Muses: Enabling Lightweight Learning-Based Congestion Control for Mobile Devices

Zhiren Zhong[*†‡], Wei Wang[†], Yiyang Shao[†], Zhenyu Li[*‡¶], Heng Pan[*¶],
Hongtao Guan[*], Gareth Tyson[‖x], Gaogang Xie[‡§], Kai Zheng[†]
[*]Institute of Computing Technology, Chinese Academy of Sciences, [†]Huawei, China,
[‡]University of Chinese Academy of Sciences, [§]Computer Network Information Center, Chinese Academy of Sciences,
[¶]Purple Mountain Laboratories, China, [‖]Queen Mary University of London,
[x]Hong Kong University of Science & Technology
{zhongzhiren,zyli,pangheng,guanhongtao}@ict.ac.cn, xie@cnic.cn,
{wangwei375,shaoyiyang,kai.zheng}@huawei.com, gareth.tyson@qmul.ac.uk

***Abstract*—Various congestion control (CC) algorithms have been designed to target specific scenarios. To automate this process, researchers have begun to use machine learning to automatically control the congestion window. These, however, often rely on heavyweight learning models (*e.g.,* neural networks). This can make them unsuitable for resource-constrained mobile devices. On the other hand, lightweight models (*e.g.,* decision trees) are often incapable of reflecting the complexity of diverse mobile wireless environments. To address this, we present *Muses*, a learning-based approach for generating lightweight congestion control algorithms. *Muses* relies on imitation learning to train a universal (heavy) LSTM model, which is then used to extract (lightweight) decision tree models that are each targeted at an individual environment. *Muses* then dynamically selects the most appropriate decision tree on a per-flow basis. We show that *Muses* can generate high throughput policies across a diverse set of environments, and it is sufficiently light to operate on mobile devices.**

I. Introduction

Congestion control (CC) has been a popular research topic for over 30 years. Many congestion control algorithms have been proposed, often relying on domain specific expertise to "hard-code" algorithms that manage the congestion window (*cwnd*) based on a range of signals, including packet loss [1], [2], jitter [3], [4], ECN [5], [6], BDP variation [7], and hybrids [8], [9].

The limitation of these hard-coded algorithms is that they rely on assumptions about the underlying network. As these assumptions feed into the corresponding control function (*e.g.,* linear [2], cubic [1], inverse proportional [3]), these algorithms may perform poorly when deployed in networks in which the assumptions do not hold. For instance, it has been proven that the use of traditional (*e.g.,* Reno) CC algorithms in wireless networks is problematic [10] [11]. This is because wireless networks have two major characteristics that make CC challenging: large variability in channel capacity, and "noisy" congestion signals that are difficult to interpret. These two obstacles make it difficult to manually design a suitable control logic based on assumptions related to feedback signals for *all* wireless networks.

To overcome this, recent proposed learning-based algorithms [12]–[17] have tried to avoid hard-coding heuristics. Instead, they strive to autonomously *learn* the optimal congestion policy. For example, PCC [13] and Vivace [17] try to decide how to change the sending rate using online optimization theory; whereas, Indigo [14] generates a neural network model by training offline across a wide range of network parameters.

The performance of these learning-based algorithms has shown notable gains compared to hard-coded algorithms [18]. However, there are significant challenges when porting them to mobile wireless environments. Most notably, many rely on heavyweight models such as LSTMs [19]. These have high computational and memory costs, making them infeasible for low cost mobile devices. For example, loading an LSTM (with 2 layers and 256 units) on an Android device uses about 74MB of memory, with inference times reaching 25 ms (Section II-C).

An obvious solution would be to rely on lightweight models such as decision trees. However, in most cases, they are not rich enough to deal with the diversity of wireless scenarios. For example, we find that when working in a stable environment (*i.e.,* where the characteristics of the underlying network are consistent), a decision tree gets similar F1-Score prediction performance compared to the LSTM. However, it is much worse (9%) when the environment becomes dynamical (*i.e.,* the characteristics of the underlying network are diverse), as shown in Section II-C.

Motivated by the above observations, we seek a solution for a lightweight learning-based congestion control scheme for mobile devices to offer both low overhead and high performance. We underpin our approach with one key insight: it is not necessary to train a model that performs well in *all* networks, rather we can pre-train multiple models for different networks and select the most appropriate one at runtime.

This paper proposes a multi-stage learning-based CC scheme, named *Muses*. *Muses* generates multiple lightweight decision trees by extracting and decomposing the policy of a

The work of Zhiren Zhong was completed during his internship at Huawei.

universal heavyweight model LSTM. *Muses* then selects the most feasible decision tree on a per-flow basis. Specifically, *Muses* follows three stages: (*i*) It leverages Imitation Learning (IL) to train an LSTM in a comprehensive "global" environment. (*ii*) It decomposes the LSTM into many decision trees for different sub-environments based on a K-means clustering method. (*iii*) It selects the most appropriate decision tree at runtime for each individual flow.

Our experiments show that the lightweight model significantly reduces *inference time* and *memory usage* compared to the heavyweight model. For example, on average, it achieves 4.2us inference time and 61KB runtime memory, which are 3 orders of magnitude superior to the heavyweight model. It also attains both higher throughput and lower delay compared to typical hard-coded (*e.g.,* Cubic, Verus) and learning-based (*e.g.,* Indigo, Orca) algorithms. For example, *Muses* reaches 97.8% bandwidth utilization, and achieves 2.29× lower 95th percentile one-way queuing delay compared to Cubic in a steady Wi-Fi environment; in real LTE networks, it gains 1.75× higher throughput than Verus [20], while reducing delay by 9%. Finally, we integrate *Muses* into WebRTC [21] and show that it achieves a higher video bitrate (1.34×) with similar frames per second compared to the default GCC algorithm [22].

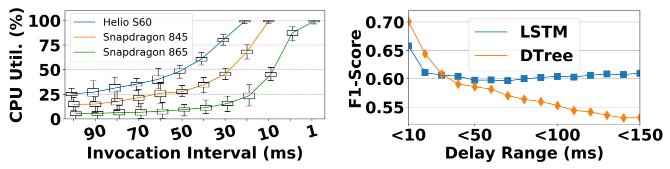
II. Background & Motivation

A. Wireless CC is Hard for Hard-Coded Algorithms

Whereas in prior years, mobile devices acted largely as data recipients, they have since become huge data generators. Social media (*e.g.,* TikTok,), backup services (*e.g.,* Dropbox) and live video streamings (*e.g.,* Facebook Live) have meant that mobile devices frequently upload large volumes. This means that building CC algorithms tailored to *sending* from mobile devices has become a key research challenge.

