Partially Oblivious Congestion Control for the Internet via Reinforcement Learning

Partially Oblivious Congestion Control for the Internet via Reinforcement Learning

Alessio Sacco, *Student Member, IEEE,* Matteo Flocco, Flavio Esposito, *Member, IEEE,* and Guido Marchetto, *Senior Member, IEEE,* .

***Abstract*—Despite years of research on transport protocols, the tussle between in-network and end-to-end congestion control has not been solved. 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 [2].

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 [3]–[7]. In particular, problems may occur in cellular and wireless networks, where TCP, i.e., Cubic since the default on many devices, misinterprets the stochastic packet losses as congestion, hence leading to performance degradation [7]. 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 [8], [9] and edge networks [6], [7].

This paper is an extended version of [1]

This work has been partially supported by NSF awards 1836906 and 1908574.

Alessio Sacco and Guido Marchetto are with DAUIN, Politecnico di Torino, 10129 Turin, Italy (e-mail: alessio_sacco@polito.it, guido.marchetto@polito.it).

Matteo Flocco and Flavio Esposito are with the Department of Computer Science, Saint Louis University, St. Louis, MO 63103 USA (e-mail: matteo.flocco@slu.edu, flavio.esposito@slu.edu).

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 [10]–[12]. 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 [11], [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 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 [8], [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*, 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 seventeen transport implementations. Some of these solutions were designed for wireless networks, such as Sprout [6] 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 consistent 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.

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. We then introduce our reinforcement learning framework and some of its main functionalities in Section III. In Section IV we explain our problem formulation and our protocol design, while Section V shows the rate stability analysis. Section VI summarizes our implementation, which is then evaluated in Section VII, where we show the benefits of our protocol. Finally, Section VIII concludes our paper and indicates some future directions.

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], Exll [29], BBR [25], and Data Center TCP (DCTCP) [8]. 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 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.

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]. The majority of these approaches are specifically designed to cope with a resource-constrained network, including IoT [10] and WANETs [11], [13], [31]; others instead address a wider range of network architectures [5], [14], [32]. Recent end-to-end congestion control solutions, such as Remy [5], PCC [14], PCC-Vivace [33], define an objective function to optimize the process of online actions definition, *e.g.*, on every ACK or periodically. Remy [5], 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 its variants, *i.e.*, PCC-Proteus [34] and PCC-Vivace [33], perform online optimizations. For instance, PCC adapts to the varying conditions in the network by 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-based 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.

Reinforcement Learning-based Congestion Control. Similar to previous solutions, we use ML to adapt the *cwnd* estimation, but setting this problem by means of Reinforcement Learning (RL). Recently, RL has permeated many congestion control mechanisms, such as Orca [35] and Aurora [21], where in Aurora, the previous Performance-oriented Congestion Control (PCC) protocol was extended with a Deep-RL approach. Our RL approach differs from prior work as our design combines features from both the transport and the network layers. Furthermore, our implementation uses inter-process communication between user and kernel space of a single host, without significant burden to the Linux kernel module. Moreover, unlike other reinforcement learning-based algorithms, the actions taken by our agents are guided by a utility function that has stability guarantees.

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 [8] modifies the Red Early Drop thresholds of ECN to achieve high throughput, high burst tolerance, while keeping queues empty hence experiencing low latency. RCP [36], XCP [37], and D^3 [9] modify switches behavior to feedback rates to end-hosts, while recent NATCP [38] and HPCC [39] leverages switches (or a centralized entity for NATCP) to send information about bottleneck links. 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. More recently, Swift [40], improved the intra-datacenter communication basing the congestion control on network delay. 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. Reinforcement Learning Framework

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: Background

In every reinforcement learning problem [41], 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 [42] 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. Deep Reinforcement Learning

To deal with the large state and action spaces, we approximate the Q-table via neural networks, reducing the total available actions. This technique is referred to as deep reinforcement learning and, specifically, deep Q-learning, that uses neural networks, parameterized by θ, to approximate the Q-function. Hence, the Q-values are now denoted as $Q(s, a; \theta)$, and the neural network is referred to as Q-network. In our algorithm, the Q-learning process consists of two parts: *(i)* the approximation of Q-values for the action selection, *(ii)* the Q-network update, where the loss between predicted Q-values and target Q-values is used to update the Q-network parameters θ, using the gradient method:

$$\theta \leftarrow \theta + \alpha \left(target\_Q - Q(s,a;\theta) \right) \nabla_\theta Q(s,a;\theta). \quad (2)$$

The "$target\_Q$" is a target value calculated as follows:

$$target\_Q = r + \gamma \max_{b \in actions} Q(s', b; \theta). \quad (3)$$

However, it can happen that the Q-function diverges due to dynamical and frequent changes in the target [41]. Therefore, a separate network is introduced, the target network. It is a copy of the Q-function and is used to calculate the target value. This approach is usually denoted in the literature as Deep Q Network (DQN), and we configure a periodic update of the target network with the current Q-function. This approach, however, arises several challenges, such as a complex and time-consuming learning phase, which can also cause catastrophic forgetting: the phenomenon in which the agent forgets how to perform previously trained tasks [43]–[45]. To efficiently solve these issues derived from using function approximation, motivated by other studies [20], [46], we use a technique called experience replay [47]. Experience replay consists of storing, at each time step, model transitions in a circular buffer called the replay buffer. Then, during training, instead of using the latest transition to compute the loss and its gradient, our agent computes them using a mini-batch of transitions sampled from the replay buffer. This approach leads to: better data efficiency by reusing each transition in many updates and better stability using uncorrelated transitions in a batch. However, since it can be difficult to use histories of arbitrary length as inputs to a neural network [46], it is common to have Q-function operate on fixed length representation of histories produced by a function $\phi(s_t)$ or ϕ_t for short. Although more recent versions of DQN have appeared proposing the use of a dueling DQN [48] or double DQN [49], we experienced that no particular differences are visible and, therefore, we stand with a simpler and less memory-consuming version as the one proposed in [20].

IV. Problem Formulation and Protocol Design

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 by the sender, used to select the next *cwnd* adjustment. We then describe this procedure in a schematic way.

