POLITECNICO DI TORINO
Repository ISTITUZIONALE

Owl: Congestion Control with Partially Invisible Networks via Reinforcement Learning

Original
Owl: Congestion Control with Partially Invisible Networks via Reinforcement Learning / Sacco, A., Flocco, M., Esposito, F., Marchetto, G.. - ELETTRONICO. - (2021). (IEEE INFOCOM 2021 - IEEE Conference on Computer Communications Virtual Event 10-13 May 2021) [10.1109/INFOCOM42981.2021.9488851].

Availability:
This version is available at: 11583/2862067 since: 2021-09-15T15:49:41Z

Publisher:
IEEE

Published
DOI:10.1109/INFOCOM42981.2021.9488851

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

08 September 2026

Owl: Congestion Control with Partially Invisible Networks via Reinforcement Learning

Alessio Sacco† Matteo Flocco⋆ Flavio Esposito⋆ Guido Marchetto†
†Politecnico di Torino, Italy ⋆Saint Louis University, USA

***Abstract*—Years of research on transport protocols have not solved the tussle between in-network and end-to-end congestion control. This debate is due to the variance of conditions and assumptions in different network scenarios, e.g., cellular versus data center networks. Recently, the community has proposed a few transport protocols driven by machine learning, nonetheless limited to end-to-end approaches.**

In this paper, we present Owl, a transport protocol based on reinforcement learning, whose goal is to select the proper congestion window learning from end-to-end features and network signals, when available. We show that our solution converges to a fair resource allocation after the learning overhead. Our kernel implementation, deployed over emulated and large scale virtual network testbeds, outperforms all benchmark solutions based on end-to-end or in-network congestion control.

***Index Terms*—TCP, congestion control, reinforcement learning**

I. Introduction

A performing congestion control protocol is fundamental for proper network operation as it ensures telecommunication stability, fairness in computer network resource utilization, high throughput, and a low switch queuing delay. Although many solutions have been proposed in the last decade, Transport Control Protocol (TCP) still constitutes the overwhelming majority of current Internet and Long Term Evolution (LTE) communications, and the vast majority of congestion control mechanisms are implemented on TCP [1].

Despite the wide deployment of TCP, various studies have shown how it performs poorly in scenarios that require adaptability or that departs from the original network conditions on which it was designed in the '70s [2]–[6]. In particular, problems may occur in cellular and wireless networks, where TCP misinterprets the stochastic packet losses as congestion, hence leading to performance degradation [6]. This issue has motivated many authors to propose innovative congestion control approaches that follow a domain-specific design philosophy, in which the design is limited to a specific network scenario and it leverages its specific characteristics to boost the performance. Examples are in data centers [7], [8] and edge networks [5], [6].

The challenge of adequately updating the *congestion window (cwnd)* in resource-constrained networks, such as wireless networks and IoT, is exacerbated by inherent problems arising from their limited bandwidth, processing, and battery power, as well as from their dynamic conditions [9]–[11]. The deterministic nature of TCP is indeed more prone to cause *cwnd* synchronization problems and higher contention losses, due to node mobility that continuously modifies wireless multi-hop paths [12], [13]. Several TCP variations (*e.g.*, PCC [14] and Copa [15], to mention a few) have been recently proposed to overcome these shortcomings. Nevertheless, the fixed rule strategies used by these solutions are often inadequate to adapt to the rapidly changing environment.

To solve the problem of an adequate congestion window update strategy, we present *Owl*, a novel transport protocol based on *reinforcement learning (RL)*. Differently from other Machine Learning-based approaches for transport protocols, we conduct online training at the source and decide the next value of *cwnd* using also an in-network mechanism, when available. Many transport protocols have been designed, with reinforcement learning [16], [17] or without for a network-aware solution [7], [18], [19]. The most recent solutions using RL, however, do not exploit network intelligence fully.

An optimal *cwnd* update increases the throughput and fairness while reducing the number of packets lost and delay. Our transport protocol *Owl* is able to achieve these goals by learning from several end-to-end and in-network metrics. In particular, our contributions are summarized as follows. We designed and implemented as a kernel module *Owl*[1], a new congestion control protocol that leverages partial network knowledge to train a reinforcement learning model based on Deep Q-Learning [20], improving the network performance with respect to recent work [21]. The outcome of Owl model is the next congestion window value, a crucial and volatile parameter for any reliable telecommunication. We then evaluate our solution extensively: first, we compare Owl with other sixteen transport implementations. Some of these solutions were designed for wireless networks, such as Sprout [5] or the more recent ABC [18], while others [22]–[25] were chosen since they are widely deployed in several Linux distributions.

Our performance results (obtained using emulations with real available traces from Verizon and T-Mobile and a deployment over the GENI testbed [26]) show that Owl has consistently bandwidth and delay improvements across several scenarios. We also evaluate the parameters of our deep neural network used in our reinforcement learning and tested Owl's fairness performance, finding that our transport protocol behaves less aggressively than others.

[1]As owls that (are wise and) can see with poor light conditions, our protocol operates with partially visible networks.

Besides, we evaluate the impact of partial network visibility, and we demonstrate that our agent can efficiently operate with partial or even without in-network congestion signals. Lastly, we show that the sender can learn the optimal congestion window adjustment strategies in a variety of network deployments and can adequately react to network changes.

The remainder of the paper is outlined as follows. Section II presents the related work and most relevant mechanisms we compare with. In Section III we explain our problem formulation with reinforcement learning while Section IV describes our protocol design and shows the rate stability analysis. Section V summarizes our implementation, which is then evaluated in Section VI, where we show the benefits of our protocol. Finally, in Section VII we conclude the paper.

II. Related Work

Congestion control and avoidance problems have been widely discussed in the literature due to the great importance in reliable data transmissions. To solve the optimal congestion window inference problem, recent machine learning-based algorithms have been proposed with promising results in different network scenarios. In this section, we focus on highlighting how these solutions differ from our protocol.

Congestion Control is a fundamental service offered by TCP, so much so that significant improvements and variations have been proposed over the years. A few examples are TCP Vegas [24], Compound [27], Fast [28], BBR [25], and Data Center TCP (DCTCP) [7]. Rather than relying on indications of lost packets to adjust the *cwnd* as traditionally happens, BBR considers RTT and average delivery rate measurements to decide how fast to send data over the network. This enables BBR to be resilient to the bufferbloat problem, but it frequently exceeds the link capacity, causing excessive queuing delays [18]. Other protocols, *e.g.*, Compound [27] and Fast [28], instead attempt to optimize losses, but they rely on some predefined functions or rules to handle network conditions.

