the server sends back different randomly sized files (1KB-50MB) just as with the emulated testbed. Each request and the corresponding reply form a new TCP flow. Each experiment lasts for half an hour. In total, we run experimentation over one week and train over 50K TCP flows (7K each day). For the rest of this section, we use the combination of emulated and real world training data for evaluation.

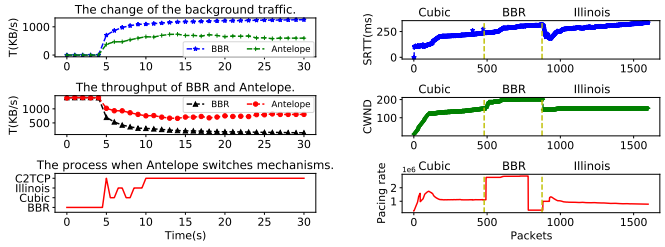
C. Performance Evaluation

We first show how Antelope switches between CC mechanisms (including BBR, Cubic, C2TCP, Vegas, Illinois and Westwood) and how the TCP parameters change. We then describe the performance of Antelope in both evaluation and production environments.

1) Validating Switching Mechanism: We first validate that when network condition changes, (*e.g.* novel congestion is encountered) Antelope can switch mechanisms in the kernel without causing issues. To test this, in the emulated network environment, we initiate a flow from the client to the server. We then, after a period of time, add background traffic between the second client and server to trigger congestion (using Cubic). We then monitor which CC algorithms are selected and validate Antelope's capacity to dynamically switch without degrading performance. As a baseline, we compare against vanilla BBR.

In Figure 6a, the top plot shows the rate of background traffic, the middle plot shows the throughput of Antelope vs. BBR, and the bottom plot shows the CC algorithms that Antelope switches between. Unsurprisingly, we see that the throughput of both Antelope and BBR decreases as the background traffic grows. However, the throughput of BBR decreases much more than Antelope. This occurs because Antelope dynamically switches between algorithms to reflect the new operating conditions. This is demonstrated in the bottom plot of Figure 6a, which depicts the CC algorithms selected by Antelope during the experiment. At the beginning, Antleope selects BBR; when the background traffic arrives, it switches between Cubic, Illinois and C2TCP. After this exploratory phase (appox. 5 seconds), Antelope switches to C2TCP stably. This occurs because C2TCP learns (correctly) that when competing with Cubic, C2TCP achieves the best performance.

We next wish to validate that Antelope can perform these switches without undermining the pre-existing TCP parameters used by the previous CC algorithm. Figure 6b presents



(a) Antelope vs. other CC mechanisms (b) Continuity of TCP parameters

Fig. 6: Validation when Antelope switches different CC mechanisms.

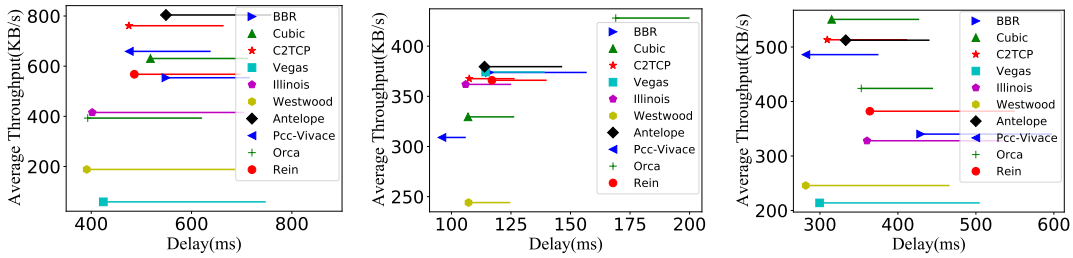
the change of srtt, CWND and pacing rate when Antelope switches between different CC mechanisms. When Antelope changes from Cubic to BBR, srtt changes smoothly, but CWND and pacing rate increase immediately; this is because BBR probes a higher bottleneck bandwidth. The delay of the flow changes slowly so the srtt changes smoothly. This validates that Antelope can maintain the continuity of TCP parameters when switching CC mechanisms.

2) Performance Evaluation in Emulated Networks: We next compare the performance of Antelope in an emulated network (using Mahimahi) against BBR, Cubic, C2TCP, Vegas, Illinois and Westwood, as well as two ML-based CC mechanisms that provide kernel implementations: PCC-Vivace and Orca. PCC-Viavce uses online learning to adjust the sending rate; for Orca, we use the trained model that is provided by Orca's authors. Finally, we also compare against another CC switching mechanism, Rein [11], which uses a rule-based algorithm to select the CC algorithm. As Rein's source code is not open, we implement Rein according to the algorithm it provides in the paper: using Cubic by default, switching to BBR in a small buffer network and switching to Westwood for WiFi connections.