Variability of Channel Capacity. A key difference between a wireless link and a wired link is the variability of its link capacity for the user. The available wireless capacity depends on many conditions (*e.g.,* user's competition, radio signal path loss, channel interference, noise, MCS scheme) [23]. Some link/physical layer technologies such as Carrier Aggregation in 4G and 5G Network, Frame Aggregation in Wi-Fi also affect how the user's bandwidth changes. Human-designed decisions to increase/decrease the congestion window (such as AIMD [2], cubic function [1]) in hard-coded CC algorithms find it hard to match such changes in fluctuating and variable wireless link capacity.

Noisy Congestion Signals. Hard-coded CC algorithms usually rely on signals (*e.g.,* packet loss [1], delay delta/gradient [20], [24]) or estimates of underlying network conditions (*e.g.,* max_btlbw [7]). However, these signals and estimates could be inaccurate due to unreliable link quality and variable settings. For example, in the Unacknowledged Mode (UM) of LTE, the fail recovery by HARQ only recovers a small part of random loss packets, resulting in a significant throughput degradation for loss-based CC [25]. While in Acknowledged Mode (AM), the mechanism of reordering in the Radio Link Control (RLC) layer leads to head-of-line blocking. Then the delay jitter and the burst delivery of packets result in overestimated estimations of bandwidth [25]–[27].



(a) CPU Utilization (b) Prediction Performance

Fig. 1: Overhead and Performance for LSTM vs Tree on Typical Flagship, Mid-range and Low-end Mobile Phones

B. Advantages of Learning-based CC

Compared with hard-coded CC, learning-based CC has advantages in the following aspects. First, using more signals as an indicator may provide a less biased estimate of network states. The increased number of signals makes algorithm design more cumbersome for human developers, but is easy for learning-based models. Second, the decision-making of a learning model is more fine-grained and can better match changes in various network conditions (such as changes in link capacity in different degrees). Finally, statistical learning can better reveal the relationship between the states observed at end-points and the actual congestion states inside the network. We can also train models to better identify signals (from the noise). Several prior results [12], [14] have confirmed that learning-based approaches outperform hard-coded algorithms.

C. Learning-based CCs are Impractical on Mobile Devices

Despite better performance, learning-based congestion control algorithms are not always suitable for low cost mobile devices due to limited hardware resources.

Setup. To test these limitations, we reproduce Indigo's learning framework and train an LSTM model with 2 layers of 256 units [19] and a decision tree with 500 leaf nodes. We use the same input/output space but training in an environment composed of the networks of which parameters are drawn from the ranges defined in Table II. In this paper, we abstract the network and environment from the end-point perspective, where a network is a series of data packet delivery behaviors tied to a flow transmission task and a (network) environment is a group of these networks. More details about constructing the environment is in Section III-D. As a learning goal, we train the models to estimate link BDP (*i.e.,* the optimal congestion window) in real-time.

Overheads of Heavyweight Model on Mobile Devices To test the resource overhead, we run these models on three typical flagship, mid-range and low-end mobile phones (with Qualcomm 865, Qualcomm 845 and Helio S60, respectively). We measure the CPU utilization by executing the models with the different invocation frequencies. Figure 1a shows the result of the LSTM. For mid-range devices, the CPU usage increases from 15% to 100% as the invocation interval goes from 100ms down to 10ms. Typically, the congestion window should be updated every ACK, or a fixed interval related to RTT (*e.g.,*, 10ms [14]). For a typical 10ms invocation interval, this means the model consumes a remarkable 45% of CPU usage, even on flagship SoCs. In contrast, for the decision tree, the CPU utilization is less than 1% even with a 1ms invocation interval on the low-end SoCs (not shown in the figure).

Inefficacy of One-size-fits-all Lightweight Model. Next we see how well the models learn in environments with differing complexity. We train LSTM models and decision trees in several environments. The ranges of delay parameter of these environments are set to different upper bounds and the others are the same as Table II. A larger range means that the environment is more complex.

For each environment, we run 1K tests for each model and compute their F1-Scores. As shown in Figure 1b, the score of the decision tree continues to decrease as the delay range widens, but the LSTM remains at the same level. When the environment becomes more complex, the gap between the decision tree and the LSTM becomes larger. For this reason, training a one-size-fit-all lightweight model is not feasible because it is not sufficiently expressive to capture all networks.

D. Implications on Design

To summarize, learning-based models have superior performance than hard-coded algorithms in mobile networks. Heavyweight models suffer from high resource overheads, while lightweight ones yield sub-optimal performance. To mitigate this, our key idea is to generate targeted lightweight models that are specialized for particular sub-environments. A device can then dynamically select the most appropriate model on a per-flow basis. To achieve this, there are two remaining challenges: (*i*) How to determine the scope of a sub-environment and train the corresponding decision trees. (*ii*) How to dynamically match the correct model for each flow.

III. Design of *Muses*

A. Primer on Muses

We first provide a high-level primer on *Muses*, which operates in three stages (see Figure 2). An introduction of *Muses* framework is presented in Section III-B. First, we train a heavyweight LSTM (Section III-C) model within an emulated network environment (Section III-D). Using Imitation Learning (IL) [28], the goal is to build a comprehensive "global" CC model that can mimic the optimal congestion window in a wide range of network scenarios. Second, we decompose the CC policy (*i.e.,* a mapping from observed state to action used to adjust the congestion window) learned by the LSTM into a series of lightweight decision trees (Section III-E) that can easily execute on low cost mobile devices. Each decision tree is capable of imitating the LSTM's decision behavior in a specific sub-environment. Third, each flow dynamically selects the decision tree that best match its own environment (Section III-F).

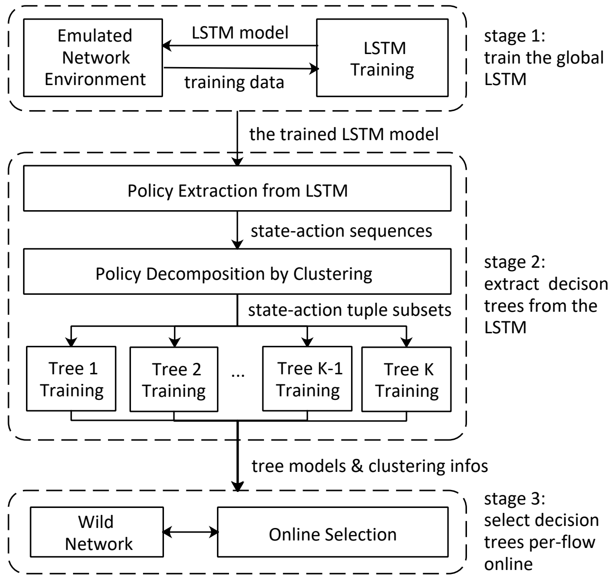

Emulated Network Environment
LSTM model
training data
LSTM Training
stage 1: train the global LSTM
the trained LSTM model
Policy Extraction from LSTM
state-action sequences
Policy Decomposition by Clustering
state-action tuple subsets
Tree 1 Training
Tree 2 Training
...
Tree K-1 Training
Tree K Training
stage 2: extract decison trees from the LSTM
tree models & clustering infos
Wild Network
Online Selection
stage 3: select decision trees per-flow online

Fig. 2: Framework of *Muses*

B. Framework of Muses

The framework of *Muses* is shown in Figure 2. In stage 1, an IL task is launched for training a universal LSTM to learn the best policy in a comprehensive global environment (Section III-C). To construct this environment, the *environment constructor* generates a large number of emulated networks by configuring a wide range of network parameter settings (Section III-D).