A. Congestion Control via Reinforcement Learning

We now overview our primary components in the RL method, starting with our considered state set, then with the

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 Congestion (PNC) [packets]
11	Partial Network Availability (PNA)
12	Percentage of known network [%]

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, 11 and 12). 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 three features represent the partial information coming from the network (features 10 and 11), and a parameter stating the quantity of knowledge, as a percentage of the whole network (feature 12), respectively. In particular, the percentage of the known network is defined as the fraction of controlled network nodes in the path between a source and a destination divided by the total number of nodes in such a path. This value can be accessed for example via *traceroute*. Our partial network knowledge is constituted by two main metrics: *Partial Network Congestion* (PNC) and *Partial Network Availability* (PNA). *Partial Network Congestion* (PNC) represents an indicator of the known level of congestion within the network. In particular, 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 $|P_{in} - P_{out}|$. Given a source receiving statistics or updates from z switches on the path between a source and a destination, PNC is computed using the following equation:

$$PNC = \max(diff_1, diff_2, ..., diff_z). \tag{4}$$

While PNC informs about the current congestion level, and consequently, the loss rate occurring in the network, *Partial Network Availability* (PNA) informs about the available (bandwidth) resources in the network. It indicates the spare capacity of the network, in a similar way to [37]. For any link j, given C its capacity and *cr* the current traffic rate, we define the spare capacity on such a link, sc_j, as $\frac{C-cr}{C}$. Then, given w links on the path between a source and a destination, we define PNA as follows:

$$PNA = \min(sc_1, sc_2, ..., sc_w). \tag{5}$$

We choose PNC and PNA as they are 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-B. Nonetheless, Owl is able to automatically understand when network knowledge is hidden, impractical to obtain, or simply misleading. Inspired by the more known action masking [50], we decided to equip the solution with a *state masking* mechanism so that the same model has validity both in the presence and absence of the network feedback, as explained later in this section.

In defining our states, we also consider a history window of k values for each chosen feature of our state space. 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 VII-H). 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 in order to prevent 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. As a consequence, 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 offline training algorithm based on RL to update the *cwnd* properly.

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:

$$A = \{-10, -3, -1, +0, +1, +3, +10\}. \tag{6}$$

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. Ultimately, the intermediate 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. While the action set A in Eq. 6 represents our default setting, we designed the systems so that it is a policy that can be tailored to specific use cases. For example, the action of the RL model can be the value of the congestion window size (as we did) or other parameters 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, few losses, and low delay". In particular, the utility U_i of sender i is a function of throughput of client i (λ_i), packet loss rate for i (p_i), and the RTT of i (RTT_i), as follows:

$$U_i = \lambda_i - \delta_i \lambda_i \left(\frac{1}{1-p_i}\right) - \beta_i \log\left(\frac{RTT_i}{RTT_i^{min}}\right) \quad (7)$$

where $p_i \in [0, 1)$, and RTT_i^{min} is the smallest RTT observed over a sufficiently long period of time. In our implementation we consider a period of 10 seconds as suggested by other studies [15], [51]. The normalization of RTT $\left(\frac{RTT}{RTT^{min}}\right)$ and the logarithm function enable the applicability to various contexts and highly varying networks, *e.g.*, cellular networks, where RTT can abruptly change in a fraction of time; δ_i and β_i are two adjustable coefficients determining the importance of the components. For example, a δ_i larger than β_i implies that lower packet losses are preferable to the packet delay. These coefficients are per user as it is possible that applications might have different preferences. 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, studying thoroughly the behavior of our solution when it considers both aspects in the utility, *i.e.*, loss and delay, or when $\beta = 0$ and it works as a pure loss-based version. We refer to this latter version with the name of Owl-Loss since it only considers packet losses and not network delay.

State masking. Consider Table I, where we report the list of the 12 features that are considered in the DRL model. In some cases, only a subset of such 12 metrics is available. Therefore, to make our DRL model general enough and suitable regardless from the presence of the network knowledge (features 10 and 11 in Table I), we use a technique, known in the literature as masking, over the state space. We refer to this methodology as state masking, and consists of two steps. First, the features list is padded to fit a given standard length, so that the model always expects the same number of inputs. Once all samples have a uniform length, the features to be ignored are marked. By doing so, the model is informed of what are the padded values so that they can be ignored by the DRL when processing the data. Such state masking pre-processing can be viewed as adding an extra layer in front of the Q-network responsible for selecting the best action. The state masking mechanism can be applied even if the network knowledge is available, but it is not convenient to use, or it is convenient only partially. As we show in Section VII, there exist circumstances where weighting such information may lead to a significant performance increase of our solution. The extreme case is when this information is fully considered or ignored, while in between reside possible values of weight representing a model parameter learned by our algorithm during training.

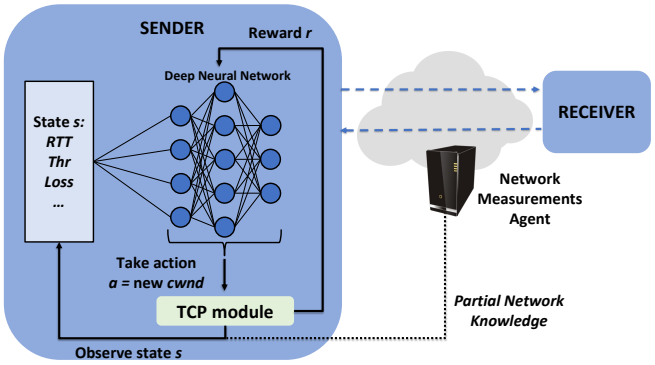


Fig. 1: Owl Overview: reinforcement learning sender's agent interaction with the network.

B. Owl Protocol Design

Consider Fig. 1, where we detail the main actions performed by the sender. All collected metrics are given to the Neural Network, and the protocol starts (Algorithm 1).

Algorithm 1 `Owl cwnd update`

Let t be the time step, and T total number of steps
Let S and D be the target source and destination
$F \leftarrow$ flow connecting S and D
At time $t = 0$ initialize $Q_0(s, a)$ with random wights and set reward r as in Eq. (7)
for $t = 1$ to T **do**
    Collect state vector s_t for flow F
    $\texttt{cwnd}^{\star}\texttt{(t)} \leftarrow \max_{cwnd} Q(\phi(s_t), s_t; \theta)$
    Set `cwnd` to $\texttt{cwnd}^{\star}\texttt{(t)}$
    Observe r and s'
    Set $\phi_{t+1} = \phi(s')$
    Store transition $(\phi_t, cwnd, r, \phi_{t+1})$ in replay memory
    Perform a gradient descent step according to Eq. (2)

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 PNC and PNA by controlling the underneath topology and notifies these quantities to the sender. Note that even when the network knowledge (feature 10 and 11 in Table I) is incomplete or unavailable, the neural network does not use the in-network features but our protocol can still provides valuable results (Section VII).