The emulated network is similar to the setup described earlier. However, to differ from the training environment, we use different traces and parameters in Mahimahi (as described below). All the throughput and delay results are the averages taken from 30 runs.

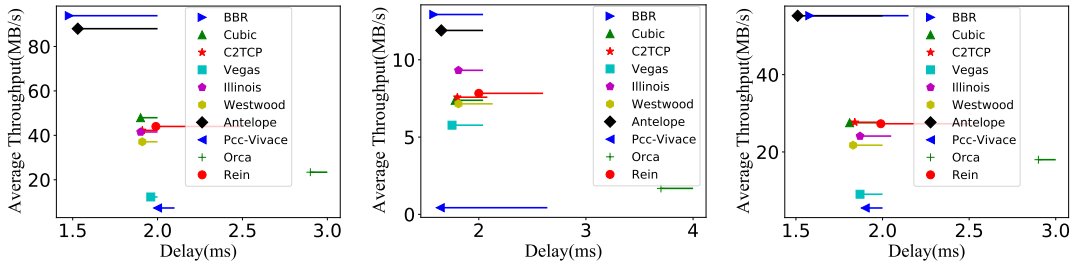
WAN. To evaluate a WAN environment, we set the link's delay and bandwidth to 100ms and 1.4MB/s in MahiMahi. The queue length is 5*BDP and the queue is tail drop first. We introduce background traffic via requests to another server, which is also connected via MahiMahi. As we introduce background traffic, the resulting packets loss rate is between 1% to 2%. We run three groups of experiments, consisting of long, short and mixed flow sizes. For long flows, the size of the requested files is randomly selected from between 3MB to 50MB. For short flows, the size is randomly selected from between 1KB to 3MB. To generate a mixture of flows, we also run experiments where we randomly select sizes between 1KB to 50MB.

Figure 7 compares the performance of different CC mech-



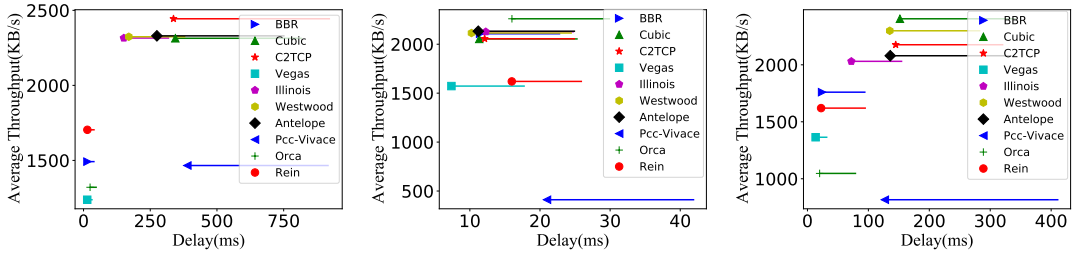
(a) wan long (b) wan short (c) wan mix

Fig. 7: Comparison of throughput and delay for different CC mechanisms in an emulated WAN environment. For delay, the icons are the average value and end of line is the 95% value.



(a) datacenter long (b) datacenter short (c) datacenter mix

Fig. 8: Comparison of throughput and delay for different CC mechanism in an emulated data center environment. For delay, the icons are the average value and end of line is the 95% value.



(a) cellular long (b) cellular short (c) cellular mix

Fig. 9: Comparison of throughput and delay for different CC mechanism in an emulated cellular network environment. For delay, the icons are the average value and end of line is the 95% value.

anisms in this environment. The x-axis is the delay and the y-axis is throughput. For delay, each icon represents the average value, and the right end of line is the 95% value. For throughput, the value of the line is the average value. The mechanism which is on the top left corner is the best. The figures show that, in a WAN environment, Antelope achieves the highest or second highest throughput compared with other CC mechanisms when requesting long, short and mixed-size files. The average delay of Antelope is in the middle compared with other CC mechanisms. We find that for most of the time, Antelope chooses Cubic or C2TCP, not BBR. Antelope achieves an average of 30% more throughput than BBR in total. Rein's performance is close to Cubic as WAN has a large buffer (Rein switches to Cubic in large buffer environments). PCC-Vivace performs poor when transferring short files, possibly because it has not converged to its optimal before the end of the transfer. Orca's performance also varies greatly. Recall that Antelope is also much more lightweight than Orca or PCC-Vivace, as it is built on the CC mechanisms available in main-stream Linux kernels.