In stage 2, *Muses* extracts the learned policy in the LSTM and decomposes it into different sub-policies to train decision trees (Section III-E). To this end, it first runs a sender to do flow transmissions in many networks drawn from the global environment, using the LSTM as its CC algorithm. Through this, it can gather the set of state-action sequences, which contain the global knowledge of the LSTM's policy. To decompose this policy, the sequences are clustered into multiple clusters by a K-means clustering method [29], which uses the information of the state-action distribution of sequence. For each cluster, all state-action tuples of sequences in this cluster are added into a subset. Each subset represents a sub-policy that is suitable for a sub-environment, which is consisted of those networks from which the corresponding state-action tuples are collected. For each subset of state-action tuples, a decision tree is trained.

In stage 3, *Muses* stores all the trained decision trees on a mobile device. During runtime, when launching a flow transmission, it selects the most appropriate decision tree by comparing the state-action distribution of the online flow against the cluster centers (a cluster center is regarded as a weight distribution of those sequences in that cluster). In particular, it can switch to a new decision tree when the current distribution changes, usually because of underlying network change, *e.g.,* access type switch, handover (Section III-F). The rest of this section details the three stages of *Muses*.

C. Stage 1: Training the Global LSTM

We treat the LSTM as a classification model. Once trained, an inference is performed using the LSTM upon every congestion control interval (*ci*, usually 10ms). This is triggered by ACK or an RTT-based timer. The goal is to take a state $s \in \mathcal{S}$, observed on the end-point as an input and then output an action, $a \in \mathcal{A}$, to set the current congestion window. Thus, the learning task is to train a model to fit the best policy $\pi^* : \mathcal{S} \to \mathcal{A}$ for a specific network environment.

Learning Process. The learning process of the LSTM is run over a series of iterations in a global environment constructed by the *environment constructor* (Section III-D). The configurations of this environment are in Table II. Each iteration includes two steps: model evaluation and model training. In the evaluation step, a sender performs a flow transmission in a random emulated network drawn from the global environment, using the LSTM as its CC algorithm. During transmission, the LSTM model periodically performs inference with an input state, s_t, at the time point q_t, until it repeats L times (default is 1000). Given the q_t, the best action b_t of each s_t is calculated by the *Oracle* interface of the *environment constructor*. After the transmission finishes, a flow data sequence $(..., (s_t, b_t), ..)$ is added as a sample to a training set D. This step is repeated in many different emulated networks until the LSTM converges, so that the training is exposed to a diverse set of scenarios.

In the training step, Tensorflow is used as the training platform. AdamOptimizer [30] is used as the optimization process to reduce the average cross entropy loss over D. This optimization process is executed dozens of epochs at each training step. After that, a retained LSTM will be launched for the next iteration's evaluation. The termination condition is when the ratio of average loss is below a small value, ϵ (default is 5-e3), for a number of iterations (*e.g.,* 100). The ratio of average loss is defined as the changing rate of the mean average loss of all optimization epochs between current iteration and the previous 10 iterations.

Input and Output. The input state is composed of serval raw congestion signals, such as RTTs and packet loss. These are normalized by an Exponentially-Weighted Moving Average (EWMA) or max-min filter. All signals used in *Muses* are listed in Table I.

Each time a congestion event is detected, the state is updated. The action, a, is taken from a predefined action space $\mathcal{A} = \{/M_n, ..., /M_1, +0.0, *M_1, *M_n\}$, where $M_n \in (1, 2]$ and $M_{n-1} < M_n$. Each action is an operation that determines how to expand or reduce the congestion window. In this paper, the action space is set to $\mathcal{A} = \{/2.0, /1.2, /1.05, +0.0, *1.05, *1.2, *2.0\}$. Note that this space can be tuned by system designers, and our approach is not tied to any specific action space.

D. Emulated Network Environment Construction

Stage 1 relies on a global environment that contains various emulated networks to support to train a universal LSTM. we design an *environment constructor* to construct such an environment. The *environment constructor* creates an emulated network by seeding random parameters from a given range. We parameterize the emulated networks using two factors: (*i*) network-layer conditions, *e.g.,* packet loss; and (*ii*) wireless channel conditions, *e.g.,* noise.

TABLE I: Model Input State

Signal	Description
sRTT	The RTT smoothed by EWMA since last control interval (*ci*).
qdelay	The queuing delay ratio smoothed by EWMA since the last *ci*. Delay ratio is queuing delay divided by minimum RTT.
minRTT	The minimum RTT since connection establishment.
srate	The sending rate smoothed by EWMA since the last *ci*.
drate	The delivery rate smoothed by EWMA since the last *ci*.
loss	An estimated loss rate, defined as 1 minus the ratio of bytes received and bytes sent during the last *ci*.
cwnd	The congestion window at the end of last *ci*.

Emulating Network-layer Conditions We use Mahimahi [31] as the emulator to emulate a network. The emulated network contains two paths: the data path and the ACK path. The data path is configured using a tuple $(trace, delay, loss, buffer)$, where $trace$ is a trace of underlying wireless channel conditions, $delay$ is one-way delay, $loss$ is a random loss rate, and $buffer$ is the max queuing size. For simplicity, we assume that the bandwidth bottleneck link is always in the data path since ACKs are much smaller than data packets.

Emulating Wireless Channel Conditions. Mahimahi relies on wireless channel traces to model the arrival times of packets. We obtain these traces from two sources: (*i*) Existing LTE/3G traces, bundled with Mahimahi [32]; and (*ii*) Newly simulated traces generated in Mininet-wifi [33]. The former are collected from different cellular networks (LTE, UMTS and EVDO) of several ISPs (AT&T, Verizon and T-Mobile). We generate the latter from an emulated Wi-Fi link in a two-hop topology from a mobile client $\to$ access point $\to$ Server. We run Saturator [12] to record the packet arrival times as the wireless link trace. Such a trace is defined by a four-tuple configuration: $(ple, variance, location, speed)$. The path loss exponent ple and Gaussian noises level $variance$ are two parameters of the *Log-Normal Shadowing Propagation Loss Model* [23]. The latter two are used in a random waypoint mobility model: a client moves in a straight line at a random speed $speed$ from a random starting point $location$ (*e.g.,* the distance to the access point) and moves to another random endpoint. When it reaches the endpoint, it selects a new random point to move to.