In summary, all these solutions share the limitation of a fixed-rule strategies, that is, their performance is challenged in networks that require rapid adaptations. Our solution, instead, uses a (reinforcement) learning approach to overcome this limitation and predicts the best *cwnd* update at each transmission event.

Recent end-to-end congestion control solutions, such as Remy [4], PCC [14], PCC-Vivace [29], define an objective function to optimize the process of online actions definition, *e.g.*, on every ACK or periodically. Remy [4], for example, offline trains every possible network condition to find the optimal mapping with the sender's behavior. These mappings are stored a-priori in a lookup table, and rely on what has been seen and hence can accommodate new network conditions only by recomputing the lookup table.

On the other hand, PCC [14] and PCC-Vivace [29] perform online optimizations. For instance, PCC can rapidly adapt to the varying conditions in the network by aggressively searching for more accurate actions to change the sending rate. However, these online rules are often complex and require considerable lags in estimating all the parameters to be accurate.

Based on a similar utility-based behavior idea, Copa [15] employs a delay-based congestion control algorithm, by adjusting the *cwnd* depending on whether the current rate is lower or higher than a well-defined target rate. This approach allows converging quickly to the correct fair rates, even in the face of significant flow churn. Our protocol also uses a utility-bases approach, but exploiting a deep neural network to better adapt to a specific network, leaving the utility customization as a policy that can be tailored to more specific requirements.

Learning for Congestion Control. As a recent trend, Machine Learning (ML) has been widely applied to various problems arising in network operation and management [30]. We use ML to adapt the *cwnd* estimation. Other approaches based on reinforcement learning have been proposed to address this problem. The majority of these approaches are specifically designed to cope with a resource-constrained network, including IoT [9] and WANETs [10], [12], [13], [31]; others instead address a wider range of network architectures [4], [14], [32]. Recently, RL has permeated many congestion control mechanisms, such as Aurora [21], where the previous Performance-oriented Congestion Control (PCC) protocol was extended with a Deep-RL approach. Our RL approach differs from others for the agent state: we effectively combine features from both the transport and the network layers, without significant burdens to the Linux kernel module.

In-Network versus End-to-End Congestion Control. Several protocols leverage the Explicit Congestion Notification (ECN) to provide network-level feedback to end hosts. For example, DCTCP [7] modifies the Red Early Drop thresholds of ECN to achieve high throughput, high burst tolerance, while keeping queues empty hence experiencing low latency. ABC [18] instead improves on ECN by sending accelerate and brake signals instead of merely random early drop signals, and hence more accurately adjusts the source sending rate. As ABC, Owl also uses network-level information as well (when available), however, our feedback comes from a network controller, *e.g.*, a measurement agent or an SDN controller, that computes statistics about device utilization. Also, Owl does not need any modifications to packets headers or custom routing devices logic, which leads to challenging deployments. In fact, Owl only relies upon client-side changes and a network statistics collector, a standard operation across multiple network scenarios. On the one hand, our network-level feedback carries more information than a simple bit in the TCP header. On the other hand, Owl functions properly also without network knowledge, while ABC and other ECN-based approaches require network knowledge to work.

III. Problem Formulation and Reinforcement Learning Model

The proposed congestion control algorithm behind Owl computes the next *cwnd* values by leveraging statistics gathered by the sender. In this section, we overview the

reinforcement learning model that we use and describe the overall idea of our approach.

A. Reinforcement Learning Model

In every reinforcement learning problem [33], an agent, *i.e.*, a decision-maker, tries to learn the behavior of a dynamic system interacting with it in multiple iterations. Specifically, at each iteration, an agent receives the current state and the reward from the dynamic system and outputs an action that optimizes a given objective.

Thus, state and reward are the values that the agent receives from the system, whereas the action is the only input that the system acquires from the agent. A reward value indicates the success of the agent's action decisions, and the agent learns which actions to be selected to provide the highest accumulated reward over time, *i.e.*, the long-term revenue. Hence, the critical feature for reinforcement learning is to perform incentive solution searching with regards to the system reward.

Q-Learning [34] estimates the value of executing an action from a given state. Such estimations are referred to as state-action values, or sometimes simply Q-values, $Q(s, a)$. This quality function represents the quality for taking action a at the current state s. Q-values are learned iteratively by updating the current Q-value towards the observed reward and estimated utility of the resulting state s' according to:

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right), \quad (1)$$

where $\alpha \in [0, 1)$ is the learning rate that determines the override extent of the newly acquired information to the old one, $\gamma \in [0, 1)$ is the discount factor that determines the importance of future rewards, and r is the reward at time t. In this case, the agent utilizes the highest quality function at state s' regarding all possible actions.

To handle the complexity of having to keep a separate state-action pair for too many states, models that approximate the Q-values are beneficial. To solve our congestion inference problem, we select a Deep Q-Learning approach [20], in which the model is a neural network parameterized by weights and biases collectively denoted as θ.

B. Congestion Control via Reinforcement Learning

We now overview our primary components in the RL method, starting with our considered state set, then with the set of actions on the congestion windows, and finally, with the utility that drives the choice of the next protocol action.

State Space. Table I summarizes the features that we selected to build our model state space. We consider both end-to-end statistics (features 1 to 8) and network-level statistics (features 10 and 11). Thus, the former set of features is collected at the sender side at each time interval, any *jiffy*, where jiffy is the finest time granularity on Linux systems. Instead, the last two features represent the partial information coming from the network (feature 10), and a parameter stating the quantity of knowledge, as a percentage of the whole network (feature 11), respectively. For each switch under control, let P_{in} be the total number of packets received in a given time interval (one second in our implementation), and P_{out} the total number of outgoing packets. We then define `diff` as $|P_{in} - P_{out}|$. Our *Partial Network Knowledge* (PNK) represents an indicator of the known level of congestion within the network. In particular, given a source receiving statistics or updates from z switches on the path between a source and a destination, PNK is computed using the following equation:

$$PNK = \max(diff_1, diff_2, ..., diff_z). \quad (2)$$

PNK informs about the current congestion level, and consequently, the loss rate occurring in the network. We choose PNK as it is easy to compute and accessible by a vast number of protocols and network measurement applications, such as OpenFlow or NetFlow. Further information regarding the network environment whereby our protocol performs best is in Section IV-A. Nonetheless, we remark that Owl can also be configured as an end-to-end protocol, in case the network knowledge is hidden or impractical to obtain.