Once Owl has collected such values, it selects the next *cwnd* by choosing the "action" according to the mapping policy. During the training phase, the next *cwnd* value is selected according to the ϵ-greedy policy: With probability ϵ it selects the action randomly (exploration), and with probability $1 - \epsilon$ it selects the best action on the basis of the highest expected reward (exploitation). During the testing phase, the algorithm

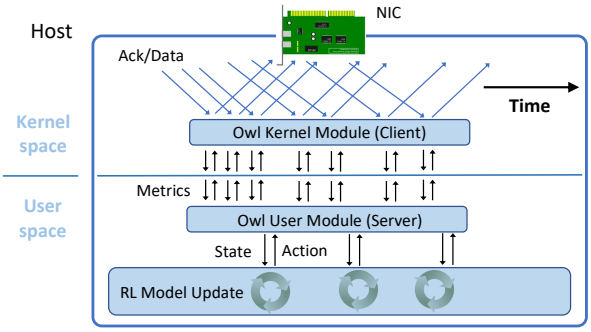


Fig. 2: Packets transmission with asynchronous interaction between the sender agent and the RL model agent.

avails the states, actions, and reward to select the best value for the *cwnd* (as shown in Algorithm 1); once the action is set, the program updates the experience memory and the Q-functions. 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.

We clarify the process dictating the packet transmission and the learning phase in Fig. 2. We design our solution with the goal of processing data efficiently and swiftly by an asynchronous communication between the kernel and user-space. By separating the responsibilities, we are able to transmit at line rate, but properly handling the complexity of keeping an RL process. The kernel module component can thus collect data and let the user-space module handle them, at a larger time scale using the logic to select the next *cwnd*. In particular, the reinforcement learning-based congestion controller agent accumulates network statistics from ACKs over a fixed period and sends the action asynchronously in a separate thread. Such separation of concerns between user and kernel space is also necessary since the RL model requires a considerable amount of memory, which may not be available within the kernel. At the same time, the user-space component provides immediate feedback to the kernel, communicating the new increment or decrement of the window using the model trained so far. In such a way, the TCP state can evolve regularly.

V. Stability Analysis

In this section, we focus on the utility's motivation, referring to the Eq. 7 when the weight $\beta_i = 0$. In particular, we show that processes running our Owl-Loss converge to a stable rate assignment. We demonstrate how 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:

$$U_i(\lambda_1, ..., \lambda_i, ..., \lambda_n) > U_i(\lambda_1, ..., x, ..., \lambda_n), \quad \forall i, \tag{8}$$

where $U_i(\lambda_1, ..., \lambda_i, ..., \lambda_n)$ denotes the sender i's payoff as a function of its and other strategies, and x is any non-negative sending rate. The following theorem holds.

Theorem V.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 7, the sending rates converge to a stable equilibrium. Moreover for every sender i, we have:*

$$\lambda_i = \frac{C\left(\frac{n}{\delta_i} - \hat{z}\right)}{n+1}, \tag{9}$$

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:

$$p_i = \begin{cases} 1 - \frac{C}{\sum_i \lambda_i} & \text{if } \sum_i \lambda_i > C \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

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 7, 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.

□

VI. 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 VII 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. Fig. 3 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.

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 IV).

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. As emerged in [52], exposing congestion signals, *i.e.*, RTT, losses, *etc.*, to an external module would enable providing new capabilities. 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 (e.g., delays and losses) and requires proper management of the socket channel. For these reasons, we

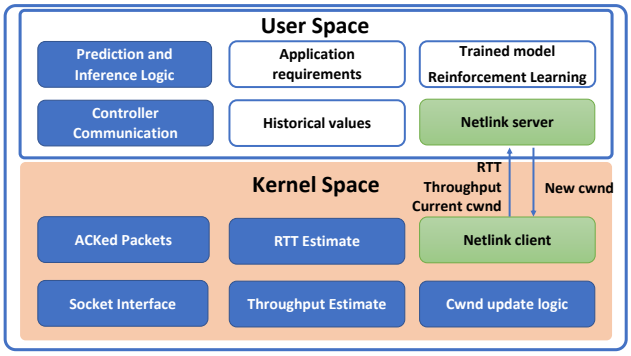


Fig. 3: Owl has a component that runs in the Linux kernel, and a component that runs at user-space to collect statistics to be used by our reinforcement learning algorithm.

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 [53].

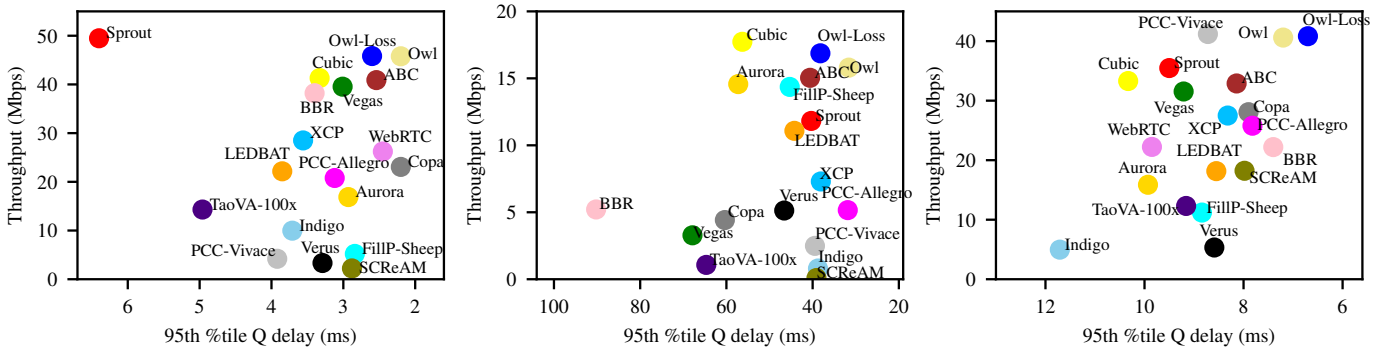
Moreover, our RL component is in charge of setting congestion window only during congestion avoidance phase of TCP. As we rely on Cubic for setting other TCP parameters, such as timeouts and threshold (ssthresh), we also adhere to the separation into two main phases during *cwnd* increment: slow start and congestion avoidance. The former starts with the initial *cwnd* and lasts until one packet gets lost or the window exceeds the ssthresh value. After overcoming this threshold, the congestion avoidance phase begins. Although the initial strategy is referred to as slow start, its congestion window growth is quite aggressive, and certainly more aggressive than the congestion avoidance phase. We have experienced that the most critical stage is the latter, since (as the name suggests) it attempts to avoid sending more data than the network is capable of forwarding, that is, to avoid causing network congestion. For this reason, we modify the traditional behavior of TCP only in the congestion avoidance phase, but we exploit years of research about TCP. This approach also balances the disadvantages of RL-based systems, well-known in the literature [21], [35].

