A Machine Learning-Based Framework for Dynamic Selection of Congestion Control Algorithms

Jianer Zhou, Xinyi Qiu, Zhenyu Li, *Member, IEEE*, Qing Li, Gareth Tyson, Jingpu Duan, Yi Wang, *Member, IEEE*, Heng Pan, and Qinghua Wu

***Abstract*—Most congestion control algorithms (CCAs) are designed for specific network environments. As such, there is no known algorithm that achieves uniformly good performance in all scenarios for all flows. Rather than devising a one-size-fits-all algorithm (which is a likely impossible task), we propose a system to dynamically switch between the most suitable CCAs 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 the stack to use the most suitable CCA for individual flows. We build a machine learning model to learn which algorithm works best for individual conditions and implement kernel-level support for dynamically switching between CCAs. The framework also takes application requirements of performance into consideration to fine-tune the selection based on application-layer needs. Moreover, to reduce the overhead introduced by machine learning on individual front-end servers, we (optionally) implement the CCA selection process in the cloud, which allows the share of models and the selection among front-end servers. We have implemented Antelope in Linux, and evaluated it in both emulated and production networks. The results demonstrate the effectiveness of Antelope via dynamic adjusting the CCAs for individual flows. Specifically, Antelope achieves an average 16% improvement in throughput compared with BBR, and an average 19% improvement in throughput and 10% reduction in delay compared with CUBIC.**

***Index Terms*—Congestion control, eBPF, machine learning.**

I. Introduction

SINCE the birth of TCP, many congestion control algorithms (CCAs) have been proposed [1], [2], [3], [4], [5], [6]. However, none of these *individual* algorithms can achieve high network performance across *all* environments and user requirements. Two reasons account for this. First, each algorithm is designed for a particular environment. For example, Sprout [7], C2TCP [8], [9] and Verus [10] are designed for cellular networks; DCTCP [11], pFabric [12] and Swift [13] are designed for datacenter networks; TACK [14], HACK [15] and Westwood [16] are designed for wireless local area networks (WLANs). Second, network environments and application requirements have evolved over decades. For example, when CUBIC [2] (the default Linux TCP mechanism) was proposed, improving bandwidth utilization was the most important goal. However, for modern cloud gaming or live streaming applications, latency is much more critical. Our goal is therefore to devise a congestion control selection 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 CCAs have been proposed. These strive to autonomously learn the optimal CC policy for any given scenario. For example, RemyCC [17] 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 [18] uses online learning, while DeepCC [19] and Orca [20] use deep reinforcement learning (DRL) to adjust their sending rates based on network feedback. Further, AUTO [21] and MOCC [22] use multi-objective reinforcement learning to design CC mechanisms to satisfy different data transfer requirements. 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 [23]. Our goal is to *devise a CC framework (based on the CCAs 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 introduce a 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 information to user space, where we can exploit pre-existing machine learning libraries. Using supervised classification, Antelope then predicts which CCA could achieve the best performance for that particular flow. The prediction also takes application-layer needs (*e.g.* delay sensitive or throughput sensitive) into consideration. Ante-

Manuscript received 29 March 2022; revised 5 September 2022; accepted 29 October 2022; approved by IEEE/ACM Transactions on Networking Editor S. Magnusson. Date of publication 16 November 2022; date of current version 18 August 2023. This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB1802800, in part by the National Natural Science Foundation of China under Grant 62002149 and Grant 61902171, and in part by the Major Key Project of PCL under Grant PCL2021A15. *(Jianer Zhou and Xinyi Qiu are co-first authors.) (Corresponding authors: Zhenyu Li; Qing Li.)*

Jianer Zhou and Yi Wang are with the Southern University of Science and Technology, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: zhouje1005@gmail.com; wy@ieee.org).

Xinyi Qiu, Qing Li, and Jingpu Duan are with the Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: qiuxy@pcl.ac.cn; liq@pcl.ac.cn; duanjp@sustech.edu.cn).

Zhenyu Li, Heng Pan, and Qinghua Wu are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China (e-mail: zyli@ict.ac.cn; panheng@ict.ac.cn; wuqinghua@ict.ac.cn).

Gareth Tyson is with The Hong Kong University of Science and Technology (Guangzhou), Guangzhou 510000, China (e-mail: gtyson@ust.hk).

Digital Object Identifier 10.1109/TNET.2022.3220225

1558-2566 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

lope then continues to monitor the flow and changes the CCA dynamically if the network environment or flow state changes. To coordinate this, Antelope uses eBPF (a new kernel function which supports more control in the kernel from user space) [24], [25] to deliver information between the user space and kernel. We have implemented Antelope in Linux, and also propose remote cloud-based learning as an alternative implementation to reduce the overhead due to model training and prediction on individual front-end servers where the applications run. Through extensive experiments, we demonstrate that Antelope nearly always chooses the most suitable mechanism for each flow. Although most of time Antelope chooses the CCAs which are specifically designed for that network, we find cases where network fluctuations lead it to choose other (unexpected) CCAs. We show that these choices, indeed, result in better performance and confirm that Antelope selects suitable CCAs adaptively.

Our key contributions are:

- We design and implement Antelope, an adaptive CC framework which dynamically reconfigures between the most suitable CCA on a per-flow basis. Antelope only needs changes at the TCP sender 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 CCAs for flows that have similar patterns with the training data, but also on the flows that have not appeared before. We show that eBPF, as part of Antelope, is an effective choice to manage CCAs in the kernel, even after a TCP flow has been established. The selection can also be implemented in a centralized cloud server. By doing so, the models and the selection can be shared among front-end servers that serve individual flows, and thus the extra overhead introduced by the machine learning training and processing is amortised.
- Extensive experiments in a wide area network (WAN), data center network (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). The experiments also proves the benefits of the shared remote cloud-based learning in reducing overhead on front-end servers.

While the basic idea of Antelope has been introduced in [26], this extended version adds two new enhancements to Antelope: (*i*) the incorporation of applications' performance preferences and the remote cloud-based learning; and (*ii*) several sets of new experiments.

The rest of the paper is structured as follows. Section II explains our motivation and challenges in detail. Section III offers an overview of the system. Section IV presents details of the system and describes the classification algorithm. Section V outlines the implementation of the system. We explain the training and our extensive experiments in Section VI. Related work is covered in Section VII and Section VIII concludes the paper.

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

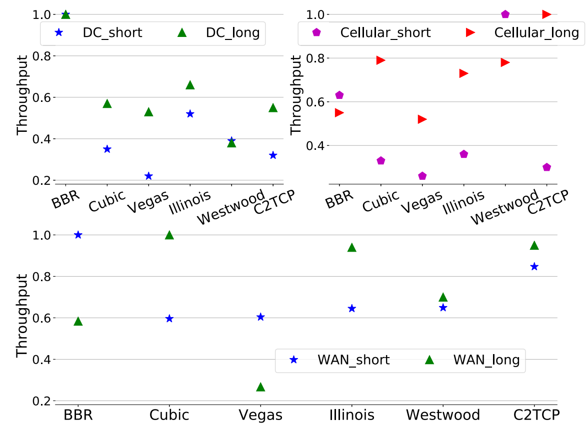


Fig. 1. Performance of different CCAs in three different networks.

II. Motivation

A. Why Switch CCAs?

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 CCA 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 CCAs over datacenter, cellular and wide area (wired) networks. This is done using the Mahimahi emulator [27], parameterized as follows. The cellular network is configured using the public trace data from [20]; 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 [3]. 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 [28]. 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 [3]. 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.

Machine learning CC mechanisms are limited. Rather than adjusting the congestion window or pacing rate using ML, we build a model to select the CCAs on a per-flow basis.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

We do this for two reasons. First, as pointed out by both Orca [20] and Rein [29], learning-based approaches (*e.g.* Indigo [30], Aurora [23]) suffer from performance degradation and slow convergence when used in unseen conditions. In contrast, hand-written classic CCAs do not have these two issues. Second, classic CCAs that have been widely used in practice, often achieve very good performance in the network environments for which they are designed (*e.g.* Westwood [16] for wireless networks).

B. Challenges

Rather than devising a one-size-fits-all CCA, we design and implement **Antelope**, a framework that can dynamically switch between the most suitable CCAs for specific flows in specific environments. This raises four unique challenges.