DCN. We evaluate the DCN environment in a similar fashion to WANs using MahiMahi. We set the bandwidth as 1GB/s. The length of the router queue is 0.1*BDP with tail drop first. The packet loss rate introduced by the background traffic is about 0.1%-0.2%. The size of requested files is the same as in the WAN experiment.

Figure 8 shows the performance of different CC mechanisms in the DCN environment. The meaning of the x-

axis and y-axis are the same as Figure 7. From the figure, we see that C2TCP and Cubic have very high delay and low throughput in the DCN environment. Antelope and BBR achieve the best performance. BBR has good performance for small BDP networks [10], which is why BBR's performance in a WAN (which has a large BDP) is not as good. In the DCN environment, Antelope chooses BBR for most of the time, so its performance is close to the best. Rein's performance is very close to Cubic in DCN. This is because in a DCN, the flows are too short to perform switching according network feedback. Antelope overcomes this by using IP-level prediction, which selects based on historical observations. We also note that the performance of PCC-Vivace and Orca drops greatly in the DCN setup. We conjecture that this is because ML-based mechanisms fail to achieve their optimal as flows are completed before they get sufficient training data. Again, Antelope addresses this by using historical observations.

Cellular network. We use the traces provided by MahiMahi to emulate cellular networks. The traces are collected from T-Mobile, ATT and Verizon's LTE network in walking, driving and stationary conditions [25]. Importantly, this differs from those traces used in the training (see Section VI-B1).

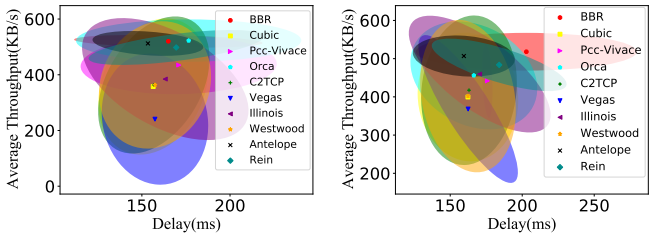
Figure 9 compares the performance of different CC mechanisms in this setup, where we can see that C2TCP achieves the highest throughput as it is specifically designed for cellular networks. Although BBR has very short delay, its throughput is low. As Antelope chooses the most suitable mechanism, its performance is one of the highest. As Rein does not have a switching rule specifically for cellular networks, its performance is not stable (sometimes close to Cubic, sometimes close to BBR). PCC-Vivace also performs poorly, possibly because of its poor adaptability in highly dynamic networks [5]; Orca again is not stable in terms of performance.

Summary. In each environment, we see that different mechanisms achieve the optimal performance. For example, C2TCP achieves high performance in WAN and cellular networks but perform poorly in DCNs; BBR's throughput is very high in DCNs, but very low in cellular networks. As Antelope selects the most suitable mechanisms, its performance is consistently one of the best in all the environments. Overall, in the 3 network environments, Antelope achieves an average of 16% improvement in throughput and 3.5% reduction in delay (for short flows) compared with BBR. Compared with Cubic, Antelope achieves an average of 19% improvement in throughput, and 10% reduction in delay. Rein, another CC switching mechanism, cannot adjust to variable network conditions and its performance is poorer than Antelope. The ML-based mechanisms (PCC-Vivace and Orca), while being more heavyweight, require a long time to converge, and thus are less stable in terms of performance.

3) Performance In-the-Wild: To evaluate Antelope's effectiveness in production networks, we use our testbed on the public cloud (see Section VI-B2). We setup servers in 5 cities located in 4 continents. Every 6 hours, we send requests from our campus to each of those servers. The clients connects to the Internet either from wired networks or via LTE. The sizes of the requested files are randomly selected between 3MB to 50MB, similar to Section VI-B. For each selection, we repeat the experiments 30 times and average the results.

Wired network. Figure 10a presents the results when clients use wired networks, where the x-axis shows the delay and the y-axis shows the throughput. The icon in the middle of each ellipse shows the mean average value of delay and throughput. The ellipses show the standard deviations from the average results. Antelope, BBR and Orca achieve the largest throughput. However, whereas the throughput's standard deviation is lower for BBR, its delay range is much higher compared to Antelope. We find, for over 85% of the time, Antelope chooses BBR in this setup. This is unsurprising as it has been proven that BBR achieves the best performance in inter-continental environments [10]. Antelope also utilizes C2TCP for 10% of the time and Cubic for 5%, resulting in the differing performance compared to BBR. As Rein's fixed threshold for distinguishing large or small buffers cannot adapt to the production network, it switches between Cubic and BBR irregularly. This means its final performance is between BBR and Cubic.