In defining our states, we also consider a history window of k values for each chosen feature as our state. This approach helps our algorithm to predict the network conditions adequately and to adjust the congestion window accordingly. The neural network of our deep reinforcement learning algorithm receives a matrix N by k, where k are the historical values for each of the N features. In our experiments, k has been set to 5 (more details in Section VI-D). We augment our state space with a history of generic length k to help the agent's learning. However, we do not set this hyperparameter to a large value since that prevents the state from growing unreasonably, and because forgetting history faster is beneficial.

Actions. The congestion window (*cwnd*) is one of the per-connection state variables that is used by TCP to limit the amount of data a sender can transmit before receiving an ACK. TCP was designed based on specific network conditions and handles all packet losses as network congestion. Therefore, TCP in wireless lossy links unnecessarily lowers its rate by reducing the *cwnd* at each packet loss, negatively affecting the end-to-end performance. Hence, we exploit an online training algorithm based on RL to update the *cwnd* properly.

TABLE I: The network statistics gathered for estimating the upcoming performance.

	Features of the Owl congestion window predictor
1	Time-stamp [jiffies]
2	Congestion Window Size (*cwnd*) [packets]
3	Round Trip Time (RTT) [ms]
4	RTT variation between two consecutive samples [ms]
5	Maximum Segment Size (MSS) [bytes]
6	Number of delivered packets
7	Packets lost during a transport session
8	Current packets in-flight
9	Number of retransmissions [packets]
10	Partial Network Knowledge (PNK) [packets]
11	Percentage of known network [%]

The selection of actions is the key to the proposed algorithm's effectiveness. The list of actions specifies how Owl should change the *cwnd* in response to every packet acknowledge. The set of acceptable congestion window values is large and tied to the reward of the RL system. Hence, there is no unique solution across every network condition. After an empirical evaluation, we converged on the set that has given us the highest utility, that is:

$$A = \{-10, -3, -1, 0, 1, 3, 10\}. \tag{3}$$

We allow the agent to change the *cwnd* in any direction with different intensities. The first three options reduce the size of the congestion window with a distinct extent, whereas the last three increase it by three different values. The last action does nothing to the size of the *cwnd*, letting it remains the same as before. We want to encourage the agent to explore diverse ways to influence the connection by assigning different magnitudes to the performed change. Indeed, not only the learning agent should predict when increasing or decreasing the *cwnd*, but also to what extent. For example, our algorithm must learn when the network state suggests that a large part of the bandwidth is unused to aggressively increment the window size, while it must only slightly increase it when the network approaches any congestion. Our network module starts with an initial *cwnd* of 10.

Due to the opted approach, the protocol learns how to make control decisions from experience and, thus, eliminates the need for necessary pre-coded rules to adapt to the variety of network environments. Finally, we converged to the action set in Eq. 3 after having performed a substantial number of empirical trials. Nevertheless, the action set A is a policy that can be tailored to specific use cases, by either modifying values of the congestion window size (as we did) or acting upon other TCP parameters, *e.g.*, timeout estimation or slow-start threshold.

Utility function (RL reward). The selection of the congestion control schema relies on a utility function that models the application-level goal of "high throughput and few losses". In particular, the utility U of sender i is a function of throughput λ and packet loss rate p, as follows:

$$U_i = \lambda_i - \delta_i \lambda_i \left(\frac{1}{1 - p_i} \right), \tag{4}$$

where $p \in [0, 1)$ and δ is an adjustable coefficient determining the importance of the components. For example, a larger δ implies that lower packet delays are preferable. The goal of each sender i is to maximize its utility function U_i. In what follows, we better motivate the reasons behind such an expression.

IV. Protocol Design and Stability Analysis

In this section, we present the mechanisms composing our protocol, whose design aims to continuously select the next action, *i.e.*, congestion window size, that maximizes the value of our utility function. Our protocol evaluates the reinforcement learning action based on the reward perceived

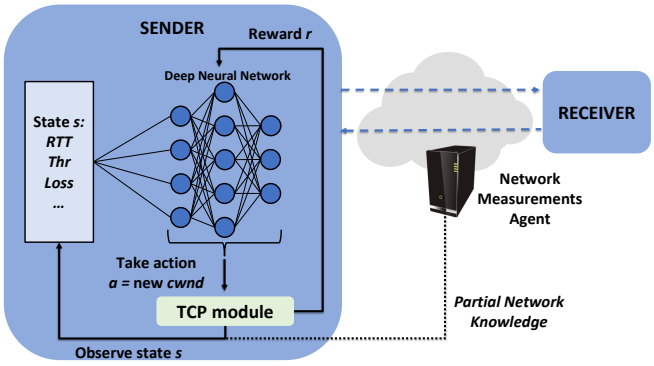


Fig. 1: Owl Overview: reinforcement learning sender's agent interaction with the network.

by the sender, used to select the next *cwnd* adjustment. We also describe the overall procedure with particular focus on the utility's motivation, leading to stability.

A. Owl Protocol Design

In Figure 1, we detail the main actions performed by the sender. The collected metrics are fed to the Neural Network, and the protocol starts (Algorithm 1).

Algorithm 1 `Owl cwnd update`

1: Let S and D be the target source and destination
2: $F \leftarrow$ flow connecting S and D
3: Collect state vector `s` at time `t` for flow F
4: `cwnd*(t)` $\leftarrow$ `cwnd_Prediction(s, t)`
5: Set `cwnd` to `cwnd*(t)`

Specifically, we collect the state of the end-to-end communication, *e.g.*, RTT and throughput, exploiting the TCP Linux API. Concerning the network feedback, the network measurement agent computes PNK by controlling the underneath topology, and notifies it to the sender. We argue it is not always possible to obtain the entire path between the source and the destination. However, even when the network feedback is incomplete or unavailable (the neural network does not use the in-network features), our protocol still provides valuable results (Section VI).