Selection of CCA. Antelope must design an appropriate reward function to select the optimal CCA for a given scenario and for a given application. 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 CCA is therefore not wise. Worse still, applications may have different performance needs. For instance, online chat and cloud gaming are more delay sensitive, while traditional VoD services are more throughput sensitive. Furthermore, manually selecting CCAs, even with machine learning support, is difficult for network operators and domain specialists [31]. 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 CCA would have been most suitable. Antelope must be able to rapidly select the most suitable CCA.

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.

Computation overhead. The computational resources are limited for front-end servers which run TCP flows for applications. Machine learning usually introduces heavy computation overhead. It thus would be better to mitigate the overhead introduced by the model training and model inference for the CCA selection in such application servers.

III. Antelope Overview

Overview. The duration of a TCP flow can be divided into three phases: connection setup, data transmission and connection 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 CCA according to the flow's characteristic. The most suitable CCA will then be passed to the Mechanism Switch component (in the kernel) which will switch to that CCA 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.

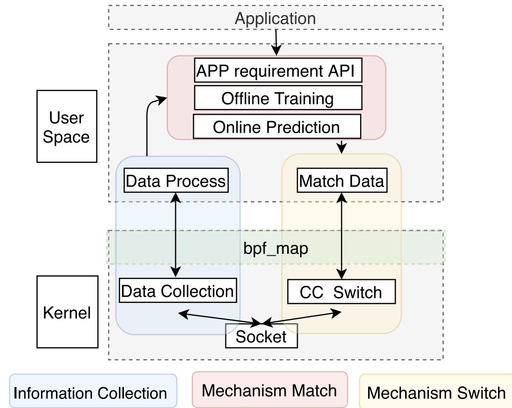


Fig. 2. High-level components of Antelope.

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 CCA 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 model. To inform this process, the Offline Training module will train the matching models using a reward function. Specifically, we build decision-tree models using XGBoost. The match component also considers the performance preferences of different applications. Some applications (*e.g.* online chat) are delay sensitive, while others (*e.g.* file transferring) are throughput sensitive. The Mechanism Match component exposes an API to the application to get the applications' performance preferences. We then predict the suitable CCAs by considering the flow and network states as well as the application requirements in the Online Predication module. Specifically, the reward function reflects the application's requirements by assigning different weights to the throughput and delay. The details of this component are shown in Section IV. Note that this component can also be implemented in the cloud, where multiple front-end servers can share one component to reduce the computation overhead on individual servers.

Mechanism Switch. The Mechanism Match component records the flow identifier (by IP and port) and the corresponding CCA for the flow under consideration. Using eBPF, these information is delivered to the kernel. Then the Mechanism Switch component (in the kernel) will switch to the selected CCA. 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 CCAs if instructed. In tcp_close, the hook function sends flow closing signals to the Mechanism Match and Mechanism Switch components.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

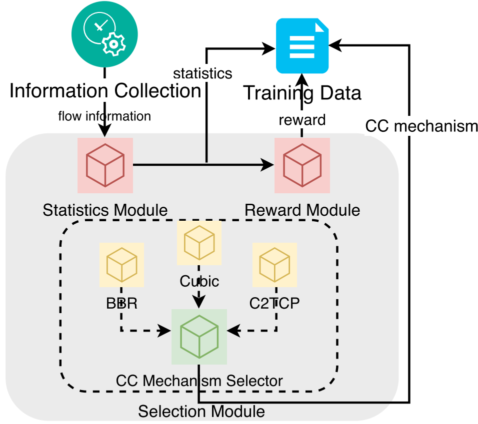


Fig. 3. Online Prediction module overview.

IV. Prediction and Training

A. Prediction Module

Overview. The Online Prediction module is the heart of matching process. Its goal is to predict the optimal CCA based on the TCP flow information. Figure 3 shows the overview of the Online Prediction module. It consists of three main modules: the Statistics Module, Reward Module and Selection Module.

As an input, Antelope takes 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 CCA 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 CCAs. By comparing the reward predictions made by each model, the best CCA is selected. When the next data unit is generated, the Reward Module analyzes its statistics to evaluate the effect of the last switching CCA.

The models are trained using a collection of tuples (s_i, r_i) for candidate CCAs. These are used for online training, where s_i is the statistical information and r_i is the reward value for the i-th CCA. The output of the Statistics Module and the Reward Module are used go generate the tuples for online training. The output of the CC Selection Module is also directed towards Training Data to inform the training process which CCA is selected.

Statistics Module. The Statistics Module is responsible for gathering flow information. It does this by reading flows' information from the Information Collection component. On receiving an ACK, the kernel updates the flow's information, *e.g.* RTT, $CWND$, sending rate, the number of lost packets. 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, *i.e.* $N = 20$), we calculate the statistics based on the flow information collected for each ACK packet. We set the data unit size as a tradeoff between computational overhead and effectiveness. Note, calculating statistics on a per data unit basis (as opposed to statistics per ACK) reduces the influence of the network noise in machine learning based decision making [32]. A summary of the statistics are shown in Table I.

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

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.

CCA 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 CCA, 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 CCA for the next period.

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

$$\begin{aligned}\hat{R} &= R/R_{max} \\ &= \left(\frac{throughput' - \eta * loss}{max\{\theta * delay', 1\}}\right) / \left(\frac{pacing_rate_max}{delay_{min}}\right) \end{aligned} \tag{1}$$

Giessler [33] showed that the effectiveness of a CCA 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 (similar to Orca [20]). 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):

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

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

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

sensitive when more packets are sent in the flow.

$$throughput' = throughput * \zeta + MAX_RATE * (1 - \zeta) \tag{3}$$

Finally, $throughput'$ is defined as in Eq. 3, where $throughput$ is the measured average throughput, MAX_RATE is the largest value for $pacing_rate_max$ (defined in kernel as $2^{23} - 1$), the parameters ζ and θ are used to capture the application preferences on throughput and delay ($\zeta \in \{0, 1\}$ and $\theta \in \{0, 1\}$). Specifically, we define three modes of applications' performance preferences: *delay sensitive mode*, *throughput sensitive mode*, and *default mode*. In the default mode, both ζ and θ are set to 1, which means that the CCA with both good throughput and low delay should have a higher reward value (and therefore should be chosen). For the delay sensitive mode, ζ and θ are set to 0 and 1 respectively; then the CCAs with lowest delay will be chosen. On the other hand, for the throughput sensitive mode, ζ and θ are set to 1 and 0 respectively; then Antelope will choose the CCAs with the best throughput. Via these two parameters, Antelope considers the application's preferences (throughput or delay sensitive) when choosing the most suitable CCAs.

Selection Module. This module is responsible for retrieving the reward predictions across the set of available CCAs (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 CCA 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 or prefix (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 CCAs. At each step t, the Selection Module observes the statistics (s_t), and then selects the CCA 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,

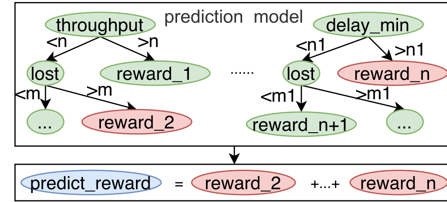


Fig. 4. Architecture of the decision tree for prediction.

when, for example, we want to predict BBR's reward for one stream, we input the flow information to the BBR prediction model. For each 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 CCA is deployed and prior using offline training.

The computational complexity of XGBoost is $O(Kd||x||_o + ||x||_o \log n)$, where d is the maximum depth of the tree, K is total number of trees, and n is training data size. Note, the values of d and K can be set in the training process. $||x||_o$ is the number of non-missing entries in the training data [34]. As shown in Table I, we have 7 features for each training data point. Suppose we have n training data, $||x||_o$ is capped at $7 * n$. So the computational complexity is $O(K*d*7*n+7*n*\log n)$. We set the value for d and K as 6 and 40 respectively as experimental results show that these values can avoid overfitting. We can see that the computational complexity grows with number of training data (n) in multiples of $\log n$.

IP-level prediction. In IP-level prediction, Antelope selects the CCA based on the historical results of the streams belonging to the same IP or a/24 segment. This allows Antelope to select an appropriate CCA before a flow has been initiated. Algorithm 2 presents the IP-level prediction pseudocode. For each IP range, Antelope records the number of times that each CCA 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. Note, the historical data is weighted by an coefficient α ($0 < \alpha < 1$) which is inversely proportional to the age of the data (the older the data is, the lower the weight is). Finally, the CCA that has been chosen most frequently with the highest reward is selected.