VII. Protocol Evaluation

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

A. Trace-Driven Emulation Results

To evaluate our congestion control algorithm, we compare it with other solutions over LTE networks. We use a virtual network testbed and Mahimahi [54], an emulator that allows testing of various network conditions, also by means of real (cellular) traces. In particular, we run comparisons over traces collected from three of the largest US telecommunication providers, Verizon, T-Mobile, and AT&T. The network is emulated through namespaces via Mininet [55] and consists of the 14-node NSFNET topology. The transmission goes through a Software-Defined Network (SDN), where switches



(a) Verizon LTE (b) T-Mobile LTE (c) AT&T LTE

Fig. 4: **LTE Trace-driven emulation.** Owl vs. previous schemes (using RL or not) tested over three cellular network traces (top-right are better). In all cases, Owl outperforms our benchmark, and has the highest performance trade-off, on average, in our tested use cases.

TABLE II: RL model parameters used for training and computer network settings.

Model parameter	Value
Episodes/Epochs	500000
Steps	5000
Learning rate	0.006
batch_train	True
warmup_steps	10
γ	0.93
policy	Epsilon Greedy
neurons per-layer	512-256
Comp. Network parameter	**Value**
bottleneck link bandwidth [Mbps]	6 - 100
min RTT [ms]	1 - 40
buffer size [pkts]	3 - 2000

interact with a centralized controller (in our implementation, we used Ryu [56]). 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. 7, where $\delta = \beta = 0.5$ to give equal importance to packet loss and delay. Unless otherwise specified, we set a default percentage of known paths to be 20%, the Mininet network to have 1000-packet queue, and 0% random loss. To evaluate each protocol, we used a 95% confidence intervals, and average 30 experiments in which each sender-receiver pair runs *TCP iperf3* for 100 seconds. Our RL model is trained offline over an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz for 12 hours, varying different network conditions, *e.g.*, cellular and wired, with different knowledge percentages (0-100). The intervals of cellular traces (Verizon, T-Mobile, AT&T as explained later) are different from the ones of the testing phase in order to avoid visible tailoring to specific network scenarios and overfitting. We summarize in Table II the main parameters of our network and our RL model, such as the number of epochs and learning rate (see [57] for a full explanation of involved variables).

To understand how Owl performs compared to other solutions, we deployed our protocol over an emulated network created with Pantheon [58], a well-known fairly recent testbed developed to evaluate congestion control schemes. In particular, we compared Owl against seventeen other protocols, divided into five categories: (i) end-to-end TCP designs: Cubic [22], Vegas [24], BBR [25], Tao-VA [59], Copa [15], PCC [14] and its variants; (ii) end-to-end cellular, *i.e.*, LTE protocols: Verus [7], Sprout [6]; (iii) Machine Learning-based transport protocols: Indigo [58] and Aurora [21]; (iv) explicit congestion control: ABC [18] and XCP [37]; and (v) mixed schemes: LEDBAT [60], SCReAM [61], WebRTC [62]. For our LTE evaluation settings, we use the publicly available [54] 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) [63]) 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.

Fig. 4a-b-c shows that Owl performs efficiently in all tested scenarios. To study separately the Owl's loss-related properties from its latency-related properties, our experiments sometimes involve evaluating our protocol when the latency weight is $\beta = 0$, i.e., studying a purely loss-based variant. We refer to this version as *Owl-Loss* and to the default combination of weights simply as *Owl*, since its utility function is a combination of loss and delay-based components. In the case of Verizon LTE traces (Fig. 4a) Owl achieves both good throughput and 95$^{\text{th}}$ percentile per-packet-delay, and no other solution has shown a better combined throughput-delay performance. At the same time, we can observe that also Owl-Loss can simultaneously lower the delay, despite the fact that the RL reward was designed to achieve high throughput and low loss rate. Similar conclusions hold even for T-Mobile traffic (Fig. 4b) and AT&T (Fig. 4c), where both versions of Owl provide a desirable trade-off between throughput and delay. It is worth noticing that none of the other algorithms outperform Owl in these tested environments: our solution appears to be more stable across traces. Our solution, thus, offers the ability to adapt to different scenarios, as in diverse cellular networks, given the effective learning process performed by our agent. This adapting behavior is a consequence of the effective learning

performed by our agent, which motivates the choice of the RL framework.

Fig. 5 shows the shortcomings of transport protocols in use and the lack of adaptation required for a good transport protocol. The Fig. 5a 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, PCC, as it is one of the best performing within utility-based approaches, and the loss-based version of Owl, given its ability to foster high throughput in general and in cellular networks as manifested in previous results. 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 manner and closely approximates the desired behavior by learning the optimal action.*

This result is also confirmed in Fig. 5b where we plot the utility (Eq. 7) 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. Moreover, to generalize the findings about throughput-delays of Fig. 4, we now consider the combined effects over throughput and losses by means of the loss-based version of the utility. Likewise, we can observe how Owl-Loss 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 react promptly to the events, confirming how our state space constitutes a valid indicator of the network conditions and our mechanism can properly react.

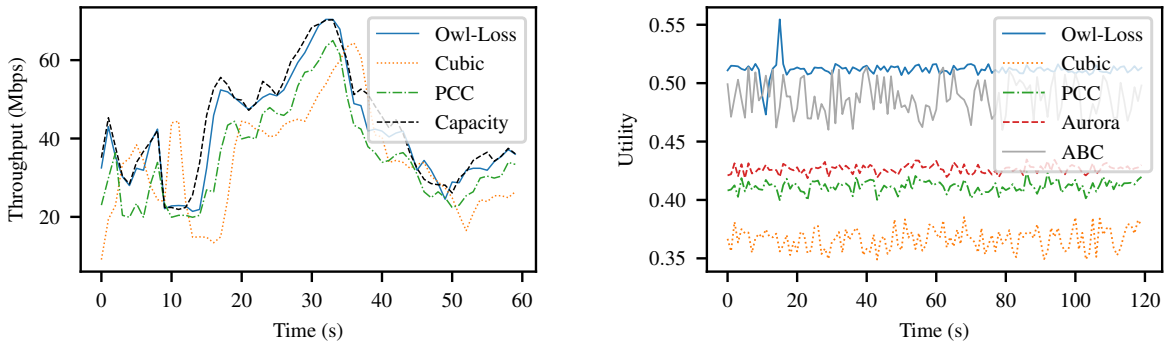
B. Network Knowledge Impact

We now discuss our experiments regarding the impact of the required network state knowledge that Owl needs to train the RL system effectively. Fig. 6 display the (a) throughput and the (b) RTT, when different transport protocols run over the same 14-node topology emulated on our local virtual network testbed. We set up our network where the maximum length of a path is 7 switches, all links are 100 MBps, the network load is at 40%, the base RTT is 30 ms, and the buffer size of 1 BDP. This network load is generated by sending UDP packets (via *iperf3*) until the desired network load is reached. Specifically, we compare against Cubic [22], as a reference end-to-end congestion control, Aurora [21], as a reference RL-based congestion control, and ABC [18], 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.