With the above model, the *environment constructor* randomly selects parameter values in the ranges defined in Table II which cover typical real environments to generate a trace and create an emulated network for each evaluation step. It is worth noting that constructing this global environment does not ensure that it includes *all* the networks in the real environment, but it is complex enough to be used to verify the effectiveness of *Muses* that decomposing the complex environment *could* improve performance.

Calculation of Optimal Congestion Window To generate data samples for LSTM training, we require ground truth knowledge of the optimal congestion window. Thus, the *environment constructor* creates an *Oracle* interface that computes the optimal congestion window. It does this as follows.

TABLE II: Environment Configurations for LSTM Training

Parameter		Range	
		Wi-Fi	LTE/3G
trace	ple	4-6	ATT and TMobile
	variance (dB)	0-20	
	location (m)	0-500	
	speed (m/s)	0-10	
one-way delay (ms)		5-150	
loss		0-5%	
buffer (KBytes)		1.5-3000	

By ensuring that the ACK path is not congested and ignoring queuing and processing delay, an un-congested data path should see an ACK returned exactly one propagation RTT after a packet is sent. With this assumption, we compute the path BDP as the volume of data sent on the data link in the last RTT. To alleviate the bias of the assumption, a configurable scaling factor, α, is used such that the estimated BDP is $\alpha * BDP$. It will achieve higher throughput if $1 \leq \alpha < 2$, and lower latency in the case of $\alpha < 1$.[1] We set α to 1.2 in this paper to achieve high throughput.

The calculation of the *optimal cwnd* at time t (denoted as ow_t), is shown in Eq. 1. Here, $f(x)$ refers to the bandwidth (bytes/s) at time x and it can be deduced using $trace$. $delay$ is the one-way delay of the data path, m is the packet size, and α is the configurable scaling factor. The optimal action, b_t, is the action to operates *current cwnd* (denoted as cw_t) that make it closest to ow_t. The definition of b_t is in Eq 2.

$$ow_t = \frac{\alpha}{m} \cdot \int_{t-2\cdot delay}^{t} f(x)\, dx \tag{1}$$

$$b_t = \underset{a \in \mathcal{A}}{\operatorname{argmin}} |ow_t - op(cw_t, a)| \tag{2}$$

E. Stage 2: Extracting Lightweight Decision Trees from LSTM

We have seen in Section II-C that the decision tree achieves similar performance to the LSTM model in simple environments. Therefore, we extract the LSTM's policy to train a series of decision trees, where each tree corresponds to a sub-environment that is much simpler than the global one.

Policy Extraction From LSTM We first extract the LSTM policy as many state-action demonstration sequences. To this end, *Muses* runs thousands of flow transmissions using the LSTM (from Stage 1) as the CC algorithm in the global environment. Specifically, at round i, a flow sequences $f_i = (..., (s_t, a_t), ...)$ is added into a set D_E, where s_t is the input state of LSTM and the a_t is the output action performed by the LSTM. D_E includes the global knowledge of LSTM about how to act (*i.e.,* adjust congestion window) for various situations in the global environment. In our current setup, the total number of rounds is 5000 and D_E contains 5 million state-action examples.

Here we use the action a_t of LSTM instead of the b_t provided by *Oracle* is because the LSTM has learned the generalization knowledge in the global environment, which is helpful to alleviate the over-fitting of the decision tree: a decision tree trained only with the *Oracle* actions for the corresponding sub-environment gets poor performance when working in another unmatched sub-environment. The situation is possible to happen when the network conditions suddenly change before switching to a new decision tree. However, the action of LSTM is a trade-off that takes into account the performance loss of other sub-environments. Hence, using a_t makes the decision tree become more robust. Note, this step of generating a_t is inspired by the policy extraction in Metis [35].

Policy Decomposition by Clustering. It is next necessary to divide D_E into multiple disjointed subsets for training decision trees. Each subset has the state-action tuples collected from a group of networks. Thus, it reflects the sub-policy for a sub-environment composed of these networks. The sub-policy contains partial knowledge of the LSTM about how to act in this sub-environment.

We use a clustering method to separate D_E to these subsets. The number of subsets, K, is defined in advance and the state-action tuples of each $f \in D_E$ are assigned to one of the subsets. The state-action distribution of f is used to capture the characteristics of its underlying network. Those tuples in flows which have similar state-action distributions are grouped into the same subset, which indicates their corresponding networks are similar. Specifically, we use a distribution feature vector $P(f)$ derived from the original state-action distribution as the data point for the clustering algorithm. $P(f)$ is computed as follows.

First, the space of the input state is reduced to a feature space $\mathcal{X}$ of n (3 by default) dimensions through Principal Component Analysis (PCA). Each state-action tuple (s, a) in f is transformed to a feature-action tuple (x, a). Second, for each f, the discrete state-action distribution $P_f(X_i, A)$ of each dimension X_i is calculated and these distributions are concatenated to obtain $P(f) = P_f(X_1, A) \circ ... \circ P_f(X_n, A)$. In order to calculate $P_f(X_i, A)$, the range of X_i's value is separated into l (default is 10) segments, and then the frequency of each action in each segment is counted based on the tuples in f. Using $P(f)$ instead of the original state-action distribution is to reduce the dimensionality. The dimension of $P(f)$ is $n \cdot l \cdot |\mathcal{A}|$, while the dimension of the original state-action distribution is $l^{dim(\mathcal{S})} \cdot |\mathcal{A}|$, where $dim(\mathcal{S})$ is the dimension of state space, $|\mathcal{A}|$ the size of action set. The latter increases with the power of l, which is infeasible for clustering. When we group the data that has similar state-action distributions to the same subset, we actually obtain a lower conditional entropy $H(A|S)$ on all tuples in D_E compared with no division or random division. $H(A|S)$ can be interpreted as the uncertainty about A when S is known. The lower the uncertainty of the dataset, the simpler the training of the model.

After computing $P(f)$ for each $f \in D_E$, we perform clustering on the set $\{P(f)|f \in D_E\}$. We use the K-means method [29] in Sklearn for this purpose. To get the best number of clusters, K, we run K-means by configuring K from 2 to 100 to measure the Silhouette Score [36] of clustering results. We find $K = 15$ is the most suitable value. When

[1]The upper bound of 2 borrows from the design of BBR [34], which limits the algorithm to injecting more than twice the BDP in-flight packets.

the clustering is complete, we group each $(s, a) \in f$ to the corresponding data subset $D_E^j, j \in [1, K]$, where j is the output label of $P(f)$ in clustering.