Algorithm 2 IP-Level Prediction Algorithm

```
1:  function IP PREDICT(ip, cc_mechanism)
2:      cc_count_map ← ip_CCs_map[ip]
3:      //Reduce the weight of all historical data.
4:      for cc in cc_count_map.keys do
5:          cc_count_map[cc] ← cc_count_map[cc] * α
6:      end for
7:      cc_count_map[cc_mechanism] += 1
8:      //Choose the cc mechanism with the largest count.
9:      cc_predict ← getMaxCountCC(cc_count_map)
10:     //Update the IP and cc mechanism in bpf_map.
11:     updateBpfMap(ip, cc_predict))
12: end function
```

B. Training Module

The above relies on a trained model that can predict the reward for a given flow using each CCA available. For training

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

the XGBoost model, we perform both offline and online training. XGBoost is a supervised learning algorithm. The training inputs are data pairs, such as $(\hat{x}_0, y_0), (\hat{x}_1, y_1) \ldots (\hat{x}_n, y_n)$, where $\hat{x}$ is the features vector and y is the label. Using this past input, XGBoost tries to predict the correct label for unseen inputs.

XGBoost integrates weak tree models to achieve strong tree models by iterative training. The computation of each decision tree model is independent. Such parallel computation makes XGBoost's learning process fast [34]. The TCP flows may be quick to finish and Antelope needs to predict CC mechanisms timely; this is why we choose XGBoost for selection. Algorithm 3 presents the pseudocode for the Training Module. Note that both offline and online training follow the same process.

Algorithm 3 Training Algorithm

```
1: function CREATETRAINDATA(statistics_t, cc, train_data)
2:     //train_data[cc] is the training data for a specific CC.
3:     cc_train_data ← train_data[cc]
4:     reward_t ← cal_reward(statistics_t)
5:
6:     pre_data.append(reward_t)
7:     cc_count ← cc_train_data.size()
8:     cc_train_data[cc_count] ← pre_data
9:     pre_data ← statistics_t
10:    //Write training data into files.
11:    if cc_count > MAX_COUNT then
12:        write_train_data(cc_train_data, cc_files[cc])
13:        delete(cc_train_data)
14:    end if
15: end function
16: function TRAINMODEL(cc_files)
17:    for cc_file in cc_files do
18:        cc_model ← XGBOOST(cc_file)
19:        //Model persistence.
20:        model_dump(cc_model)
21:    end for
22: end function
```

Offline training. We initiate training in an offline fashion, where we trigger clients to connect to the server, which then randomly selects different CCAs 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 CCA. 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 CCA 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.

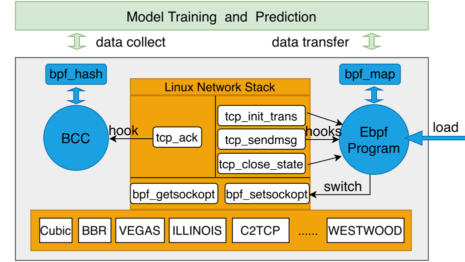


Fig. 5. Overview of the Antelope implementation.

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 CCA. The suitable CCA for this flow is then delivered back to the kernel using bpf_map. Antelope then switches the CCA in the kernel. An overview of the implementation is shown in Figure 5.

A. Collecting Flow Information

We use the BPF Compiler Collection (BCC) probe function to get the TCP flow information [35]. 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 pass flow information from the kernel to user space, we use the ebpf_hash. To pass the suitable CCA from user space to the CC Switch module in the kernel, we use the bpf_map. The suitable congestion mechanism set via ebpf_map is formatted as a key-value pair: IP+port → CCA. As at the beginning of a flow, there is not enough information to predict the best algorithm, we select the default algorithm or the one based on the historical information associated with that IP.

On receiving an ACK packet, the eBPF's user space program will get flow information from kernel via ebpf_hash. Then the information will be stored in a hashmap (using IP+port as the key). Periodically (default 20s), the data in the hashmap will be dumped to disk first and then emptied. We evaluate the memory usage due to the using of eBPF in Section VI-I.

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 algorithm [25]. We set three hook points in the kernel to trigger the switching process: tcp_init_transfer, tcp_sendmsg, tcp_close_state. For the

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

tcp_init_transfer hook, the eBPF program will set the new algorithm based on 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 algorithm 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 (*i.e.* a data unit) are received, the prediction process is triggered (by default, $N = 20$). If the prediction process finds another suitable algorithm for this flow, it updates the ebpf_map, adding the IP+flow ID $\rightarrow$ congestion algorithm item in the map. If the new algorithm 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 algorithm in the map, the eBFP program will set this flow's congestion control algorithm to the new one. To avoid switching the algorithm too frequently, we only switch upon seeing M (default 2) consecutive recommended changes.

Antelope can switch between CCAs that are implemented in mainstream Linux kernel, currently including BBR, CUBIC, C2TCP, Vegas, Illinois and Westwood. Antelope chooses these algorithms as they are wildly used and implemented in the release version of kernel. It is worth noting that Antelope can be applied for the dynamic selection of other CCAs implemented in kernel. Regardless of whether competing TCP flows use Antelope, individual flows may use different CCAs. Thus, Antelope inherits the TCP-friendliness of the chosen CCAs. For example, if Antelope chooses BBR, then it will take a larger share of the bottleneck bandwidth than CUBIC in shallow-buffered network.

D. Parameters Continuity

When dynamically switching between CCAs, it is important to ensure continuity in the flow parameters. For example, the new CCA should be initiated with the CWND of the previous CCA. Two kinds of parameters are related to this continuity: (*i*) Common parameters such as sending rate (CWND or pacing rate); and (*ii*) Measurement parameters such as RTT and packet loss rate. In the Linux kernel, the above parameters are recorded in the struct sock, which is maintained by all of the CCAs. For the sending rate, when switching to a new CCA, we use the same value as before. For measurement parameters, we also inherit the same values. This is because different CCAs use the same module to calculate these parameters and it will therefore not affect the measurement parameters' accuracy. We will validate the continuity of parameters in Section VI-C.

Except for these common parameters, different CCAs may have their own specific parameters. These parameters are completely different. For example, BBR has $pacing_gain$, $cwnd_gain$ and $full_bw_cnt$; CUBIC has $round_start$, $epoch_start$ and $sample_cnt$; Westwood has bw_ns_est; Vegas has do_vegas_now. C2TCP is based on CUBIC, so its specific parameters are the same with CUBIC. This means that a newly initiated CCA may also have to bootstrap new parameters. To address this, as all of these parameters have default values, we simply use their default values when switching to a different CCA. Note, this is similar to what Rein [29] does.

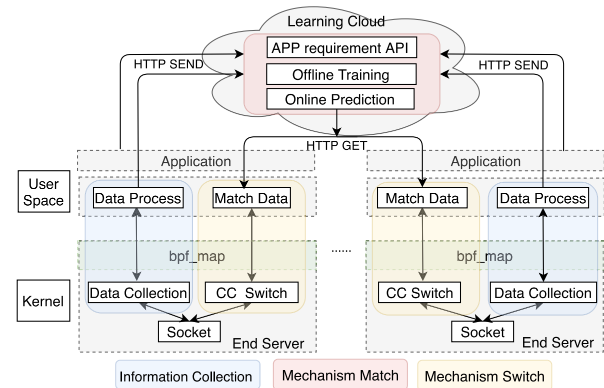


Fig. 6. The architecture of the cloud-based learning implementation.

E. Application Requirements

Antelope uses an API to receive application requirements, *i.e.* the App. requirement API in Figure 2. The API has two variables: key and $type$. We set application's port number as key to distinguish which application it is. $type$ is set as 1, 2 or 3 which means default (consider both throughput and delay), $delay$ or $throughput$ sensitive mode respectively. The API can be used after developers create TCP socket. This information is passed to Antelope's Mechanism Match component and then used to set the weight of the throughput and delay in Eq. 1 for different flows. Through this, Antelope will show preference towards either $delay$ or $throughput$ when selecting the corresponding CCAs. Note that for each CCA, we train 3 models that correspond to the 3 application preference modes. The $type$ parameter indicates which model should be used for CCA selection. If the application does not set the API, by default, Antelope will consider both the delay and throughput and choose the CCA that is balance between delay and throughput.