We can observe that when the number of known (or ABC-compliant) switches is more than 60%, our solution (i) provides better performance than ABC, (ii) speeds up the transmission in terms of throughput and (iii) reduces the end-to-end latency. Besides, ABC performs significantly worse than Owl when the number of ABC-compliant routers is relatively low. In conclusion, our evaluation reveal that even when either limited or very high number 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). However, we can also observe how the worst performance of Owl are seen approximately when half of the devices are controlled, leading to a lower throughput and higher delay than Cubic. We attribute such results to the fact that the RL agent cannot assign the proper importance to the incoming network states, resulting in occasionally misleading values. In such particular conditions, the performance of Owl are worse than those obtained using other end-to-end protocols (Cubic and Auorora) and a network-assisted solution (ABC).

This issue motivates us to use state masking and weight the incoming PNA and PNC information, as detailed in Section IV. Aside from enabling our solution to work as a pure end-to-end solution or as hybrid in-network / end-to-end, this technique allows to neglect the network feedback values when not beneficial. We then show the improvements obtained by this choice in Fig. 6d-e. In the case only partial network knowledge (around 50%) is available, Owl can still improve the overall performance. By masking the state space, the model does not suffer from a only partial visibility and can consistently outperform other protocols. Furthermore, these results validate that the value of PNC and PNA are beneficial to the algorithm, where PNC is a signal of slowing down the transmission given some traffic congestion, and PNA signals to increase the rate as some resources are underutilized.

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 Fig. 6c, 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]. The efficient solution design, as well as the set of information chosen to drive the experience, are key to this positive outcome. Another reason behind such improvements compared to other RL-based models is that, since we used the in-network information in different scenarios during the training, the general policy learned by the system is valid even when such network feedback is absent, as in Owl-0.



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

Fig. 5: **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.

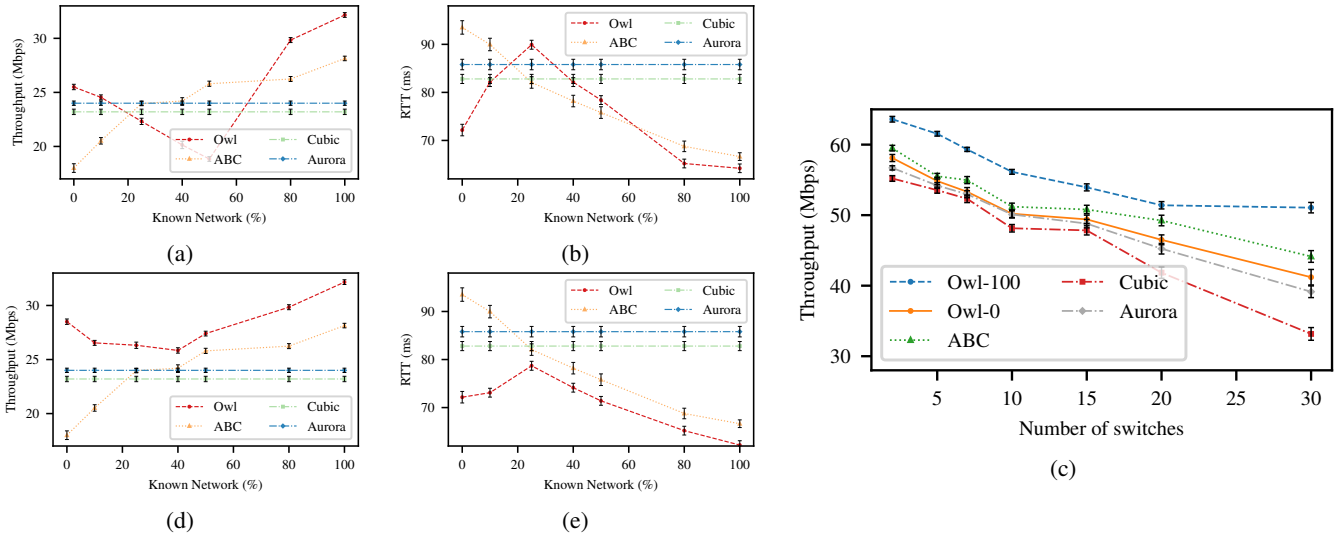


Fig. 6: **Network Knowledge Impact on Performance. (Without masking)** (a) Throughput and (b) RTT of Owl protocol for increasing percentage of known network. **(With masking)** (c) Throughput performance with or without network knowledge averaged over different network topologies and increasing number of informing switches. (d) Throughput and (e) RTT of Owl protocol for increasing percentage of known network.

C. Buffer Size Impact

Later, we consider the bottleneck saturation with varying buffer sizes. We run a single flow for 100 seconds on an emulated bottleneck link, comparing the throughput, the RTT inflation, and the overall power. The network is set up with 30ms of base RTT and capacity on the bottleneck link of 50 Mbps. The RTT inflation is calculated as:

$$RTT\ inflation = RTT - base\ RTT, \tag{11}$$

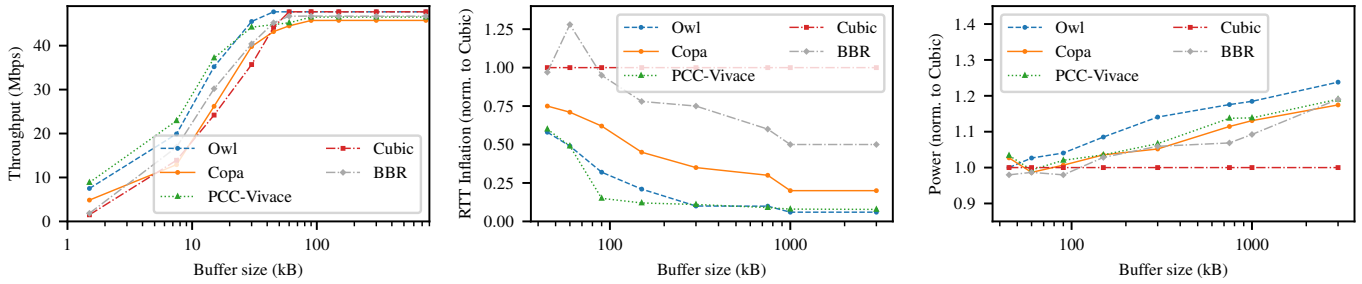
while the Power is:

$$Power = \frac{throughput}{delay}, \tag{12}$$

following the definition in [64]. A high power indicates a high throughput as well as a low delay in the network.