Training Decision Trees. Once we have decomposed D_E into K sub-datasets, we train a decision tree for each D_E^j. Each tree will learn the local policy for the corresponding sub-environment. Collectively, these decision trees contain the knowledge of the LSTM, but in a far lighter form. We use CART [37] to train the decision tree, and use information entropy as the measure of node impurity, since there is only 2% difference between information entropy and Gini index across many cases [38]. When training a decision tree, the splitting criterion is to select the feature component that maximizes the information gain for division. The hyper-parameters of decision tree are set as follows: the *max_leaf_nodes* is set to 500 to avoid over-fitting and use grid searching to decide the best setting for parameters of *min_samples_split* and *min_impurity_decrease*. Other parameters are defaults.

F. Stage 3: Selecting Decision Trees Per-Flow

Whereas stage 1 and 2 take place centrally, stage 3 is executed in-the-wild on the end device. At runtime, a flow selects the most appropriate decision tree model for its own environment. Note, for long flows, the model can be switched multiple times.

Online Selection. All decision trees along with the corresponding cluster centers (*i.e.,* the mean of all $P(f)$ in the same cluster) are stored locally in a mobile device. During a flow transmission, *Muses* selects the decision tree of which the corresponding center is closest to the distribution feature vector $P(f')$ of the online flow f'. To calculate $P(f')$, the state sample s_t is collected every ci. It collects T_l time samples to estimate $P(f')$. We set ci to 10ms and set T_l to 10s by default. At the beginning of a flow transmission, the decision tree that was selected in the previous transmission is used.

State Distribution Transformation. Note that the online samples only contain states and no LSTM's actions, which makes it impossible to calculate $P(f')$ directly (Running the LSTM locally for labeling is expensive, and uploading the samples requires a lot of traffic overhead).

In order to address this problem, we use a matrix Q to transform $P_s(f)$ to $P(f)$, where $P_s(f) = P_f(X_1) \circ ... \circ P_f(X_n)$. It includes the state margin distribution $P_f(X_i)$ instead of the state-action joint distribution $P_f(X_i, A)$. To get Q, we define a matrix $Q_f = [P(f_1)^\intercal, P(f_2)^\intercal, ...]$ and a matrix $Q_f^s = [P_s(f_1)^\intercal, P_s(f_2)^\intercal, ...]$ based on D_E, and then solve the linear equation $Q_f = Q \cdot Q_f^s$. After estimating $P_s(f')$ based on the online state samples, the estimation of feature vector is calculated: $\hat{P}(f') = (Q \cdot P_s(f')^\intercal)^\intercal$. It is worth noting that $\hat{P}(f')$ is not exactly equal to $P(f')$, even for the f in D_E, since Q_f^s usually is not a non-singular square matrix. Nevertheless, we find through evaluations that this bias is acceptable. Specifically, for all f in D_E with $K = 15$, the proportion that the nearest cluster center of $\hat{P}(f)$ is not in the first two nearest centers of $P(f)$ is less than 3%.

IV. Evaluation

In this section, we first evaluate that *Muses* can attain superior performance to existing hard-coded and learning-based CC algorithms. We then show that using the decision tree can reduce the overhead of running learning-based models, enabling practical deployment on low cost mobile devices. To further confirm this, we present a use case implementation of *Muses* in *WebRTC* to improve video chat QoE.

A. Performance in Emulated Networks

Setup. We use the *environment constructor* as introduced in Section III-D to build a set of emulated networks to test *Muses*. We experiment with three typical environments, as shown in Table III. The first two are generated from Wi-Fi scenarios and the last is composed of Verizon LTE traces taken from [32]. The two Wi-Fi parameter configurations are designed to emulate both static and high-mobility environments. For each emulated environment, we perform tests by randomly selecting parameters from the ranges defined in Table III. In total, the number of test cases is 1,000, and each test runs a flow transmission for 100 seconds for each algorithm. It is worth noting that the traces used for evaluation in this section have *not* been used for model training.

As a basline, we compare against existing off-the-shelf learning-based algorithms (Indigo, RemyCC, Vivace, Orca), hard-coded ones (BBR, Cubic, Verus, Sprout).[2] We capture three evaluative metrics: bandwidth utilization, the 95th one-way delay, and packet loss rate. We also compare against a decision tree (denoted as *TreeG*) which is trained in the global environment in the same fashion as the LSTM, *i.e.,* without breaking down the environment into subsets.

Bandwidth Utilization & Delay. Figure 3a-3c report the bandwidth utilization vs. 95th percentile one-way delay for each environment. We use the performance results of all test cases to draw a 2D Gaussian distribution. The width and height of the ellipse area are one standard deviations for each independent direction. Superior performance (in terms of bandwidth utilization and lower delay) is indicated by placement on the upper left corner of the graph.

In comparison with TreeG, *Muses* improves both the bandwidth utilization and delay significantly. Furthermore, *Muses* attain high throughput (average bandwidth utilization is $>$ 90% in all three environments) while maintaining low latency compared to the other algorithms. For example, *Muses* reaches 99.7% and 97.8% utilization in the static and high-mobility Wi-Fi environments respectively, while reducing delay by over $2\times$, compared to Cubic. In the LTE environment, *Muses* on average achieves $1.26\times$ higher bandwidth utilization and $1.29\times$ lower delay, compared to Verus (an algorithm designed for cellular network). Finally, with similar delay as in Indigo, *Muses* achieves $1.1\times$ higher average bandwidth utilization in all three environments.

[2]For BBR, Cubic and Orca, we use the version implemented in Linux 4.13.1 with the Orca patch. Other algorithms are cloned from Pantheon [18]. Note that RemyCC is trained in the 100× range of link rates environment described in [39], and the RL model of Orca is provided in [40].

We next inspect how these results are close to the optimal ones. Thus, we randomly select 100 test cases from the high-mobility Wi-Fi environment. Figure 4 presents the scatter-plot comparing the real-time link BDP vs. the in-flight packets for each algorithm. The dashed line represents the optimal performance, indicating that the link BDP and the in-flight packets are equal. We see that the BDP varies a lot during the experiments, as channel conditions change. The scattered points of *Muses* show a strong positive correlation, with the majority of window sizes clustered around the optimal line. To measure how close an algorithm is to the optimal we compute the RMSE between the link BDP and the real in-flight packets.

Muses attains $3.17\times$, $1.78\times$ and $2.25\times$ lower RMSE compared to BBR, TreeG and RemyCC, respectively. BBR is consistently the most aggressive, with more in-flight packets than the BDP can accommodate. This is because BBR allows $2\times$BDP (estimated). However, in a wireless network, the BDP is particularly hard to estimate. Besides, compared with the *Muses*, both TreeG and RemyCC have a far wider range of window sizes. Note that RemyCC is trained for the general environment, where its simple tree structure and insufficient input state space (4–5 states) may not be sufficient to capture the dynamics of wireless networks. Instead, *Muses* performs better since each decision tree is trained for a particular sub-environment.