F. Remote Cloud-Based Learning

To reduce the computation overhead introduced by the Online Prediction and Offline Training modules on individual end servers, our framework can host certain components in the cloud, as an alternative option to perform training locally. Figure 6 presents the architecture of the cloud-based implementation. This implementation divides Antelope into two parts: the end server and the learning cloud. The Information Collection and Mechanism Switch components are both operated on the end server, whereas the Mechanism Match component is placed in the learning cloud. TCP information is passed to the learning cloud using HTTPS from the end server. The most suitable CCA is then computed using the Online Prediction and Offline Training modules and sent back from the learning cloud to the end server via HTTPS. To reduce the connection overhead introduced by HTTPS, we use long-live HTTP sessions between the learning cloud and the end server.

Note, this implementation has two main benefits. First, it reduces the computational overhead introduced by the Online Prediction and Offline Training modules in end server. Second, by centralizing the process, more TCP information can be accumulated to improve the effectiveness of the training results. However, the main downside is that this creates a communication overhead between the learning cloud and the end server. In Section VI-I we will evaluate both of the benefit and cost of the learning cloud.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

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 [27].

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 in 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.

Emulated Training. We test more than 30 network environments using Mahimahi (their characteristics are shown in Table II).[2] 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 [12]. In the cellular network we do not set its BDP as it is emulated by the traces [27].

We generate request flows of different sizes (flow size between 1KB and 50MB). The training procedures for each parameter combination are repeated 3 times. We run all CCAs in each setup and collect the training data for each CCA. Specifically, for every network environment, we set the sender to a fixed CCA then randomly switch to other algorithms 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).

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.

[2]All of our environment's setting values are those that have been tested by *iperf* or *ping*.

TABLE II
Range of Evaluated Env. During the Training

BW (Mb/s)	RTT(ms)	BDP	Back. traffic
12-8000	1-100	0.1-6	1KB-50MB

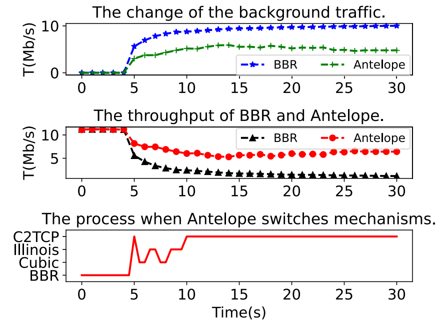


Fig. 7. Performance of Antelope vs. other CCAs.

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, 2.4Mb/s; and for inter-continental cases, are approximately 200ms, 2.4Mb/s respectively. In order to measure the effectiveness of the CCAs 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 CCAs and then select the algorithm 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 CCAs (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 CCAs 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 monitor which CCAms are selected and validate Antelope's capacity to dynamically switch without degrading performance. As a baseline, we compare against vanilla BBR.

In Figure 7, 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 CCAs 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

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

TABLE III

SUMMARY OF WHICH CCAS ANTELOPE SELECTS ACROSS DIFFERENT ENVIRONMENTS AND FLOW TYPES. THE PERCENTAGE INDICATES THE PERCENTAGE OF THE STREAM THAT EACH CCA IS USED

Type		long flow	short flow	mix flow
Emulation	**WAN**	CUBIC (59%) BBR (27%) C2TCP (10%)	BBR (80%) vegas (17%)	CUBIC (68%) C2TCP (23%) BBR (5%)
	DCN	BBR (98%)	BBR (95%)	BBR (97%)
	Cellular	C2TCP (63%) Westwood (21%)	C2TCP(52%) Westwood(23%) CUBIC (15%)	C2TCP (68%) Westwood (18%) Illinois(7%)
Wild	**Wired**	BBR (71%) CUBIC (17%) C2TCP (10%)		
	Celluar	C2TCP (51%) BBR (38%) Westwood (7%)		

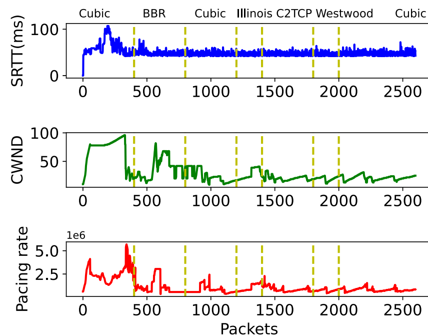


Fig. 8. The continuity of parameter values in the sock struct structure when Antelope switches between different CCAs.

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 7, which depicts the CCAs 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 CCA. Figure 8 presents the change of srtt, CWND and pacing rate when Antelope switches between different CCAs in one experiment. We can see that the 3 parameters change smoothly when Antelope switches between CCAs.

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 CCAs 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 [29], which uses a rule-based algorithm to select the CCA. 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.

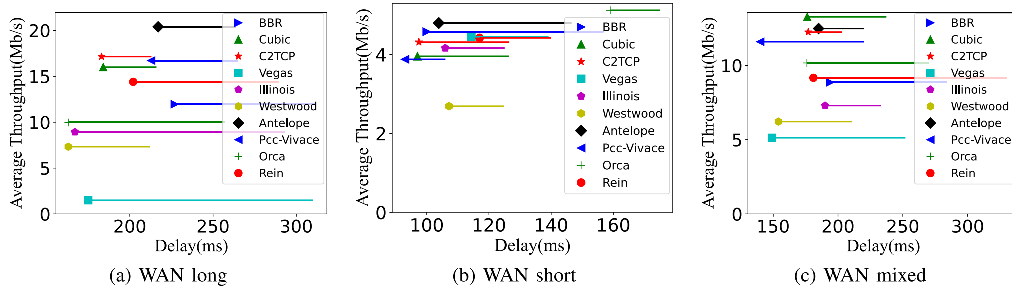
WAN. To evaluate a WAN environment, we set the link's delay and bandwidth to 100ms and 120Mb/s in MahiMahi. The queue length is 5*BDP and the queue is tail drop first. We introduce background traffic via requests to another server, which is also connected via MahiMahi. As we introduce background traffic, 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 and 50MB. For short flows, the size is randomly selected from between 1KB and 3MB. To generate a mixture of flows, we also run experiments where we randomly select sizes between 1KB and 50MB.

Figure 9 compares the performance of different CCAs in this environment. The x-axis is the delay and the y-axis is throughput. For delay, the marker is the average value and the end of the line is the 95-th percentile 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 CCAs when requesting long, short and mixed-size files. The average delay of Antelope is in the middle compared with other CCAs. We find that for most of the time, Antelope chooses CUBIC or C2TCP, not BBR (see Table III for more details). 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 CCAs 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 8Gb/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%. We set the flow size following DCTCP [11], which shows that while the flow size ranges from 2KB to 50MB, most of the flows are small (as web search dominates). As such, we set the background traffic size in DCN as follows: 50% flows are less than 100KB, 40% flows are between 100KB to 1MB and 10% flows are between 1MB to 50MB. The size of requested files is the same as in the WAN experiment.

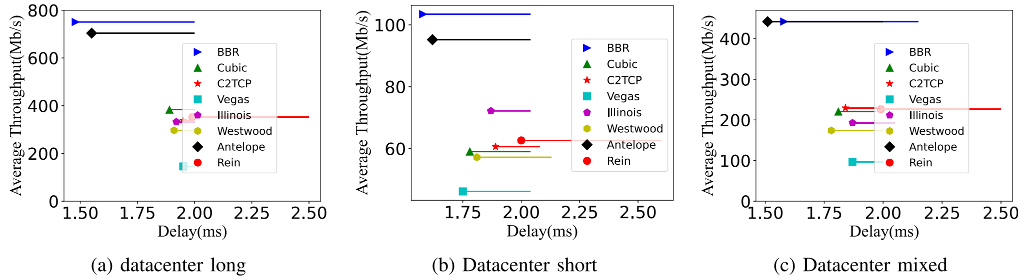
Figure 10 shows the performance of different CCAs in the DCN environment. The meaning of the x-axis and y-axis are the same as Figure 9. From the figure, we see that C2TCP and CUBIC have very high delay and low throughput in the DCN

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.