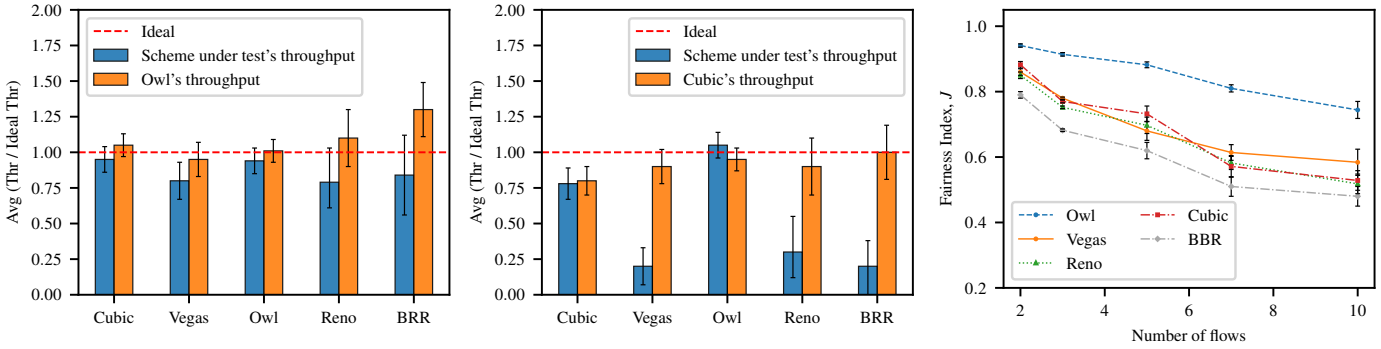
Starting by analyzing the throughput in Fig. 7a, we compare against a utility-based protocol as PCC-Vivace that can efficiently handle large and small buffers. Rather than Vegas, which performs similarly to Cubic, we consider a delay-driven protocol as Copa. For the sake of clarity, confidence intervals in the graphs are omitted because they are negligible. From the graph, we can observe how Owl needs almost the same buffer size of PCC-Vivace to achieve at least 90% capacity utilization (45Mbps). However, Owl converges to a higher throughput value.

Regarding the delay evolution shown in Fig. 7b, Owl can prevent the bufferbloat phenomena, as opposed to Cubic (used here as baseline). Having the delay notion in the reward function, our solution can learn when to reduce the *cwnd* to avoid filling up the buffer. Owl and PCC-Vivac can maintain the inflation a quarter of Cubic's inflation when the buffer is greater than $150kB$. Fig. 7c confirms the ability of Owl to jointly maximize the throughput while minimizing delay. Our protocol can achieve this objective even more accurately than PCC-Vivace.



(a) (b) (c)

Fig. 7: **Buffer Size Impact.** The evolution of (a) Throughput, (b) RTT inflation, and (c) Power of Owl for increasing size of buffer of bottleneck link.



(a) (b) (c)

Fig. 8: **Fairness and Friendliness Analysis.** (a) Different schemes utilization and how they share the available bandwidth. The ideal fairness value is 1. Owl-Loss 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) We summarize the impact on the fairness using the Jain's index when the same protocol is employed for an increasing number of competing flows.

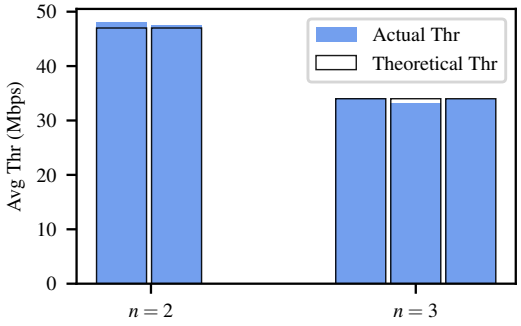


Fig. 9: Experimental analysis of throughput when multiple Owl-Loss senders transmit concurrently. The results confirm the theoretical analysis in Eq. 9.

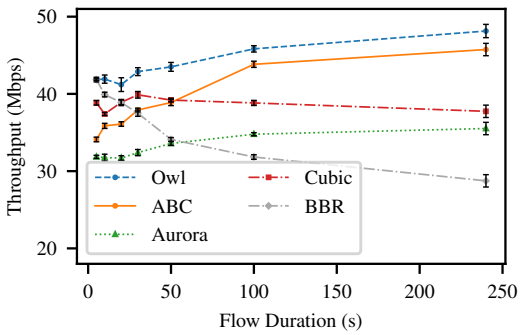


Fig. 10: Throughput evolution for increasing duration of the communication. Owl is suitable for diverse type of applications.

D. Owl Fairness and Friendliness

In this subsection we evaluate the fairness among several flows all running Owl-Loss and competing with each other, as a verification of the stability analysis. We 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 30ms RTT and a bandwidth of 50 Mbps. First, we evaluate the friendliness against all congestion control solutions that are installed on Linux by default (Fig. 8a). 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 (Fig. 8a.) 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 (Fig. 8b). By having the objective of minimizing the losses in combination with some network knowledge, our protocol can understand when reducing the sending rate to improve its and others' performance. 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.

To further support this conclusion, we also consider the Jain's fairness index [65], one of the most widely used measures of fairness. The Jain's index is formally defined as:

$$J = \frac{\left(\sum_{i=1}^{k} f_i\right)^2}{k \sum_{i=1}^{k} f_i^2}, \tag{13}$$

where f_i is the throughput for the i-th connection, k is the number of flows, and $0 \leq J \leq 1$. One of the advantages of this metric is its intuitiveness, *i.e.*, a large value of J represents a more fair resource allocation. We investigate how the Jain's index evolves over time for multiple flows. Results given in Fig. 8c confirm that Owl can increase the fairness when compared to other protocols and outperforms other learning-based solutions. In particular, even when more flows contend the same resources, our solution is able to accommodate the demands and provide an equal throughput to these transmissions.

Finally, to demonstrate the validity of Eq. 9, we quantify the throughput of multiple Owl-Loss flows sharing a bottleneck of capacity 100 Mbps. We measure the average throughput when the number of senders varies between 2 and 3, and all of them set the value of $\delta = 0.7$ (and $\beta = 0$). As shown in Fig. 9, where we report both the theoretical value and the obtained one, the empirical analysis confirms the stability results expressed in Section V.

E. Evaluating the Impact of Traffic Flows Duration Diversity

In this subsection, we analyze the effects of shorter flows (Fig. 10). We observe how diverse approaches to explicit congestion control perform well for long-lived flows, where the devices have time to notify the hosts and the traffic can be optimized. The performance of end-to-end congestion control protocols is generally suboptimal for longer communications. We can see how our solution outperforms the other protocols picked as benchmarks for both short-lived and long-lived flows. We attribute the reason for such better performance to our proposed integration of in-network signals into a learning module; such technique mitigates the drawbacks of the pure explicit congestion notification *à la* ABC [18] and standard end-to-end approaches as Cubic [22]. This result made explicit the ability of Owl to adhere to the network conditions it encounters.