Once Owl has collected such values, it selects the next *cwnd* by choosing the "action" according to the Q-table. The algorithm to predict the next *cwnd* value is detailed in Algorithm 2. In particular, the algorithm avails the states,

Algorithm 2 `cwnd_Prediction(s: state, t: time)`

1: At time $t = 0$ initialize $Q(s, a) = 0$ and set reward r as in Eq. (4)
2: At time t :
3: Observe r as a consequence of the last action
4: Update $Q(s, a)$ function according to Eq. (1)
5: `cwnd*(t)` $\leftarrow$ `softmax(a,s,t)`
6: **return** `cwnd*(t)`

actions, and reward to select the best value and update the

Q-table. The prediction of the best *cwnd* occurs every time a packet is acknowledged to guarantee an adequate refresh of the *cwnd* used in the congestion avoidance phase. The state set is then updated to assure k historical values for each metric at any interval.

B. Stability Analysis

In this subsection, we show that processes running our protocol converge to a stable rate assignment. In particular, no sender has the incentive to deviate its sending rate from the strategy defined by our Owl protocol objective function, hence reaching a Nash equilibrium. At the equilibrium condition, we have the n-tuple of sending rates defined as $(\lambda_1, ..., \lambda_n)$. Formally we have that:

$$U_i(\lambda_1, ..., \lambda_i, ..., \lambda_n) > U_i(\lambda_1, ..., x, ..., \lambda_n), \tag{5}$$

for any sender i and any non-negative sending rate x, and so the following theorem holds.

Theorem IV.1. *(Stability). Consider n senders sharing a bottleneck link, and λ_i to be the rate of sender i; if for every sender i the objective function is defined by Equation 4, the sending rates converge to a stable equilibrium. Moreover for every sender i, we have:*

$$\lambda_i = \frac{C\left(\frac{n}{\delta_i} - \hat{z}\right)}{n+1}, \tag{6}$$

where $\hat{z} = \sum_{j \neq i} \frac{1}{\delta_j}$.

Proof. We need to show the existence of a Nash equilibrium, *i.e.*, no sender can increase its objective function value by unilaterally changing its rate. We consider a network model with n competing senders sharing a bottleneck link of capacity C and a FIFO-queue. Assuming a tail drop queue eviction policy, the loss rate function can be described as:

$$p_i = \begin{cases} 1 - \frac{C}{\sum_i \lambda_i} & \text{if } \sum_i \lambda_i > C \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Let us denote the arrival rate in the queue by $S = \sum_i \lambda_i$. Since the term $1 - \frac{C}{S} = \frac{S-C}{S}$ is independent of i and it is equal for all senders, all senders should experience the same loss rate, we denote p_i simply by p. By substituting these new terms into Equation 4, we obtain:

$$U_i = \lambda_i - \delta_i \lambda_i \frac{S}{C}.$$

First we compute the partial derivative, $\frac{\partial U_i}{\partial \lambda_i}$, and we split S into the two addends $S = \lambda_i + \sum_{j \neq i} \lambda_j$. Thus, for each i yields:

$$\frac{\partial U_i}{\partial \lambda_i} = 1 - 2\frac{\delta_i}{C}\lambda_i - \frac{\delta_i}{C}\sum_{j \neq i} \lambda_j.$$

We then compute the second derivative of U_i, with respect to the rate, and we obtain the negative quantity $-\frac{2\delta_i}{C}$. Hence, the utility is concave and the Nash equilibrium is achieved if, and only if, $\frac{\partial U_i}{\partial \lambda_i} = 0$. Next, to find the rate at which the equilibrium condition is achieved, we introduce $\hat{z}$ defined as $\hat{z} = \sum_{j \neq i} \frac{1}{\delta_j}$. Hence we have:

$$1 - 2\frac{\delta_i}{C}\lambda_i - \frac{\delta_i}{C}\sum_{j \neq i} \lambda_j = 0$$

$$2\lambda_i + \sum_{j \neq i} \lambda_j = \frac{C}{\delta_i}$$

The solution to the stated system of linear equations is:

$$\lambda_i = \frac{C\left(\frac{n}{\delta_i} - \hat{z}\right)}{n+1},$$

which is the desired sending rate of sender i.

□

V. Owl Prototype Implementation

Network Scenario. In designing our protocol, we considered practical scenarios in which networks are *partially unknown*. Wide-area networks may require (undesirable) cooperation and coordination of multiple (federated) gateways, and unstable network conditions may hide information. Part of our evaluation in Section VI focuses on the performance analysis of our protocol with such partial network knowledge, showing that the in-network information may add value if available, but it is not required as in other in-network congestion control mechanisms.

To analyze and respect this partial unavailability constraint, we designed and implemented a system in which a software-defined network (SDN) controller acts as a measurement collector and manages only some of the deployed (virtual) switches. While we use an SDN controller in our implementation, our approach is not limited to this specific technology. The controller interacts periodically with the switches to collect statistics about the number of packets transmitted and received. Such statistics are then used by our implementation to learn and predict the end-to-end action to take given the level of congestion. In our implementation, the controller receives packets' statistics from all switches with a (reconfigurable) sampling rate of one-second, a good trade-off between overload and freshness of information. The controller also runs a simplistic web server and exposes REST API to obtain these values, which are part of the input of our RL algorithm.

Kernel Module. The Owl module is responsible for setting the optimal congestion window. To operate, it obtains network states by communicating with a measurement agent, for example, an SDN controller. Figure 2 shows the main architecture components of our implementation. Our prototype is composed of two main processes: one running in the kernel and one in user-space. The kernel module exploits functions included in the classical *tcp_cong.c* to have access to the underlying congestion control functionalities of TCP. Like any other module, our kernel implementation can be mounted as a pluggable congestion control algorithm. It can set and get end-to-end transport states such as Sequence Number, ACKed Packets, RTT, and efficiently compute the throughput.

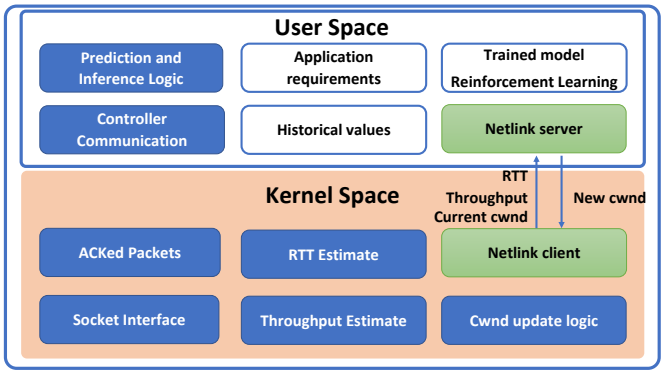


Fig. 2: Owl has a component that runs in the Linux kernel, and a component that runs at user-space to collect statistics to be used by our reinforcement learning algorithm.