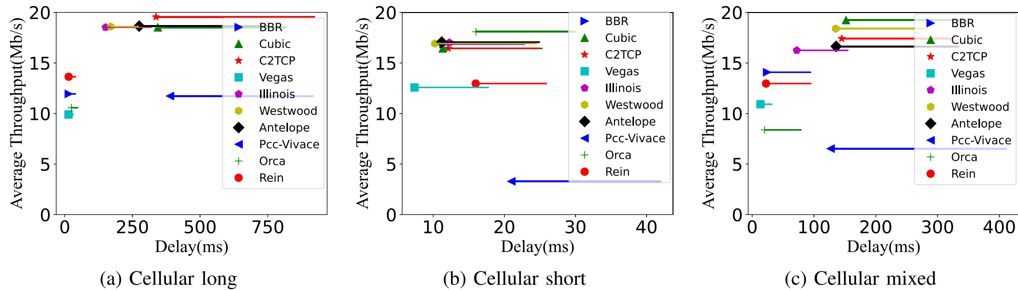
(a) WAN long (b) WAN short (c) WAN mixed

Fig. 9. Comparison of throughput and delay for different CCAs in an emulated WAN environment. For delay, the marker is the average value and the end of the line is the 95-th percentile value.



(a) datacenter long (b) Datacenter short (c) Datacenter mixed

Fig. 10. Comparison of throughput and delay for different CCAs in an emulated data center environment. For delay, the marker is the average value and the end of the line is the 95-th percentile value.



(a) Cellular long (b) Cellular short (c) Cellular mixed

Fig. 11. Comparison of throughput and delay for different CCAs in an emulated cellular network environment. For delay, the marker is the average value and the end of the line is the 95-th percentile value.

environment. Antelope and BBR achieve the best performance. BBR has good performance for small BDP networks [3], 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 (see Table III for more details), 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. Antelope overcomes this by using IP-level prediction, which performs the selection based on historical observations. Note that, as Orca and PCC-Vivace are not targeted for DCNs, we do not compare them for fairness.

Cellular network. We use the traces provided by MahiMahi to emulate cellular networks. The traces are collected from T-Mobile, AT&T and Verizon's LTE network in walking, driving and stationary conditions [27]. Importantly, this differs from those traces used in the training (see Section VI-B).

Figure 11 compares the performance of different CCAs 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 CCA (see Table III for more details), 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 [20]; Orca again is not stable in terms of performance.

Summary. In each environment, we see that different algorithms 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 algorithm, its performance is consistently one of the best in all the environments. This is particularly helpful for servers, which need to handle flows from WANs, DCNs and cellular networks at the same time (which is common for servers in the cloud [36]). However sometimes Antelope may make wrong actions brought by the limitations of machine learning algorithm. That is why when Antelope is first applied in a new network environment, it needs online training to improve its learning effectiveness. Overall, in the three 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 19% improvement in throughput, and a 10% reduction in delay. Rein, another CC

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

switching mechanism, does not adjust to variable network conditions and performs poorer than Antelope. The ML-based mechanisms (PCC-Vivace and Orca), while being more heavy-weight, require a longer time to converge, and thus are less stable in terms of performance.

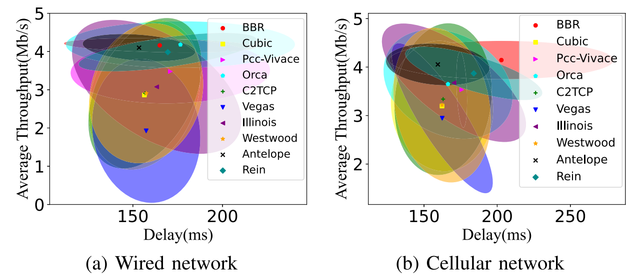
3) Performance In-the-Wild: To evaluate Antelope's effectiveness in production networks, we use our testbed on the public cloud (see Section VI-B). We setup servers in 5 cities located in 4 continents (Beijing, New York, London, Sydney and Dubai). Every 6 hours, we send requests from our campus (Shenzhen in China) to each of those servers. The clients connect to the Internet either from wired networks or via LTE. For the wired networks, the RTT between Shenzhen and Beijing, New York, London, Sydney and Dubai ranges from 70ms to 180ms. For LTE networks, the RTT between them are close to the wired networks but fluctuate more (affected by the cellular network). All of the servers' bandwidth is 4Mb/s (depending on the server's package configuration we purchased in the public cloud). The sizes of the requested files are randomly selected between 3MB to 50MB, as discussed in Section VI-B. For each configuration, we repeat the experiment 30 times and average the results. Specifically, the sizes of the requested files are 3MB, 5MB, 10MB, 15MB, 20MB, 25MB, 30MB, 40MB, 50MB. Each request uniformly chooses a file at random for transferring. For each CCA, we send 30 requests (one by one) to one server in each run. As we have 5 servers, we send 150 requests for each CCA in each run. As we take 3 runs per day and our experiment lasts one week, totally we send 3,150 request for each CCA.

Wired network. Figure 12a presents the results when clients use wired networks, where the x-axis shows the delay and the y-axis shows the throughput. The marker 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 highest 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 (see Table III for more details). This is unsurprising as it has been proven that BBR achieves the best performance in inter-continental environments [3]. 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 12b reports results in the LTE network. Antelope and BBR achieve the highest throughput, but it has worse delay. Most of time Antelope chooses BBR (see Table III for more details). However when the delay becomes large, other CCAs (*e.g.* C2TCP) are chosen. As we observed in the emulated networks, the two ML-based approaches (Orca and PCC-Vivace) fail to achieve as high throughput as Antelope.

D. Dissecting CC Selection

In this subsection, we analyze which CCAs Antelope selects for each of the environments under study. Table III shows the CCAs that Antelope selects in each environment for different kind of flows. The percentage reflects how long the specific



(a) Wired network (b) Cellular network

Fig. 12. The results of different CCAs in the inter-continental production environment.

CCA is used across the whole life of the flow and it is the average across all flows.

In the emulation environment, shown in Table III, we divide the flows into long, short and mixed type. We see that, for the WAN, Antelope switches between CUBIC and BBR. For long flows in WAN, CUBIC is chosen while for long flows Antelope chooses BBR most of time. This is probably because WANs are large buffer networks and for long flows, CUBIC can occupy the bandwidth and buffer. But for short flows, they usually finish before they can occupy the buffer.

In the emulated DCN, flows largely use BBR. This is likely because in a DCN, the switch's buffer is limited and the small buffer limits the performance of CUBIC (as it is a packet loss based CCA). For the emulated cellular network, most of the time Antelope chooses C2TCP. This is because the cellular network's situation changes rapidly. Among all of these CCAs, C2TCP is most suitable for such fluctuating networks. Sometimes Antelope also chooses Westwood (18-23%). This is because Westwood is designed for wireless environment (such as WiFi), which have similar characteristics to cellular networks.

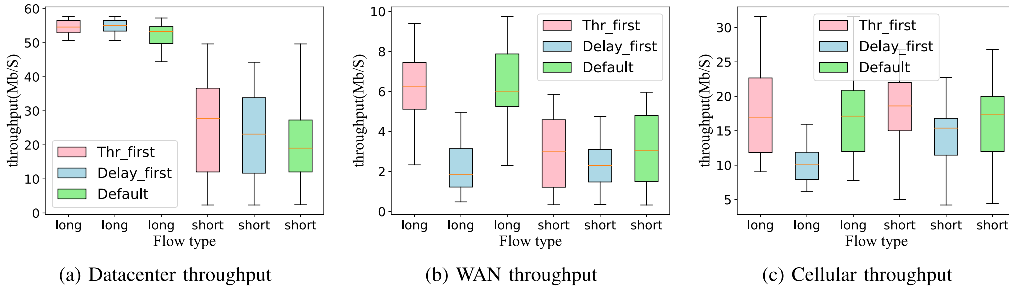
We also inspect the CCAs used in our in-the-wild experiments. We find that Antelope chooses BBR most of time for the wired setup. This is the same result as the long flows in the emulated WAN. For the real cellular network, Antelope chooses C2TCP. Again, this confirms the correctness of our emulated cellular network, which also selects C2TCP.

In summary, we find that Antelope chooses CCAs that are specifically designed for that environment. However, sometimes it selects other (unexpected) CCAs. We conjecture that network fluctuations account for it. Specifically, as Antelope observes network behavior fluctuations, the corresponding CCAs do not fit the network environment at that time. Thus, Antelope changes the CCAs according to the online feedback of the flows and environment.