F. 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 [26]. In these experiments, we evaluate how the congestion control schemes under consideration behave across two federated GENI aggregates. We measure the performance of each schema when competing with other flows. To evaluate our protocol in these realistic settings, we average the throughput and end-to-end delays obtained over 60-second flows, while the senders share a bottleneck link with 3ms RTT and a bandwidth of 100 Mbps.

We compared the performance of our protocol with all other protocols currently available on Linux, considering also the *loss* version of Owl. The results are summarized in Fig. 11a, where we evaluate the ability to achieve the goal of high throughput and low loss rate defined by our utility function. Our prototype evaluation deployed in real settings matches our emulation results: Our implementation can jointly achieve high throughput and a low loss rate when compared to other solutions, effectively balancing the two components. Not only, but the two versions provide similar results, with a slight improvement in the loss rate for the loss-based version. Since the general version of Owl provides valid results, we continue the experiments with this version in what follows.

We then analyze if our solution is agnostic to the flow duration, as in our emulated scenarios. To this end, we perform a set of experiments and report the results in Fig. 11b. Comparing the obtained throughput, we observe how Owl is beneficial regardless of the flow duration.

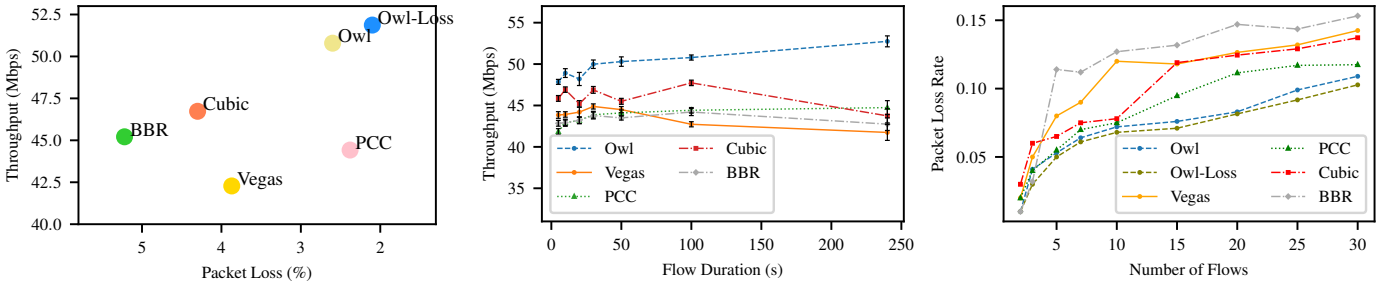
Moreover, we consider how an increased number of concurrent flows damages the single transmission (Fig. 11c). Clearly, as the competing flows increase, the loss rate increases as well. However, Owl is able to maintain a low loss rate, ensuring stable transmissions.

G. Study of Our State and Action Space

In the rest of this section we show the results that motivated our design choice. In particular, we analyzed *what is the most performant state and action set that our reinforcement learning algorithm should use*? This analysis is often omitted in prior reinforcement learning work, while it has been always lacking in previous work on RL for TCP congestion control [17], [21], [35].

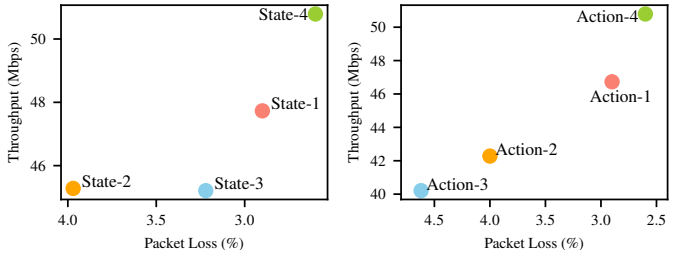
State Space. We start by considering different set of states and we define the following policies: *(i) State-set-1* or simply *State-1* a state space limited to three features, as described in [21]. This vector of statistics consists of: latency gradient, *i.e.*, the derivative of latency with respect to time; the latency ratio, *i.e.*, the ratio of the current's mean latency within the time interval to minimum latency observed in the connection's history; sending ratio, *i.e.*, the ratio of packets sent to packets acknowledged by the receiver. *(ii) State-2* a twenty features list as mentioned in [51]. This list includes features as in-flight bytes, RTT, RTT variation, and is thus partially overlapped with our state space, but includes more statistics regarding the throughout and lost packets. [1] *(iii) State-3* in which we consider nine metrics, as in [35]. This policy considers the current *cwnd*, together with both statistics calculated during the monitoring interval, *i.e.*, throughput, loss rate, delay, ACKs, elapsed time, and metrics regarding the overall communication, *i.e.*, the minimum delay, maximum throughput, and smooth RTT measured since the transmission started. It is worth noticing that all these space-set policies, similar to Owl, consider a fixed-length history. Our proposed

[1]The complete list of features is as follows: last RTT, smooth RTT, min RTT, standing RTT, RTT variance, delay, *cwnd*, inflight bytes, writable bytes, sent bytes, received bytes, re-transmitted packets and bytes, ACKed bytes, lost packets and bytes, throughput, number of retransmission due to timeout, number of timeouts, flag indicating congestion. For further details refer to [51].



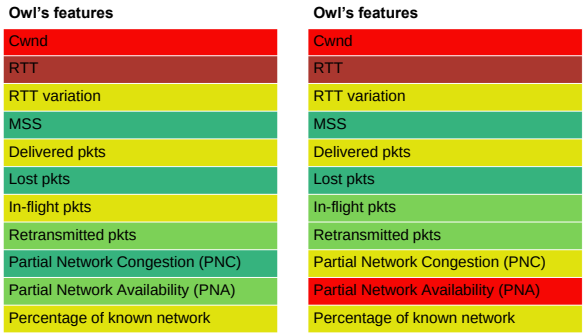
(a) (b) (c)

Fig. 11: **GENI testbed evaluation.** (a) Throughput-loss rate trade-off for kernel-level solutions over real networks. Owl optimizes the two quantities simultaneously. (b) Effect of diverse flow duration over the achievable throughput. (c) Performance degradation when the number of communications increases. Our solution can well mitigate this circumstance.



(a) (b)

Fig. 12: **Design evaluation.** Analysis of different (a) state spaces and (b) action spaces. The results motivate our assumptions and our choices.