Consistency. Figure 3a-3c also show that *Muses* attains more consistent performance across different networks, due to its ability to dynamically switch policies for the target environment. To measure consistency, we use the ellipse area (smaller is better). Visually, one can see that *Muses* has substantially smaller variation compared to other algorithms. For example, *Muses* obtains $0.95\times$ (in steady Wi-Fi), $0.62\times$ (in fluctuating Wi-Fi), and $0.93\times$ (in LTE) the ellipse area of the second best algorithm in each environment. Importantly, while other algorithms are excellent in individual environments (*e.g.,* Vivace in Steady Wi-Fi), they fall behind in alternates (*e.g.,* Vivace in Verizon LTE). In contrast, *Muses* performs consistently well across all, as it dynamically switches to the most appropriate model based on observed conditions.

Packet Loss. As well as bandwidth utilization and delay, we inspect the average packet loss rate (PLR) across the 1,000 test cases for each environment. Figure 3d shows the average PLR for each algorithm. Compared to the other algorithms, *Muses* has very low loss rates ($< 0.3\%$ in all three environments). For example, *Muses* attains $< 0.09\%$ loss rate but achieves $1.66\times$ higher throughput than Sprout in the steady Wi-Fi environment. *Muses* achieves almost the same packet loss rate as Indigo. The other two learning-based algorithms (RemyCC and Vivace) get fairly high loss rates though ($> 2\%$ in all environments). Curiously, we also observe that RemyCC makes unreasonable decisions at certain points: it generates sporadic bursts, which results in a large number of in-flight packets, triggering loss. In contrast, we do not observe these sudden bursts for *Muses*.

TABLE III: Environment Configurations for Evaluation

Parameter		Range		
		Steady Wi-Fi	Fluctuating Wi-Fi	LTE
$trace$	ple	4-6	4-6	Verizon LTE
	variance (dB)	3	10	
	location (m)	5	20	
	speed (m/s)	0	1.2	
one-way delay (ms)		20-30	20-50	20-50
loss		0	0.1%	0
buffer (KBytes)		300	300	300

B. Performance In-the-Wild

Setup. We next evaluate *Muses* in-the-wild. We test two different access networks (Wi-Fi and LTE, both are routed through to the same ISP) at location A communicating with a remote private server located at B. The mobile client acts as the sender and the server acts as the receiver. The geographical distance between these two locations is 1300km. The min/avg/max/std RTT on the Wi-Fi path are 44.73ms/50.02ms/143.50ms/9.49ms and on the LTE path are 53.96ms/59.55ms/95.274ms/5.41ms. We again compare *Muses* with BBR, Cubic, Verus, TreeG and Vivace. In total, we run 100 tests, where each test consists of 60 seconds of data transmission from the client to the remote server.

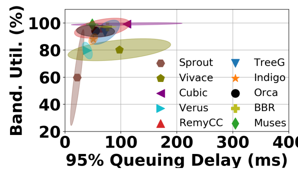
Throughput & Delay. The results are shown in Figure 5. We plot the throughput vs. 95th percentile RTT distribution for Wi-Fi (left) and LTE (right), as we did in the emulated experiments. The results show that *Muses* achieves both higher throughput and lower delay compared to TreeG, which further confirms the effectiveness of our method. It also has superior performance compared with other algorithms. Specifically, in the Wi-Fi network, *Muses* achieves $1.68\times$ lower delay compared to Cubic, while only decreasing throughput by 4.1%. Vivace achieves the lowest delay (about $1.49\times$ lower than *Muses*), but its throughput is only 15.6% of *Muses*. In the LTE network, *Muses* attains $1.09\times$ lower delay than Verus, and $1.75\times$ higher throughput. The above results confirm that the performance of *Muses* in real networks are consistent with the emulated networks.

C. Overhead

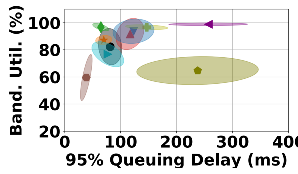
A key design goal is ensuring *Muses* is sufficiently light to run on low cost mobile devices.

Setup. To evaluate the resource overhead, we first train several decision tree models on the global environment of different complexities (referred as $TreeG_x$, of which x indicates the setting of *max_leaf_nodes*). We use the LSTM in Section III-C as a heavyweight comparison and RemyCC as a lightweight comparison. We test these algorithms on an Android device with Snapdragon 845 Soc.

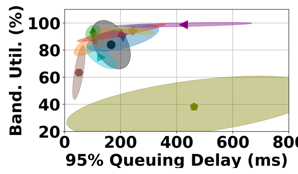
Inference Time Results. We estimate the inference time by measuring the execution time of querying the model. For the LSTM and decision tree, we measure the inference time as the time required to execute the query function. The query function is invoked with 20ms intervals. For RemyCC, we measure the execution time of an ACK-driven function which is used to search the tree to find operations on congestion window and sending intervals. We run them on a local



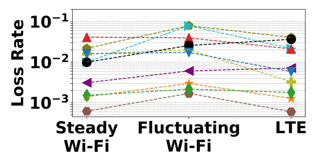
(a) Steady Wi-Fi



(b) Fluctuating Wi-Fi

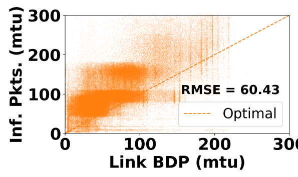


(c) Verizon LTE

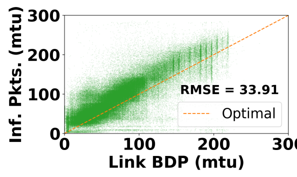


(d) Loss Rate

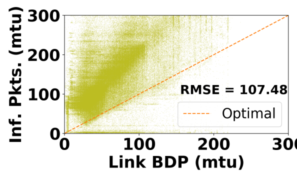
Fig. 3: Bandwidth Utilization vs Queuing Delay vs Loss Rate in Different environments



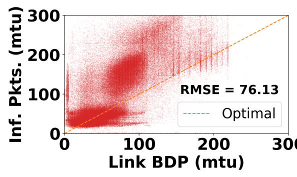
(a) TreeG



(b) *Muses*

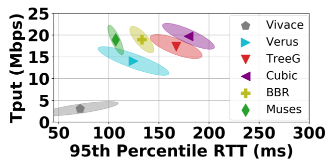


(c) BBR

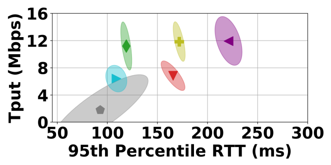


(d) RemyCC

Fig. 4: In-flight Packet Number vs Link BDP (Optimal)

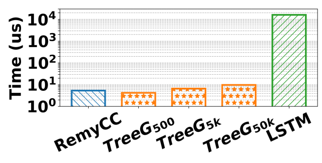


(a) Wi-Fi Access

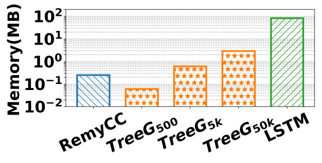


(b) LTE Access

Fig. 5: Throughput vs RTT in-the-Wild



(a) Inference Time



(b) Memory Usage

Fig. 6: Resource Overhead of LSTM and Decision Tree

loopback link for data transmission. For each algorithm, we run 20 tests and each test contains 1,000 queries.