E. Addressing Application Requirements

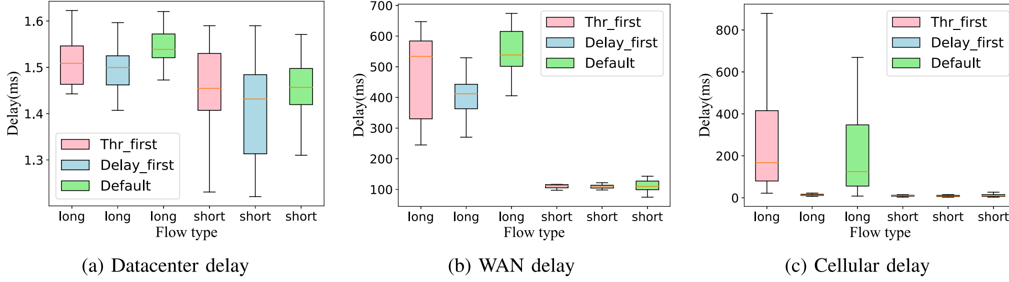
We next inspect how effectively Antelope can adapt to different applications requirements (*i.e.* related to delay *vs.* throughput). As we explain in Section IV, ζ and θ can be set as 0 and 1 respectively for delay sensitive requirement, while for throughput sensitive requirement they can be set as 1 and 0 respectively. Figure 13 shows the throughput when Antelope is at (*i*) default; (*ii*) delay first; or (*iii*) throughput first mode. Similarly, Figure 14 shows the delay. We present results across different emulated environments. The default mode means that Antelope considers both delay and throughput, whereas the other two show preferences towards delay or throughput. Across all networks, from Figure 13 we see that Antelope in the throughput sensitive mode has a higher throughput than

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.



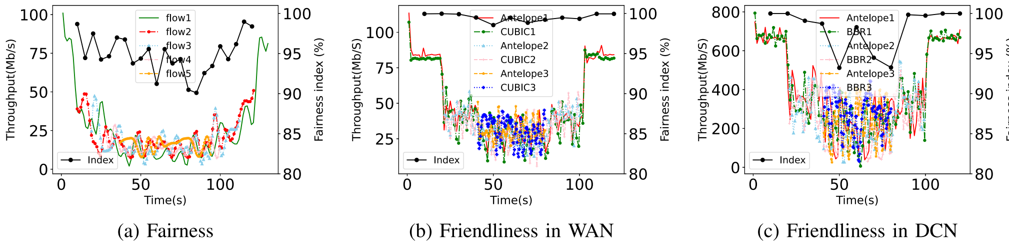
(a) Datacenter throughput (b) WAN throughput (c) Cellular throughput

Fig. 13. Comparison of the throughput when Antelope's requirements are set to default, delay sensitive and throughput sensitive.



(a) Datacenter delay (b) WAN delay (c) Cellular delay

Fig. 14. Comparison of the delay when Antelope's requirements are set to default, delay sensitive or throughput sensitive.



(a) Fairness (b) Friendliness in WAN (c) Friendliness in DCN

Fig. 15. The fairness and friendliness of Antelope in different networks.

the default and delay sensitive mode (both for long and short flows). In contrast, Figure 14 shows that when Antelope is in delay first mode, its delay is smaller (for both long and short flows), compared to the default and throughput first mode. The results confirm that Antelope can adjust its performance according to the application's delay or throughput preferences. Besides, the DCN throughput and delay are much more consistent among different modes compared to cellular/WANs, because in DCN the CC algorithm with best throughput and lowest delay are the same as Figure 10 shows.

F. Fairness and Friendliness

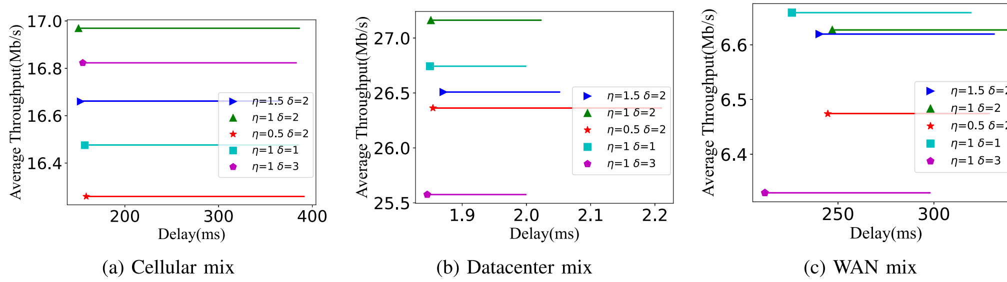
In this subsection, we discuss the fairness and friendliness for Antelope. We evaluate the fairness between flows by Jain's fairness index [37], [38], which is calculated as: $((\sum_{i=1}^{n} thr_i)^2)/(n * (\sum_{i=1}^{n}(thr_i)^2))$, where thr_i is the throughput for the $i - th$ flow. The fairness index ranges from 0 to 1 and a value close to 1 indicates that the flows fairly share the bandwidth.

First, we evaluate fairness when all flows are Antelope. Figure 15a shows how Antelope flows compete with other Antelope flows, where the left y-axis is the throughput while the right y-axis is the Jain's index. In the WAN environment (see Section VI-C), we setup five long flows. Every 5 seconds, a new flow is added and at 25s all of the five flows compete for the bottleneck. Starting at 100s, every 5s one of the flows finishes until 125 all flows are done. We can see that the bandwidth can be shared evenly across the flows, which is also evidenced by the average Jain's index around 0.95. This is because all of the Antelope flows pick the most effective CCAs to compete with each other. We find that all of the flows switch between CUBIC and BBR (the same selection in WAN for long flows, shown in Table III).

For friendliness, we first compare Antelope with CUBIC in a WAN environment. This is because Antelope usually selects CUBIC in the WAN environment (see Table III). We setup one Antelope and one CUBIC flow at 0s; we then setup a second Antelope and second CUBIC flows at 20s, and a third Antelope and CUBIC flow at 40s. Starting at 80s, every 20s, one Antelope and one CUBIC flows completes. The throughput of the six competing flows is shown in Figure 15b. We see that Antelope fairly shares the bandwidth with CUBIC when new flows enter and leave the network. The Jain's fairness index is close to 1, implying Antelope flows shares the bandwidth with CUBIC flows friendly. We further compare Antelope with BBR in a DCN environment. The results are presented in Figure 15c. Again, we see that Antelope shares the bandwidth fairly.

Discussion. Antelope chooses the CC algorithm that has the highest reward value, and that may help to maintain the friendliness in some situation. For instance, in large buffer networks, flows with CUBIC will occupy all of the bandwidth, while flows with BBR will have low performance in such situations. However, Antelope can choose CUBIC for such flows and compete with CUBIC to achieve better performance and friendliness.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.



(a) Cellular mix (b) Datacenter mix (c) WAN mix

Fig. 16. Comparison of the throughput and delay when we vary η and δ across different environments.

TABLE IV

SUMMARY OF RESOURCE CONSUMPTION FOR ANTELOPE. WE PRESENT STATISTICS ON CPU LOAD AND EXECUTION TIME, ACROSS DIFFERENT TRAFFIC LOADS

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

G. Validating Parameter Choice

Recall, η defines the influence of packet loss and δ determines the influence of delay for reward function in selecting CCAs. We next inspect the best settings for η (default 1) and δ (default 2). Figure 16 presents the performance (throughput, delay) across the emulated DCN, WAN and cellular networks. For each environment, we experiment with different values of η and δ. From the figure we can see that, when we set η to 1, the flow's performance is superior compared to the values of 0.5 and 1.5. If we set η to 0.5, the performance in the cellular network is worst. This is because the cellular network has high packet loss, and the value of 0.5 makes the reward function less sensitive to loss.

When δ is set to 1, Antelope achieves small delay but also lower throughput. On the other hand, when δ is set to 3, the flows' throughput varies substantially. For example, in the cellular environment, the throughput is larger, whereas for the DCN and WAN, the throughput is small. This is because δ will be 6 for the second data unit in this case, which will make the flows' performance more variable. Hence, we set δ's initial value as 2 to reduce the fluctuations.