The application process running in user-space collects information about the current TCP socket and uses them to build the input matrix of a Deep Neural Network running the reinforcement learning algorithm. The module takes actions in line with the RL feedback and modifies the *cwnd* as a reaction to events (Section III).

Storing the required states to run a reinforcement learning algorithm and to keep communications with the network controller can be costly at the kernel level. On the other hand, a user-space application can leverage a more extensive set of libraries to fit the learning algorithm's needs. Besides, the transmission of packets to/from the network controller could arise issues and requires proper management without having to switch from user-space to kernel-space. For this reason, we implemented the network management components of our congestion control algorithm at the user-space and marshall current TCP socket states between user-space and kernel via the *Netlink* service, commonly used for this purpose.

The reinforcement learning-based congestion controller accumulates network statistics from ACKs over a fixed period and sends the action asynchronously in a separate thread. The speed in retrieving data from the kernel is indeed higher than the rapidity of the reinforcement learning processing.

VI. Protocol Evaluation

To evaluate our proposal, we tested Owl against sixteen other transport protocols. In this section, we describe such an evaluation scenario and our application testbed deployment, followed by our performance results.

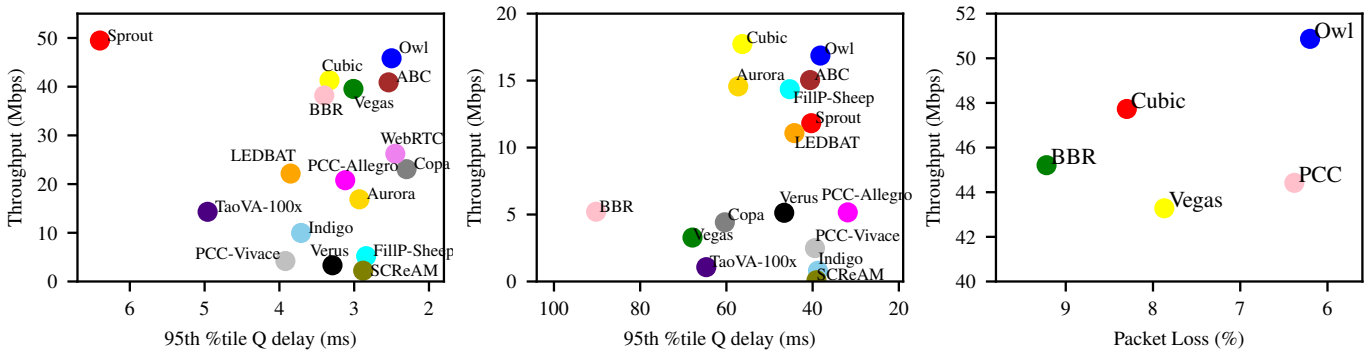
Evaluation Settings. To assess our congestion algorithms in comparison with other solutions over LTE networks, we built the services using a virtual network testbed and the Mahimahi emulator [35], a recent cellular link emulator that allows testing with real cellular traces from two of the largest US telecommunication providers, Verizon and T-Mobile. The network is emulated through namespaces, via Mininet [36]. The transmission goes through a Software-Defined Network (SDN), where switches interact with a centralized controller (in our implementation, we used Ryu). We also evaluate the performance over real hosts, and we deployed Owl over the GENI testbed [26]. Throughout our experimental campaign, we use the utility function described in Eq. 4, where δ has a value of 0.7. If not otherwise specified, we set a default percentage of known paths to be 80%. To evaluate each protocol, we average 35 experiments in which each sender-receiver pair runs *TCP iperf3* for 100 seconds.

A. Trace-Driven Emulation Results

To understand how Owl performs compared to other solutions, we deployed our protocol over an emulated network created with Pantheon [37], a well-known fairly recent testbed developed to evaluate congestion control schemes. In particular, we compared Owl against sixteen other protocols, divided into five categories: (i) end-to-end TCP designs: Cubic [22], Vegas [24], BBR [25], Copa [15], PCC [14] and its variants; (ii) end-to-end cellular, *i.e.*, LTE protocols: Verus [6], Sprout [5]; (iii) Machine Learning-based transport protocols: Indigo [37] and Aurora [21]; (iv) explicit congestion control: ABC [18] and (v) mixed schemes: LEDBAT [38], SCReAM [39], WebRTC [40], Tao-VA [41]. For our LTE evaluation settings, we use the publicly available [35] Verizon and T-Mobile traces, with separate packet delivery for uplink and downlink. The traces were captured directly on those networks. These traces are also loaded on our local SDN-based virtual network testbed. Our OpenFlow controller is only aware of the virtual switches (instances of Open Virtual Switch (OVS) [42]) that are connected to the SDN controller. For in-network algorithms, such as ABC, we emulate compliant routers as Mininet hosts that marks the packets according to the algorithm's logic.

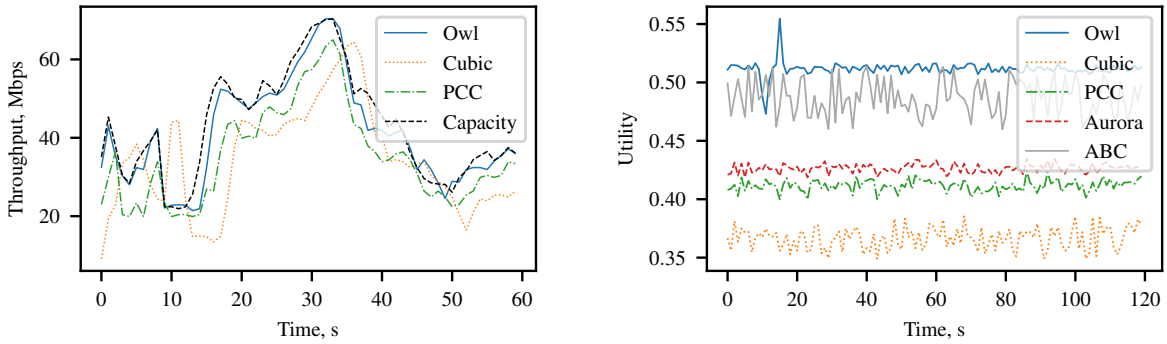
Figures 3a-b shows that Owl performs efficiently in all tested scenarios. In the case of Verizon LTE traces (Figure 3a) Owl achieves both good throughput and 95th percentile per-packet-delay, and no other solution has shown a better combined throughput-delay performance. Even though the RL reward was designed to achieve high throughput and low loss rate, we can observe that our mechanism can simultaneously obtain a low RTT, as a consequence of the imposed utility. Similar conclusions hold even for T-Mobile traffic (Figure 3b), where Owl provides a desirable trade-off between throughput and delay. It is worth noticing that none of the other algorithms outperform Owl in both tested environments: our solution appears to be more stable across traces.