LTE network. Figure 10b reports results from the LTE network. Antelope and BBR achieve the highest throughput, but BBR has worse delay. Most of time Antelope chooses BBR. However when the the delay becomes large, other CC mechanisms (*e.g.* C2TCP) are chosen. As we observed in the emulated networks, the two ML-based mechanisms (Orca and PCC-Vivace) fail to achieve as high throughput as Antelope.



(a) wired network (b) cellular network

Fig. 10: The results of different CC mechanisms in the inter-continental production environment.

D. Overhead

The overhead for Antelope includes three parts: (1) the learning overhead in user space; (2) the information exchange via eBPF; and (3) the CC mechanism switching in the kernel.

To evaluate the overhead, we setup a testbed using 4 servers with two Intel(R) Xeon(R) Silver 4208 CPUs, 16 CPU cores, 128GB memory and 100Gb/s NIC. The servers connect to a switch with 32 100Gb/s ports. Three servers act as clients to generate TCP requests to the fourth server which acts as a TCP sender. We change the traffic volume using the clients' requests and then calculate the overhead at the sender.

TABLE III: Evaluating the overhead of Antelope.

Traffic rate		5Gb/s		15Gb/s		25Gb/s		35Gb/s		45Gb/s		55Gb/s	
Overh. Type		CPU	T(ms)	CPU	T(ms)	CPU	T(ms)	CPU	T(ms)	CPU	T(ms)	CPU	T(ms)
Mechanism Learning		2%	135	3.6%	138	4.4%	140	4.6%	139	4.6%	138	4.8%	140
Mechanism Switching		0.1%	0.001	0.1%	0.001	0.1%	0.001	0.1%	0.001	0.2%	0.001	0.2%	0.001
Exchange	To Data Module	0.16%	0.4	0.49%	0.4	0.68%	0.4	0.91%	0.4	0.95%	0.4	1.0%	0.4
	To eBPF Map	0.1%	0.1	0.1%	0.1	0.1%	0.1	0.15%	0.1	0.2%	0.1	0.2%	0.1

We calculate the overhead of each constituent of overhead by keeping other two unchanged. For example, when we calculate the overhead of switching CC algorithms, we first record the overhead of running the whole process. We then repeat the same process but *without* the switching. We define the computation overhead as the difference of CPU utilisation (%) between these two measurements. The time overhead is simply calculated by recording the time spent in each function. Table III presents the results taken as an average across 10 runs. It is worth noting the CPU overhead is the computation overhead introduced by *all* TCP traffic, while the time consumed (T) is computed on per flow basis.

We see that the overhead introduced by the mechanism switching (using eBPF in the kernel) is only 0.1%-0.2%, taking 0.001ms even when the traffic rate is 55Gb/s. The interaction between kernel and user space for TCP stats and control messages also incurs negligible overhead (the last two rows) thanks to eBPF. This confirms previous findings that eBPF is suitable for handling TCP-related operations in the kernel [9].

Unsurprisingly, the learning process incurs the largest computational overhead: about 2-4.8% of CPU usage, where the prediction time is about 140ms. Note that some flows may be shorter than the time taken for learning. This is the primary reason for why we set a default CC mechanism at the beginning of a flow and apply new CC mechanisms to flows after some time (see Section III). That said, we note that the CPU overhead is potentially a little heavy for those servers which have a large number of concurrent clients. Moving our prediction module to a central controller could reduce the overhead.

VII. Related Work

TCP varieties. We are not the first to observe that CC algorithms can be optimized for different environments. For instance, Sprout [33], C2TCP [3], ExLL [27] and PBE-CC [34] are specifically designed for cellular networks. Similarly, DCTCP [6], pFabric [7], Timely [23] and Swift [18] are designed for datacenter networks by using Explicit Congestion Notification [17]. In our work, we do not attempt to devise new CC algorithms but, rather, we exploit this observation to dynamically select the best algorithm for observed network conditions on demand.

Selection of optimal CC mechanisms. Most related to Antelope is Rein [11], which also tries to select the most suitable TCP algorithm for different networks. Rein relies on rule-based selection. It first classifies the network environment (*e.g.* WiFi or wired) and uses the CC mechanism that is manually designated to this environment. In contrast, Antelope predicts TCP mechanism more accurately via machine learning. Furthermore, Rein uses *pipe* to exchange information between user space and kernel while Antelope relies on eBPF. This makes it easier to extend Antelope with new CC algorithms and learning mechanisms. TCP-RL [26] is another work that selects suitable CC mechanisms using reinforcement learning. However, TCP-RL implements the selection entirely in user space based on Pantheon [35], which means individual applications need to implement support.