H. Overhead

This section evaluates the extra overhead of Antelope when we deploy all the three components (*i.e.* Information Collection, Mechanism Match and Mechanism Switch, see Figure 2) on individual front-end servers. The overhead for Antelope includes three parts: (1) the learning overhead in user space; (2) the information exchange via eBPF; and (3) the CCA 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.

We calculate the overhead of each constituent of overhead by keeping other two unchanged. For example, when we calculate the overhead of switching CCAs, 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 IV 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 [25].

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 CCA at the beginning of a flow and apply new CCAs 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 the cloud could reduce the overhead, which we evaluate in the following section.

I. Overhead Savings via Cloud Learning

Recall, we introduce the concept of cloud-based learning to reduce the overhead introduced by Antelope on end servers. We next evaluate these savings.

Setup. In our testbed, we set one learning cloud server and 10 end servers. Note, the end servers are those that are using Antelope to send traffic to their clients. 5 end servers are in the same DCN cloud with the learning cloud server (all loacted in Shenzhen, China). The other 5 end servers are in different DCN cloud (loacted in Beijing, China) and they connect to the learning cloud server over a WAN. The clients which send file requests to end servers are in the same DCN with end servers.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

TABLE V
BENEFIT AND COST OF CLOUD-BASED LEARNING

Traf. (bps)	Same Cloud					Different Clouds					eBPF memory (MB)
	CPU (%)		Time (ms)			CPU (%)		Time (ms)			
	Cloud	End	Total	Learning	Comm	Cloud	End	Total	Learning	Comm	
40M	4.6	0.5	64	62	2	4.6	0.5	164.6	61.9	102.7	3
120M	4.8	0.9	66.8	64.8	2	5.2	0.8	170.7	65.9	104.8	5
200M	5.2	1.1	70.8	69.1	1.7	5.1	1.2	174.3	69.6	104.7	17
280M	5.3	1.2	71.8	70	1.8	5.9	1.2	178.8	70.3	108.5	17.5
360M	5.3	1.4	71.8	70	1.8	6.2	1.5	182.1	78.6	103.5	19
440M	5.8	1.6	72	70	2	6.7	1.7	186.7	80.9	105.8	22

Table V shows the CPU and time overhead of the whole system. The results are divided into same cloud and different cloud setups. We collect the CPU overhead of both the learning cloud server and the end servers. The time is divided into three parts: (*i*) total; (*ii*) learning time; (*iii*) communication time. The total is the time interval which begins when the end server sends the request to the learning cloud server and ends when it receives the corresponding CC algorithms' names. The learning time is the time that the learning cloud uses to compute the most suitable CC algorithm. The communication time is the time taken to send the (HTTP) requests between the learning cloud and the end server. We change the TCP traffic volumes of the end servers by changing the bandwidth setting by Mahimahi and the clients' file requests (then record the corresponding overhead). For example 40Mbps means that all of the clients' file requests return an average of 40Mbps TCP traffic at each end server. The results are taken as an average across 10 runs.

We further evaluate the memory usage due to the use of eBPF at end servers in Table V. We see that the memory usage increases as the traffic increases. However, it does not multiply with the traffic. This is because we deletes the data recorded by eBPF periodically to reduce memory usage (see Section V-B). Overall, the memory consumption is reasonable in practice.

Benefits. From Table V we see that as the training is moved to the learning cloud, the CPU overhead (introduced by Antelope) at the end server is reduced. Compared to the overhead of performing training locally (see Table V), we find that the load at the end server is to between 0.5% to 1.7%. This is consistent for end servers co-located with the learning server as well as in different clouds.

This is because the end servers offloads the CPU overhead to the learning cloud server. This leaves greater CPU resources for application logic locally. As the number of end servers increase, the relative benefits also increase as the cost can be amortized. Furthermore, we can dynamically adjust the learning cloud resources according to the number of end servers. We conjecture if we use GPU to accelerate the computation process in learning cloud, we could reduce the learning time further.

Costs. To implement cloud-based learning, it is necessary for end servers to communicate with the remote learning cloud servers. The communication latency between the learning cloud and the end servers is an extra cost. Table V presents this latency, confirming that it is very small. Even when we increase the TCP traffic rate, the communication latency remains low at just 1∼2ms. For long flows, the latency for switching CC algorithm is almost negligible. For short TCP flows, as we have the default CC algorithm to use, such influence is also acceptable. That said, for end servers residing in different clouds to the learning server, we see much higher latency, exceeding 100ms. Such latency may not be acceptable, especially for the short TCP flows. Therefore, we recommend to co-locate the end servers and cloud learning server as close as possible (*e.g.* in the same DCN).

VII. RELATED WORK

TCP varieties. We are not the first to observe that CCAs can be optimized for different environments. For instance, Sprout [7], C2TCP [8], ExLL [39] and PBE-CC [40] are specifically designed for cellular networks. Similarly, DCTCP [11], pFabric [12], Timely [41] and Swift [13] are designed for datacenter networks by using Explicit Congestion Notification [42]. BFC [43] achieves near optimal throughput and tail latency behavior in datacenter by hop-by-hop control. TCPLS offers more control to the application by providing multiplexing, connection migration service in TCP [44]. Orca [20], Libra [45], TCP-Drinc [46] and [23] adjust CCAs' parameters (*e.g.* congestion window size) using deep reinforcement learning. In our work, we do not attempt to devise new CCAs or adjust their parameters but, rather, we dynamically select the best algorithm for observed network conditions on demand. DCTCP [11], pFabric [12] and ACC [47] use ECN to infer the network status (such as buffer and channel capacities) and then adjust the congestion. Such mechanisms need the support of routers. Our mechanism only requires end host support and is therefore easier to deploy.

Selection of optimal CCAs. Most related to Antelope is Rein [29], which also tries to select the most suitable CCAs for different networks. Rein relies on rule-based selection. It first classifies the network environment (*e.g.* WiFi or wired) and uses the CCA that is manually designated to this environment. In contrast, Antelope predicts CCAs 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 CCAs and learning mechanisms. TCP-RL [48] is another work that selects suitable CCAs using reinforcement learning. However, TCP-RL implements the selection entirely in user space based on Pantheon [30], which means individual applications need to implement support. To improve CDN performance, Configanator [49] provides programmatic control for web server's configuration parameters (such as CCAs, initRTO, timestamps, version of HTTP and HTTP max frame size). Antelope only targets CCAs, wheres Configanator targets the whole web server. Antelope can be extended to any application (not just a web server) via its kernel-level integration.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

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 [35], and tcp_ebpf [25] 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 [50] 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 CCAs on a per-flow basis. Antelope predicts the most suitable CCAs using machine learning with the network environment, flow states as well as application requirements as input. We have shown that Antelope can successfully apply the optimal or near-optimal CCAs across a diverse range of network types. Through this, we can improve performance without the need for administrators to manually configure their stacks. We have also shown that the extra overhead on individual front-end servers can be greatly saved by implementing the machine learning part in the cloud and sharing it among front-end servers. Antelope paves the way for dynamic selection of CCAs.

References

[1] J. Postel, *Transmission Control Protocol*, document RFC 793, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc793, doi: 10.17487/RFC0793.

[2] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[3] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control," *Netw. Congestion*, vol. 14, pp. 20–53, Dec. 2016.

[4] J. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 4, pp. 270–280, 1996.

[5] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "LEDBAT: The new BitTorrent congestion control protocol," in *Proc. 19th Int. Conf. Comput. Commun. Netw.*, Aug. 2010, pp. 1–6.

[6] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *Proc. Appl. Netw. Res. Workshop*, Jul. 2018, pp. 329–342.

[7] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 459–471.

[8] S. Abbasloo, Y. Xu, and H. J. Chao, "C2TCP: A flexible cellular TCP to meet stringent delay requirements," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 4, pp. 918–932, Apr. 2019.

[9] S. Abbasloo, T. Li, Y. Xu, and H. J. Chao, "Cellular controlled delay TCP (C2TCP)," in *Proc. IFIP Netw. Conf. (IFIP Netw.) Workshops*, May 2018, pp. 118–126.

[10] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *Proc. ACM Conf. Special Interest Group Data Commun.*, Aug. 2015, pp. 509–522.

[11] M. Alizadeh et al., "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 63–74.

[12] M. Alizadeh et al., "pFabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 435–446, 2013.

[13] G. Kumar et al., "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 514–528.