Figure 4 shows the shortcomings of transport protocols in use and the lack of adaptation required for a good transport protocol. The Figure 4a represents a sample of the throughput evolution over the Verizon LTE downlink traces for 60 seconds. For the sake of clarity, we report only our comparison to Cubic, as it is the default in many Linux implementations, and PCC, as it one of the best performing within utility-based approaches. Owl adapts its sending rate so as to closely match the bottleneck link's available bandwidth (dashed black line in the figure). In contrast, Cubic slowly reacts to changes in the network, and PCC partially approximates the link capacity. *Our protocol can cope with rate variations in a reactive*



(a) Verizon LTE (b) T- Mobile LTE (c) GENI Testbed.

Fig. 3: **(a)-(b) LTE Trace-driven emulation.** Owl vs. previous schemes (using RL or not) tested over two cellular network traces (top-right are better). In both cases, Owl outperforms our benchmark, and has the highest fairness, on average, in both our tested use cases (Section VI-A). **(c) GENI testbed evaluation.** Throughput-loss rate trade-off for kernel-level solutions over real networks. Owl optimizes the two quantities simultaneously (Section VI-B).



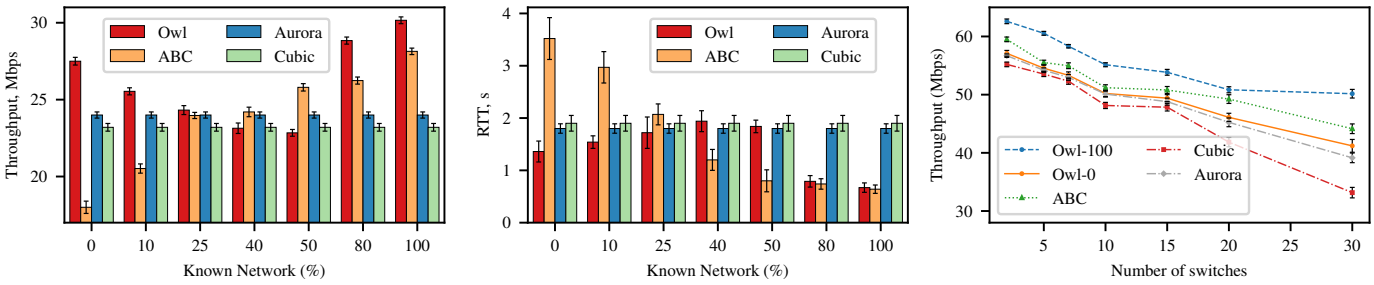
(a) Verizon LTE (b) AT&T LTE

Fig. 4: **Our protocol best follows the available bandwidth.** (a) A 60-seconds throughput's evolution compared to the actual link capacity. Owl fits best the Verizon LTE trace; while, especially for Cubic, overshoots in throughput lead to large standing queues. The curves shown have been selected for visual clarity. (b) A 120-seconds utility's evolution. Owl guarantees an adaptive response to the network dynamic changes.

manner and closely approximates the desired behavior by learning the optimal action.

This result is also confirmed in Figure 4b where we plot the utility (Eq. 4) obtained with different algorithms over AT&T LTE downlink. This time, we compare against ABC [18] as it is the most representative of explicit congestion control and Aurora as a novel RL-based congestion control algorithm. Likewise, we can observe how Owl regularly provides a higher utility than the benchmarks over time. This is due to the ability of the framework to learn the optimal behavior during training and then react efficiently during network dynamics. We can also observe how Aurora and Cubic fail to promptly react to the events.

Next, we discuss our experiments regarding the impact of the required network state knowledge that Owl needs to train the RL system effectively. Figures 5 display the (a) throughput and the (b) RTT, when different transport protocols run over a network composed of 20 nodes emulated on our local Mininet virtual network testbed. Specifically, we compare against Cubic as a reference end-to-end congestion control, Aurora, as a reference RL-based congestion control, and ABC, as a reference in-network control. The performance of Cubic and Aurora are not affected by the lack of in-network knowledge since they are both end-to-end congestion control algorithms. On the other hand, ABC performs worse than Owl when the number of ABC-compliant routers is relatively low. Our results validate that the value of PNK is beneficial to the algorithm, but our protocol works even as a pure end-to-end strategy. Our measurements reveal that even when less than 50% of the switches are utilized to collect statistics, our solution outperforms both end-to-end approaches (like Cubic) and novel in-network protocols (like ABC). On the other hand, if a partial network knowledge (more than 50%) is available, Owl drastically speeds up the transmission in terms of throughput and reduces latency. The worst result occurs approximately when half of the devices are controlled, as the agent cannot assign the proper importance to the coming information, resulting in occasionally misleading values. Nonetheless, even though in this scenario the information does not help improve the overall performance, Owl has results that



(a) (b) (c)

Fig. 5: **Network Knowledge Impact on Performance.** (a) Throughput and (b) RTT of Owl protocol for increasing percentage of known network. Somehow surprisingly, the highest performance gaps with respect to other algorithms are obtained when the percentage of network knowledge is either low or very high. (c) Throughput performance with or without network knowledge averaged over different network topologies and increasing number of informing switches.