TCP implementation in the kernel. Others have focused on streamlining updated TCP implementations in kernel space. This has partly been achieved via eBPF. For example, it is possible to read TCP flow information from the kernel using BCC [1], and tcp_ebpf [9] has implemented TCP socket operations (*e.g.* setting TCP socket parameters) using eBPF. Such eBPF based operations can let users control TCP implementations from user space. CCP [24] designs an architecture which divides the control of TCP from the datapath. We do not make contributions to this space but, rather, rely on eBPF to implement Antelope in a flexible and extensible fashion.

VIII. Conclusion

In this paper we have designed, implemented and evaluated Antelope, a system for learning suitable CC algorithms on a per-flow basis. We have shown that Antelope can successfully apply the optimal or near-optimal CC algorithms across a diverse range of network types. Through this, we can improve performance without the need for administrators to manually configure their stacks. Antelope paves the way for dynamic selection of CC algorithms. In the future, we plan to deploy it in a wider array of environments. We also note that certain applications integrate their own control loops for handling congestion (*e.g.* video streaming). We are therefore curious to understand how Antelope may interoperate with such applications.

Acknowledgment

We thank our shepherd Jinsong Han and the anonymous reviewers for their insightful feedback. This work is supported in part by National Key RD Program of China No.2019YFB1802800, the National Natural Science Foundation of China(62002149, 61902171). Corresponding author: Zhenyu Li.

REFERENCES

[1] https://github.com/iovisor/bcc.

[2] Soheil Abbasloo, Tong Li, Yang Xu, and H Jonathan Chao. Cellular controlled delay tcp (c2tcp). In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 118–126. IEEE, 2018.

[3] Soheil Abbasloo, Yang Xu, and H Jonathan Chao. C2tcp: A flexible cellular tcp to meet stringent delay requirements. *IEEE Journal on Selected Areas in Communications*, 37(4):918–932, 2019.

[4] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Make tcp great (again?!) in cellular networks: A deep reinforcement learning approach. *arXiv preprint arXiv:1912.11735*, 2019.

[5] Soheil Abbasloo, ChenYu Yen, and H Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 632–647, 2020.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[8] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 329–342, 2018.

[9] Lawrence Brakmo. Tcp-bpf: Programmatically tuning tcp behavior through bpf. *NetDev 2.2*, 2017.

[10] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.

[11] Kefan Chen, Danfeng Shan, Xiaohui Luo, Tong Zhang, Yajun Yang, and Fengyuan Ren. One rein to rule them all: A framework for datacenter-to-user congestion control. In *4th Asia-Pacific Workshop on Networking*, pages 44–51, 2020.

[12] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. {PCC} vivace: Online-learning congestion control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 343–356, 2018.

[13] Alfred Giessler, J Haenle, Andreas König, and E Pade. Free buffer allocation—an investigation by simulation. *Computer Networks (1976)*, 2(3):191–208, 1978.

[14] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[15] Janey C Hoe. Improving the start-up behavior of a congestion control scheme for tcp. *ACM SIGCOMM Computer Communication Review*, 26(4):270–280, 1996.

[16] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*, pages 3050–3059, 2019.

[17] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[18] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[19] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 15–30, 2020.

[20] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. Tcp westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 287–297, 2001.

[21] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 154–171, 2020.

[22] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[23] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[24] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 30–43, 2018.

[25] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for {HTTP}. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 417–429, 2015.

[26] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[27] Shinik Park, Jinsung Lee, Junseon Kim, Jihoon Lee, Sangtae Ha, and Kyunghan Lee. Exll: an extremely low-latency congestion control for mobile cellular networks. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 307–319, 2018.

[28] Jon Postel et al. Transmission control protocol. 1981.

[29] Waleed Reda, Kirill Bogdanov, Alexandros Milolidakis, Hamid Ghasemirahni, Marco Chiesa, Gerald Q Maguire Jr, and Dejan Kostić. Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network. *ACM SIGCOMM Computer Communication Review*, 50(2):11–23, 2020.

[30] Dario Rossi, Claudio Testa, Silvio Valenti, and Luca Muscariello. Ledbat: the new bittorrent congestion control protocol. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2010.

[31] Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, and Brad Karp. {HACK}: Hierarchical acks for efficient wireless medium utilization. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 359–370, 2014.

[32] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. *ACM SIGCOMM Computer Communication Review*, 43(4):123–134, 2013.

[33] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 459–471, 2013.

[34] Yaxiong Xie, Fan Yi, and Kyle Jamieson. Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements. *arXiv preprint arXiv:2002.03475*, 2020.

[35] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 731–743, 2018.

[36] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 509–522, 2015.