[14] T. Li et al., "TACK: Improving wireless transport performance by taming acknowledgments," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 15–30.

[15] L. Salameh, A. Zhushi, M. Handley, K. Jamieson, and B. Karp, "$HACK$: Hierarchical ACKs for efficient wireless medium utilization," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2014, pp. 359–370.

[16] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, "TCP westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proc. 7th Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2001, pp. 287–297.

[17] K. Winstein and H. Balakrishnan, "TCP ex machina: Computer-generated congestion control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 123–134, Aug. 2013.

[18] M. Dong et al., "PCC vivace: Online-learning congestion control," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 343–356.

[19] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Wanna make your TCP scheme great for cellular networks? Let machines do it for you!" 2019, *arXiv:1912.11735*.

[20] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 632–647.

[21] X. Li, F. Tang, J. Liu, L. T. Yang, L. Fu, and L. Chen, "AUTO: Adaptive congestion control based on Multi-Objective reinforcement learning for the satellite-ground integrated network," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2021, pp. 611–624. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/li-xu

[22] Y. Ma et al., "Multi-objective congestion control," 2021, *arXiv:2107.01427*.

[23] Z. Xia et al., "A multi-objective reinforcement learning perspective on internet congestion control," in *Proc. IEEE/ACM 29th Int. Symp. Quality Service (IWQOS)*, Jun. 2021, pp. 3050–3059.

[24] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with eBPF: Experience and lessons learned," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2018, pp. 1–8.

[25] L. Brakmo, "TCP-BPF: Programmatically tuning TCP behavior through BPF," NetDev 2.2, 2017.

[26] J. Zhou et al., "Antelope: A framework for dynamic selection of congestion control algorithms," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2021, pp. 1–11.

[27] R. Netravali et al., "Mahimahi: Accurate record-and-replay for $HTTP$," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 417–429.

[28] W. Reda et al., "Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 2, pp. 11–23, May 2020.

[29] K. Chen, D. Shan, X. Luo, T. Zhang, Y. Yang, and F. Ren, "One rein to rule them all: A framework for datacenter-to-user congestion control," in *Proc. 4th Asia–Pacific Workshop Netw.*, Aug. 2020, pp. 44–51.

[30] F. Y. Yan et al., "Pantheon: The training ground for internet congestion-control research," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 731–743.

[31] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu, "Interpreting deep learning-based networking systems," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 154–171.

[32] R. K. Yadav, N. Singh, and P. Piyush, "Genetic CoCoA++: Genetic algorithm based congestion control in CoAP," in *Proc. 4th Int. Conf. Intell. Comput. Control Syst. (ICICCS)*, May 2020, pp. 808–813.

[33] A. Giessler, J. Hänle, A. König, and E. Pade, "Free buffer allocation—An investigation by simulation," *Comput. Netw.*, vol. 2, no. 3, pp. 191–208, Jul. 1978.

[34] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 785–794.

[35] *BPF Compiler Collection (BCC)*. Accessed: Aug. 2020. [Online]. Available: https://github.com/iovisor/bcc

[36] A. Saeed et al., "Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Archit., Protocols Comput. Commun.*, Jul. 2020, pp. 735–749.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.

[37] S. Floyd, Ed., *Metrics for the Evaluation of Congestion Control Mechanisms*, document RFC 5166, Mar. 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5166, doi: 10.17487/RFC5166.

[38] F. Yang, Q. Wu, Z. Li, Y. Liu, G. Pau, and G. Xie, "BBRv2+: Towards balancing aggressiveness and fairness with delay-based bandwidth probing," *Comput. Netw.*, vol. 206, Apr. 2022, Art. no. 108789.

[39] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, and K. Lee, "ExLL: An extremely low-latency congestion control for mobile cellular networks," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2018, pp. 307–319.

[40] Y. Xie, F. Yi, and K. Jamieson, "PBE-CC: Congestion control via endpoint-centric, physical-layer bandwidth measurements," 2020, *arXiv:2002.03475*.

[41] R. Mittal et al., "TIMELY: RTT-based congestion control for the datacenter," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 537–550, Oct. 2015.

[42] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2002, pp. 89–102.

[43] P. Goyal, P. Shah, N. K. Sharma, M. Alizadeh, and T. E. Anderson, "Backpressure flow control," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Renton, WA, USA: USENIX Association, Apr. 2022, pp. 1–3. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/goyal

[44] F. Rochet, E. Assogba, M. Piraux, K. Edeline, B. Donnet, and O. Bonaventure, "TCPLS: Modern transport services with TCP and TLS," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2021, pp. 45–59.

[45] Z. Du, J. Zheng, H. Yu, L. Kong, and G. Chen, "A unified congestion control framework for diverse application preferences and network conditions," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2021, pp. 282–296.

[46] K. Xiao, S. Mao, and J. K. Tugnait, "TCP-Drinc: Smart congestion control based on deep reinforcement learning," *IEEE Access*, vol. 7, pp. 11892–11904, 2019.

[47] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 384–397.

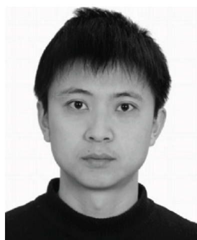
[48] X. Nie et al., "Dynamic TCP initial windows and congestion control schemes through reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, pp. 1231–1247, Jun. 2019.

[49] U. Naseer and T. A. Benson, "Configanator: A data-driven approach to improving CDN performance," in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2022, pp. 1135–1158.

[50] A. Narayan et al., "Restructuring endpoint congestion control," in *Proc. Appl. Netw. Res. Workshop*, Jul. 2018, pp. 30–43.

Jianer Zhou received the Ph.D. degree from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 2016. He is currently a Research Associate Professor with the Southern University of Science and Technology, Shenzhen China. His research interests include network performance, future internet architecture, and internet measurement.

Xinyi Qiu received the Master of Engineering degree from the Computer Network Information Center (CNIC), Chinese Academy of Sciences (CAS), in 2018. She is currently an Engineer with the Department of New Networks, Peng Cheng Laboratory, Shenzhen, China. Her research interests include network performance, future internet architecture, and internet measurement.



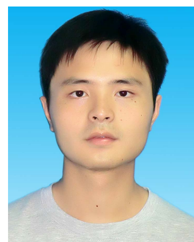
Zhenyu Li (Member, IEEE) received the B.S. degree from Nankai University in 2003 and the Ph.D. degree from the Graduate School, Chinese Academy of Sciences, in 2009. He is currently a Professor at the Institute of Computing Technology, Chinese Academy Sciences. His research interests include internet measurement and networked systems.

Qing Li received the B.S. degree from the Dalian University of Technology, Dalian, China, in 2008, and the Ph.D. degree from Tsinghua University, Beijing, China, in 2013. He is currently an Associate Researcher with the Department of Mathematics and Theories, Peng Cheng Laboratory, Shenzhen, China. His research interests include reliable and scalable routing of the internet, software-defined networking, network function virtualization, in-network caching/computing, edge computing, traffic scheduling, transmission control, and video delivery.

Gareth Tyson received the Ph.D. degree from Lancaster University in 2010. He is currently an Assistant Professor at The Hong Kong University of Science and Technology (GZ). His research interests include internet measurements, web computing, and content distribution.

Jingpu Duan received the B.E. degree from the Huazhong University of Science and Technology in 2013 and the Ph.D. degree from The University of Hong Kong in 2018. He is currently an Assistant Researcher with the Department of Broadband Communication, Peng Cheng Laboratory. His research interests include designing and implementing high-performance networking systems.

Yi Wang (Member, IEEE) received the Ph.D. degree in computer science and technology from Tsinghua University in July 2013. He is currently a Research Professor with the Sustech Institute of Future Networks, Southern University of Science and Technology. His research interests include future network architectures, information centric networking, software-defined networks, and the design and implementation of high-performance network devices.



Heng Pan received the Ph.D. degree in computer science from the University of Chinese Academy Sciences in 2018. He is currently an Associate Professor at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include SDN/NFV, distributed systems, and in-network computation.

Qinghua Wu received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2015. He is currently an Associate Researcher at the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include network transport protocol and internet measurements.

Authorized licensed use limited to: Peng Cheng Laboratory. Downloaded on December 27,2023 at 06:41:03 UTC from IEEE Xplore. Restrictions apply.