Figure 6a shows the average inference time for each model. The average inference time for the LSTM is 16.3ms, which is much larger than all $TreeG_x$ models and RemyCC. The long inference times negatively impact congestion window scaling, which then reduces performance. Further, the long inference time blocks all other protocol logic in single thread mode like sending packets and timers. The decision tree models are much faster, *e.g.,* $TreeG_{500}$ takes 4.2us for inference, which is a 3 order of magnitude improvement compared to the LSTM. Even the most complex model, $TreeG_{50k}$, takes just 9.8us. These results speak to the benefits of building lightweight models for each environment.

Memory Usage Results. We also compute the runtime memory difference before and after the model file is loaded. Figure 6b presents the memory usage for each model. The LSTM requires 83.4MB of memory, which is heavier than all the decision tree models combined. This is particularly expensive in cases where applications rely on their own user-space congestion control (*e.g.,* QUIC). The $TreeG_{500}$ takes only 61KB, which is 3 orders of magnitude less than the LSTM. This is a significant improvement, especially in low cost mobile devices and IoT instruments

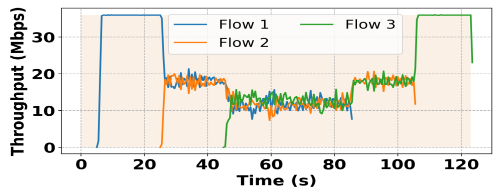


Fig. 7: Fair Convergence of *Muses*

TABLE IV: Jain's Fairness Index Comparison

Scenario	3 Flows	5 Flows	10 Flows	15 Flows	20 Flows
Muses	0.999	0.999	0.988	0.988	0.965
Cubic	0.994	0.989	0.980	0.982	0.960

D. Fairness

Setup. To test fairness, we initiate three flows in a emualted network. These run in a staggered fashion, with 20 second intervals, over a shared 36Mbps link. Each flow lasts for 80 seconds.

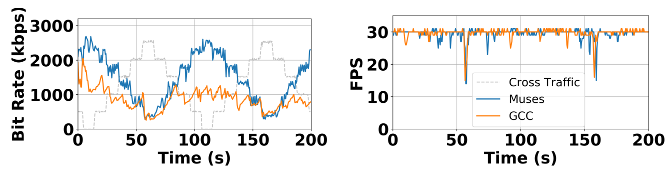
Results. Figure 7 presents the throughput of the three flows achieved by *Muses*. We see that, when Flow 2 starts, the throughput of Flow 1 decreases rapidly until the two reach an equilibrium. When Flow 3 starts, Flow 1 and 2 have the same reaction and reduce their sending rates. When Flow 1 completes, we then see that Flows 2 and 3 rapidly detect the new capacity and increase their sending rates to achieve the third equilibrium. This experiment demonstrates the reactivity of *Muses* when competing for traffic.

In order to verify that this holds when the number of flows is larger, we test different conditions of 3-20 flows on *Muses* and Cubic. The experiment setup is the same as previous. We use Jain's fairness index [41] as a fairness metric, which is in the range of 0 to 1 (The best case is 1).

Table IV presents the fairness scores. We see that *Muses* exhibit the same fairness as Cubic in multi-flows scenarios, confirming that *Muses* does not undermine the fairness requirements of traditional CC algorithms.

E. Use Case: WebRTC

To verify the effectiveness of *Muses* for real applications, we integrate it into the *WebRTC* framework (a widely used real time video chat application [21]).



(a) Bit Rate (b) FPS

Fig. 8: QoE in WebRTC: *Muses* vs GCC

Implementation in WebRTC. We implement a new WebRTC CC module. This incorporates the decision trees and other necessary information generated by previous two stage of *Muses*. In the WebRTC CC framework, a callback function will be scheduled when the sender sends the RTCP packets or receives the RTCP feedback packets. In the callback function, a hook function can be used to update the sending rate and congestion window for the sending engine and encoder. In *Muses*, the calculation sub-module of input state is implemented in both packet sent and packet received callback functions. The model inference and model selection sub-modules are implemented in the packet received function. Every 10ms the hook function is triggered when the packet received function is scheduled by the event of receiving RTCP feedback packets.

Setup. To test our implementation, we use a setup containing two Android mobile phones that are connected to Wi-Fi access points (both 802.11ac, 5GHz). The two access points are connected to an emulated network, implemented using OVS. We use Linux TC to configure the internal emulated network with 3Mbps bandwidth and 30ms one-way delay. Cross traffic is injected into the network using a stepped pattern ranging from 0kbps to 2500kbps. Each step increases or decreases the bandwidth by 500kbps for 10s. We run this experiment to highlight the practicality of *Muses* in low resource environments.

Results The results are shown in Figure 8 as a time series. We compare the *Muses* congestion control with the default congestion control algorithm used by WebRTC: GCC [22]. We present both the bitrate and frames per second (FPS) across time; note that Figure 8a also presents the cross traffic bitrate.

The two algorithms perform almost the same in terms of FPS, at around 30 frames. For bit rate, *Muses* substantially outperforms GCC and achieves 1.34× on average vs. GCC. When cross traffic suddenly drops, *Muses* quickly occupies the free bandwidth to support higher-quality frame transmission. This is driven by *Muses'* ability to more rapidly and accurately react to congestion signals. We also note that the average inference time of *Muses* is just 4us. This is comparable with GCC and reasonable on mobile devices.

V. Related Work

We roughly divide end-to-end congestion control algorithms into hard-coded and learning-based algorithms.

Hard-coded Algorithms. Perhaps the most well known algorithm, Reno [42], proposed the key components of CC. This includes the use of congestion signals, slow start, congestion avoidance, response to congestion signals, and recovery. Many TCP variants are aimed at specific networks and improve these parts. For example, Vegas [3] uses RTT as a signal of congestion to improve performance in networks with stable delay; Cubic [1] uses a cubic function to increase its window during the congestion avoidance phase; Verus [20] calculates a window-delay profile based on historical observations and relies on the signal of delta delay to select a target delay and window; BBR [7] estimates the maximum bottleneck bandwidth and propagation delay to control sending rate and inflight packets. Naturally, these algorithms rely on the accuracy of congestion signals or estimations, and the impact of noise in these measurements is significant. Similarly, when deployed in scenarios that violate their assumptions, the above algorithms tend to perform poorly [10], [11], [43].