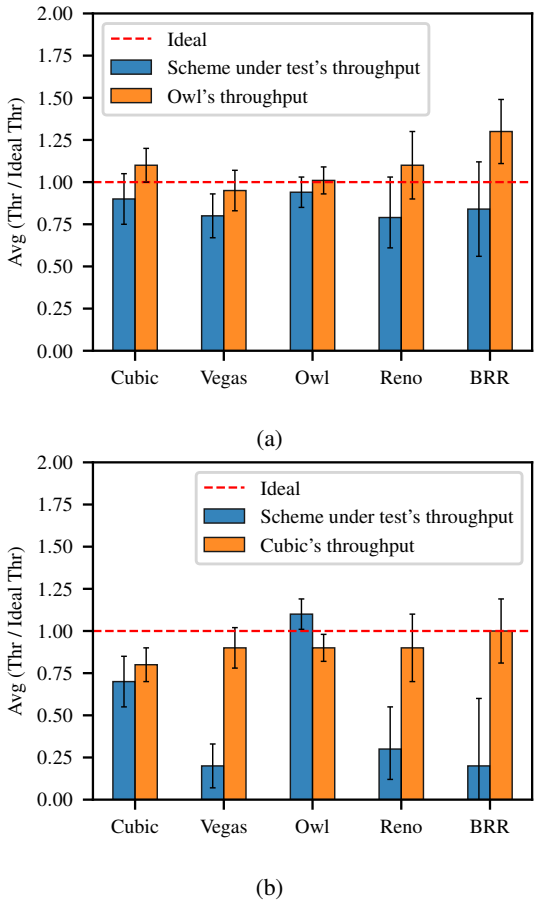
are comparable to other protocols.

Throughput Performance with respect to network size. In this experiment we compare Owl against a few representative protocols as we increase the number of informing switches over randomly generated topologies, *i.e.*, links are randomly generated while we fix the network size. The link capacity is also uniformly distributed at random between 50 and 100 Mbps. We are interested in assessing the impact of the network size on our congestion control algorithm. To this aim, we compare the perceived throughput when our solution has no in-network congestion feedback, and when the network is as informative as it can be, *i.e.*, the in-network feedback arrives from 100% of the switches. In Figure 5c, these two Owl policies are denoted with Owl-0, namely, zero-percent of total switches are communicating with the source, and Owl-100, respectively. It is notable how a full network awareness is beneficial and allows a less prominent (and inevitable) performance degradation when an increasing number of switches compose an end-to-end path. However, we note how even Owl-0 provides better results than recent end-to-end congestion control solutions based on RL [21].

B. Evaluation over the GENI Testbed

To establish the practicality of our approach and understand how Owl performs over wide-area Internet paths with real cross-traffic and real packet schedulers, we deploy our solution on the GENI testbed. In these experiments, we evaluate how congestion control schemes behave across two federated GENI aggregates. We measure the performance of each schema when competing with another flow to accentuate the possible congestion occurrences. To evaluate our protocol in these realistic settings, we average the throughput and delay over 60-second flows, while the senders share a bottleneck link with 3ms RTT and a bandwidth of 100 Mbps.

We summarize in Figure 3c the performance of our protocol when compared to other protocols available on Linux. Our prototype evaluation deployed in real settings match our emulation results: our implementation can jointly achieve high throughput and a low loss rate when compared to other solutions, balancing the two components effectively.



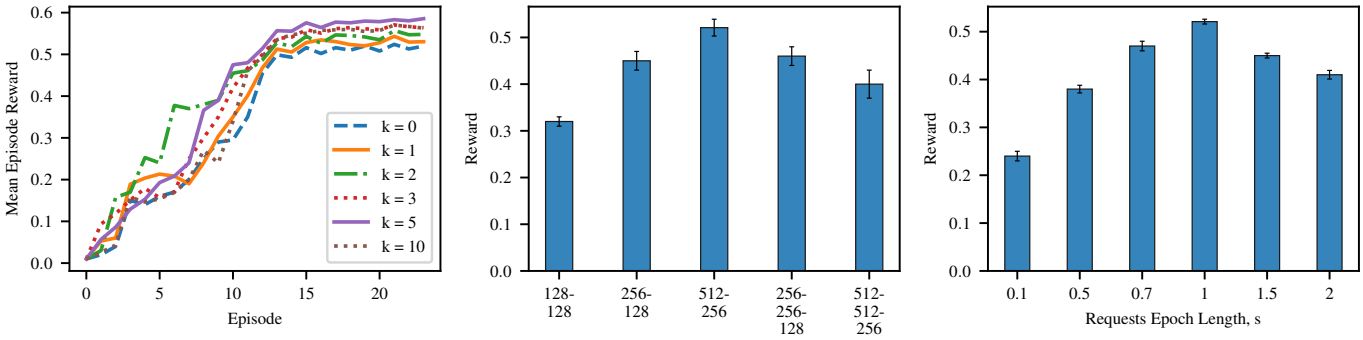
(a)

(b)

Fig. 6: **Fairness Analysis.** (a) Different schemes utilization and how they share the available bandwidth. The ideal fairness value is 1. Owl is fair, especially when used in conjunction with other Owl users. (b) We then compared Cubic's fairness to assess improvement over existing solutions.

C. Owl Fairness and Friendliness

In this subsection we evaluate the fairness among several flows all running Owl and competing with each other; we



(a) History length, k (b) Neurons per layer (c) Requests Interval Period

Fig. 7: Sensitivity analysis used to justify the choice of our default algorithm parameters; (a) k historical values of feature are used to make the next cwnd prediction; (b) Neurons per layer for Owl's neural network configuration; (c) Time interval for requests of network devices metrics.

also assess Owl's friendliness, *i.e.*, fairness when a Owl flow compete against different protocols, such as Cubic.

We set up an experiment where the network has a bottleneck link of 30 ms RTT and a bandwidth of 50 Mbps. First, we evaluate the friendliness against all congestion control solutions that are installed on Linux by default (Figure 6a). We compare the average ratios between throughput values achieved by each flow with respect to their ideal fair share. We found that Owl has a higher level of friendliness when multiple flows run Owl and when Owl competes with other transport protocols (Figure 6a.) While perfect friendliness does not hold for any of the tested schemes, we note how Cubic (that has best throughput-delay performance among its Linux counterparts), has a worse level of friendliness than Owl (Figure 6b). Second, from the same graphs, we can also derive the level of fairness of our solution compared to Cubic's one. It is easy to conclude that Owl results in a higher level of fairness.

D. Sensitivity Analysis

In this section, we report our experimental results conducted to establish the best parameter set in our congestion control algorithms and discuss their sensitivity. In particular, we focus on the Neural Network (NN)'s shape, the parameter k of the algorithm (for how long do we need to remember history for a more accurate *cwnd* prediction), and the frequency at which we should collect in-network measurements. We evaluate how these values affect performance over 30 trials on the GENI testbed. First, we examine the impact of the length of the action history in the augmented state space. Figure 7a shows the mean reward per episode obtained at training for varying values of the state history length k, when each episode entails 500 steps. We can observe that models with $k = 0$ or 1 struggle to learn, while the best performance is attained with $k = 5$ with diminishing returns beyond that value of k.

Further, we also run the same experiment with various Neural Networks to analyze how this choice may affect performance. Figure 7b exhibits the reward measured during the RL testing phase for the following Neural Networks configurations: *(a)* two layers comprised of 128 neurons each, *(b)* two layers with 256 and 128 neurons respectively, *(c)* 512 and 256 neurons, *(d)* three layers with 256, 256, and 128 neurons, *(e)* 512, 512, and 256 neurons. These results suggest that a two-layer neural network architecture works well, and that the combination 512-256 (*(c)*) provides the best reward. Hence, we empirically set this configuration as the default of our system, but we realize that this configuration is a policy.

Finally, we investigate the selection of the most valuable measurement request interval (Figure 7c). We note that, when the network measurements are gathered every 1-second, the reward is at its maximum. This value also guarantees the freshness of data without incurring in too frequent updates. We leave the analysis of alternatives approaches to further reduce training time as an interesting open question. We also plan to investigate more deeply the tolerated latency between the sender and the network measurement agent that guarantees an optimal congestion control.

VII. Conclusion

In this paper, we presented Owl, a reinforced learning-based transport protocol designed to learn from end-to-end and in-network signals. Our evaluation, with a kernel implementation and real traces, confirms that Owl is effective under various network conditions, and it can speed up transmissions and reduce delays and loss rate better than most existing protocols in the vast majority of the tested scenarios.

We also analyzed the stability condition of Owl and evaluated its fairness demonstrating that it is less aggressive than other performant solutions when it competes with other protocols and when it competes with itself across other sources. Finally, we showed how taking into account information involving the network layer leads to increasingly better results, especially when at least 50% of the network congestion state is available at the source.

Acknowledgement

This work has been partially supported by Comcast and by NSF Awards CNS-1836906 and CNS-1908574.

REFERENCES

[1] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing tcp's retransmission timer," RFc 2988, November, Tech. Rep., 2000.

[2] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck, "An in-depth study of lte: effect of network protocol and application behavior on performance," in *ACM SIGCOMM Computer Communication Review*, vol. 43. ACM, 2013, pp. 363–374.

[3] H. Jiang, Y. Wang, K. Lee, and I. Rhee, "Tackling bufferbloat in 3g/4g networks," in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012, pp. 329–342.

[4] 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.

[5] K. Winstein, A. Sivaraman, and H. Balakrishnan, "Stochastic forecasts achieve high throughput and low delay over cellular networks," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 459–471.

[6] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45. ACM, 2015, pp. 509–522.

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, 2010, p. 63–74.

[8] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41. ACM, 2011, pp. 50–61.

[9] W. Li, F. Zhou, W. Meleis, and K. Chowdhury, "Learning-based and data-driven tcp design for memory-constrained iot," in *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE, 2016, pp. 199–205.

[10] B. V. Ramana, B. Manoj, and C. S. R. Murthy, "Learning-tcp: A novel learning automata based reliable transport protocol for ad hoc wireless networks," in *2nd International Conference on Broadband Networks, 2005.* IEEE, 2005, pp. 484–493.

[11] A. Sacco, F. Esposito, and G. Marchetto, "Rope: An architecture for adaptive data-driven routing prediction at the edge," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 986–999, 2020.

[12] B. V. Ramana and C. S. R. Murthy, "Learning-tcp: a novel learning automata based congestion window updating mechanism for ad hoc wireless networks," in *International Conference on High-Performance Computing*. Springer, 2005, pp. 454–464.

[13] V. Badarla and C. S. R. Murthy, "Learning-tcp: A stochastic approach for efficient update in tcp congestion window in ad hoc wireless networks," *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, pp. 863–878, 2011.

[14] 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)*, Oakland, CA, 2015, pp. 395–408.

[15] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 329–342.

[16] Y. Kong, H. Zang, and X. Ma, "Improving tcp congestion control with machine intelligence," in *Proceedings of the 2018 Workshop on Network Meets AI & ML*, 2018, pp. 60–66.

[17] W. Li, F. Zhou, K. R. Chowdhury, and W. Meleis, "Qtcp: Adaptive congestion control with reinforcement learning," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 445–458, 2018.

[18] P. Goyal, A. Agarwal, R. Netravali, M. Alizadeh, and H. Balakrishnan, "ABC: A simple explicit congestion controller for wireless networks," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 353–372.

[19] S. Floyd, "Tcp and explicit congestion notification," *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 5, pp. 8–23, 1994.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[21] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *International Conference on Machine Learning*, 2019, pp. 3050–3059.

[22] 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.

[23] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for tcp," *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4, pp. 270–280, 1996.

[24] L. S. Brakmo and L. L. Peterson, "Tcp vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[25] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[26] Geni, Exploring Networks of the Future. [Online]. Available: https://www.geni.net/

[27] 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.

[28] C. Jin, D. X. Wei, and S. H. Low, "Fast tcp: motivation, architecture, algorithms, performance," in *IEEE INFOCOM 2004*, vol. 4. IEEE, 2004, pp. 2490–2501.

[29] 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.

[30] R. Boutaba, M. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. Caicedo Rendon, "A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities," *Journal of Internet Services and Applications*, vol. 9, 05 2018.

[31] H. Jiang, Y. Luo, Q. Zhang, M. Yin, and C. Wu, "Tcp-gvegas with prediction and adaptation in multi-hop ad hoc networks," *Wireless Networks*, vol. 23, no. 5, pp. 1535–1548, 2017.

[32] V. Badarla and C. Siva Ram Murthy, "A novel learning based solution for efficient data transport in heterogeneous wireless networks," *Wireless Networks*, vol. 16, no. 6, pp. 1777–1798, 2010.

[33] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 135.

[34] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[35] 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.

[36] R. L. S. De Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. R. Prete, "Using mininet for emulation and prototyping software-defined networks," in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, 2014, pp. 1–6.

[37] 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.

[38] S. Shalunov, G. Hazel, J. Iyengar, M. Kuehlewind *et al.*, "Low extra delay background transport (ledbat)," in *RFC 6817*, 2012.

[39] I. Johansson, "Self-clocked rate adaptation for conversational video in lte," in *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, 2014, pp. 51–56.

[40] 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.

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

[42] N. Networks, "Open vswitch: An open virtual switch," 2020. [Online]. Available: http://www.openvswitch.org/