Learning-Based Algorithm. To overcome these limitations, several learning-based algorithms have emerged. Vivace [17] and PCC [13] adjust the sending rate in real time, and determine the size of the change according to a performance utility function gradient. However, this estimation of utility relies on the stability and predictability of the underlying network, which does not always hold in dynamic wireless conditions. RemyCC [12] iteratively searches for a state-action mapping table to maximize an objective function. As discussed above, this mapping table is based on a tree structure that we have shown is too simple to support more complex wireless environments. Indigo [14] uses an LSTM model to train across a wide range of scenarios, guided by reaching the optimal operating point of the network. However, this has significant resource requirements, making is infeasible for the low cost mobile devices supported by *Muses*. Orca [16] combines a learning model and hard-coded algorithm to ensure recovery from the wrong equilibrium. Yet, for mobile devices, the invocation frequency in its deep learning agent is still too high to scale (about every 20ms).

VI. Conclusion

This paper has presented *Muses*, a multi-stage learning scheme for generating multiple lightweight congestion control models, suitable for different network sub-environments. *Muses* first trains an LSTM model in a global network environment, and then extracts decision trees from this comprehensive model. Each flow then dynamically selects the best decision tree for its own environment. This combines the expressiveness of heavyweight models with the low overheads of lightweight models, making it ideal for deployment on low cost mobile devices. We show through extensive experiments that *Muses* can attain superior performance to prior learning-based approaches. Vitally, we demonstrate the viability of *Muses* on real devices and applications (*e.g.,* WebRTC), substantially lowering the resource consumption of prior techniques.

VII. Acknowledgments

We thank Tong Li, Fanzhao Wang and other colleagues from Huawei Computer Network and Protocol Lab for the work and support over the years. Gaogang Xie and Heng Pan are supported by National Key R&D Program of China (Grant No. 2019YFB1802800), and by the National Natural Science Foundation of China (Grant No. 61725206 and 62002344).

References

[1] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[2] S. Floyd, A. Gurtov, and T. Henderson, "The newreno modification to tcp's fast recovery algorithm," 2004.

[3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," in *Proceedings of the conference on Communications architectures, protocols and applications*, 1994, pp. 24–35.

[4] D. Rossi, C. Testa, S. Valenti, and L. Muscariello, "Ledbat: the new bittorrent congestion control protocol," in *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, 2010, pp. 1–6.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 63–74, 2011.

[6] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 115–126, 2012.

[7] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," 2016.

[8] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," 2018.

[9] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound tcp approach for high-speed and long distance networks," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12.

[10] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, "Tcp performance issues over wireless links," *IEEE communications magazine*, vol. 39, no. 4, pp. 52–58, 2001.

[11] J. Wang, Y. Zheng, Y. Ni, C. Xu, F. Qian, W. Li, W. Jiang, Y. Cheng, Z. Cheng, Y. Li, and et al., "An active-passive measurement study of tcp performance over lte on high-speed rails," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019.

[12] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 123–134, 2013.

[13] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "{PCC}: Re-architecting congestion control for consistent high performance," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 395–408.

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

[15] W. Li, F. Zhou, K. R. Chowdhury, and W. M. Meleis, "Qtcp: Adaptive congestion control with reinforcement learning," *IEEE Transactions on Network Science and Engineering*, 2018.

[16] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: A pragmatic learning-based congestion control for the internet," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 632–647.

[17] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "{PCC} vivace: Online-learning congestion control," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 343–356.

[18] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "A community evaluation platform for academic research on congestion control," https://pantheon.stanford.edu/, 2018.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 509–522.

[21] A. Bergkvist, D. C. Burnett, C. Jennings, A. Narayanan, and B. Aboba, "Webrtc 1.0: Real-time communication between browsers," *Working draft, W3C*, vol. 91, 2012.

[22] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (webrtc)," in *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 2016, p. 13.

[23] T. S. Rappaport *et al.*, *Wireless communications: principles and practice*. prentice hall PTR New Jersey, 1996, vol. 2.

[24] D. A. Hayes and G. Armitage, "Revisiting tcp congestion control using delay gradients," in *International Conference on Research in Networking*. Springer, 2011, pp. 328–341.

[25] H.-S. Park, J.-Y. Lee, and B.-C. Kim, "Tcp performance issues in lte networks," in *ICTC 2011*. IEEE, 2011, pp. 493–496.

[26] E. Atxutegi, Å. Arvidsson, F. Liberal, K.-J. Grinnemo, and A. Brunstrom, "Tcp performance over current cellular access: A comprehensive analysis," *Autonomous Control for a Reliable Internet of Services*, pp. 371–400, 2018.

[27] A. Parichehreh, S. Alfredsson, and A. Brunstrom, "Measurement analysis of tcp congestion control algorithms in lte uplink," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2018, pp. 1–8.

[28] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-regret reductions for imitation learning and structured prediction," in *In AISTATS*. Citeseer, 2011.

[29] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[31] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for {HTTP}," in *2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15)*, 2015, pp. 417–429.

[32] Ravi., "Mahimahi," https://github.com/ravinet/mahimahi/, 2015.

[33] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-wifi: Emulating software-defined wireless networks," in *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, 2015, pp. 384–389.

[34] N. Cardwell, Y. Cheng, S. Yeganeh, and V. Jacobson, "Bbr congestion control draft-cardwell-iccrg-bbr-congestion-control-00. google," *Inc Std*, 2017.

[35] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu, "Interpreting deep learning-based networking systems," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 154–171.

[36] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[37] C. J. Stone, "Classification and regression trees," *Wadsworth International Group*, vol. 8, pp. 452–456, 1984.

[38] L. E. Raileanu and K. Stoffel, "Theoretical comparison between the gini index and information gain criteria," *Annals of Mathematics and Artificial Intelligence*, vol. 41, no. 1, pp. 77–93, 2004.

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

[40] S. Abbasloo, "Orca," https://github.com/Soheil-ab/Orca/, 2018.

[41] R. Jain, A. Durresi, and G. Babic, "Throughput fairness index: An explanation," in *ATM Forum contribution*, vol. 99, no. 45, 1999.

[42] V. Jacobson, "Modified tcp congestion avoidance algorithm," *Email to the end2end-interest mailing list*, 1990. [Online]. Available: ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail

[43] T. Li, K. Zheng, K. Xu, R. A. Jadhav, T. Xiong, K. Winstein, and K. Tan, "Tack: Improving wireless transport performance by taming acknowledgments," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 15–30.