(a) Known net=20% (b) Known net=70%

Fig. 13: **Features Importance Analysis.** Visualization of the role of each feature in the decision process from the highest degree of importance (dark red) to the lowest (dark green) when (a) the percentage of known network is low and (b) high.

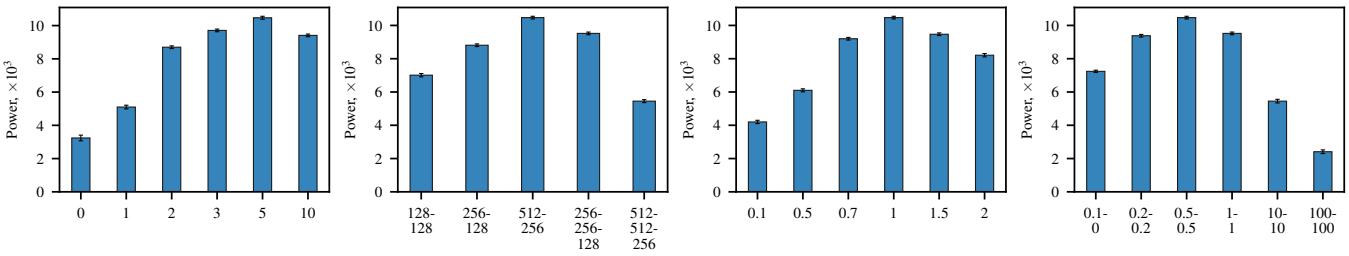
space set, detailed in Section IV, is denoted in our evaluation figures with *State-4*. We offline trained these alternatives under the same conditions encountered by ours.

In Fig. 12a we show our evaluation results on the impact of such state set policy in the throughput / losses diagram. The considered scenario refers to the same GENI network, when two flows compete on this bottleneck, in order to accentuate the possible congestion occurrences. *We observe how our design (State-4) successfully balance the importance of having information from each feature and the complexity of handling such a large state space.* In particular, this experiment shows how our choice of network statistics that we use as features represents effectively the TCP transmission state. Aside from the in-network metrics, all other statistics are easily obtained by our kernel module, thus having a minor impact on the data collection process. Besides, the results suggest that also network knowledge is precious and beneficial rather than a burden.

Action Space. After analysing the impact of the state space, we focus on the action space to assess what is the most effective set. Specifically, we evaluate alternative modification of the congestion window, as proposed in other RL-based approaches. We denote with an action set the following: *(i) Action-1* $\{0, -10, +10, \times 2, /2\}$ as in [51], *(ii) Action-2* $cwnd \times 2^{\alpha}$ where $-2 < \alpha < 2, \alpha \in \mathbb{R}$, as in [35], *(iii) Action-3* $\{x \times (1+0.025a), x/(1-0.025a)\}$ where $a \in \mathbb{R}$ and x is the sending rate, as in [21]. Comparing the performance of these action set allows us to evaluate *(i)* a different set of discrete actions operating directly on the TCP window, but including more aggressive actions that multiply and divide the current value; *(ii)* a large action space, composed by continuous values (α), and used by an agent acting on the window; *(iii)* a large action space, composed by continuous values (a), and used to set the sending rate. We report results in Fig. 12b, where we refer to the actions of Section IV as *Action-4*. It is worth noticing that these alternatives only refer to the action space and not to the algorithm applied in the congestion control protocol. Besides, also in this case, we offline trained the alternatives before the test experiments.

The obtained results strongly support our choice. First, a discrete and limited set of actions shortens the convergence time while assuring adequate exploration. Second, operating directly on the window reduces the operations in both user and kernel space. Third, our possible actions allow moderately increasing and decreasing the *cwnd* without incurring drastic and sharp changes, which are harmful to TCP communication states.

In our considered state space then, we study the importance of each feature in the decision process. Although more recent and more advanced techniques can be used, such as PCA [66], Occlusion and Saliency Maps [67], they mostly find applicability in supervised and unsupervised learning problems.



(a) History length, k (b) Neurons per layer (c) Requests Interval Period (s) (d) $\delta - \beta$ combinations

Fig. 14: **Sensitivity analysis.** This analysis is 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. (d) Combinations of the coefficients δ and β in the reward function.

Therefore, given the deep RL nature of our model, we use an approach based on observing the weights of the neural network (NN), where the relative importance of a specific variable can be determined by identifying all weighted connections between the nodes of interest [68]. This procedure is based on the observation that the weights dictate the relative influence of information that is processed in the NN such that input variables that are not relevant in their correlation with a response variable are suppressed by the weights. Then, since we have a matrix in input, we average the obtained results over the historical k values to determine the importance of each single feature. We report in Fig. 13 the graphical visualization of the features' importance for (a) limited visibility of the network and (b) higher visibility. We can observe how the network feedback plays a minor role when the visibility is limited while, among the end-to-end signals, the values of *cwnd*, RTT, its variation, and delivered packets guide the model's decision for the next *cwnd* value.

H. Sensitivity Analysis

In this subsection, 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), the frequency at which we should collect in-network measurements, and the default values of coefficients δ and β of the utility function. We evaluate how these values affect performance with 95% confidence intervals, obtained with 30 trials on the GENI testbed with the same network conditions of the previous subsection. First, we examine the impact of the length of the action history in the augmented state space. Fig. 14a shows the power obtained at training for varying values of the state history length k. 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. Fig. 14b exhibits the power 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 combination of throughput and delay. Hence, we empirically set this configuration as the default of our system, but we realize that this configuration is a policy.

We then investigate the selection of the most valuable measurement request interval (Fig. 14c). We note that, when the network measurements are gathered every 1-second, the power is at its maximum. This value also guarantees the freshness of data without incurring in too frequent updates.

Finally, we consider how the coefficients δ and β in Eq. 7 impact the power performance. Empirically, we can observe how giving equal importance to the delay and packet loss, *i.e.*, $\delta = \beta = 0.5$ leads to the highest power value. As mentioned for previous analyses, we set these values as default, but they constitute a policy that can be changed if different desiderata hold, *e.g.*, if a very low delay in the transmission is preferable in spite of higher losses, it is advisable to set a β higher than δ.

VIII. Discussion and 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.

Since our solution uses reinforcement learning, it inevitably inherits its shortcomings. For example, its performance depends on the balance between exploration and exploitation of the learning policy. Moreover, in computer networks with conditions different from those tested in our evaluation results, e.g., satellite or data center networks, performance results of Owl may not hold. As such, despite the obtained promising

results, we believe that exploring whether or not the reinforcement learning model used in Owl can be transferred to other computer network scenarios is an interesting open problem. Knowledge transfer [69], [70] in general and generalizability of reinforcement learning in particular [71]–[73] are active areas of research in AI/ML, and hence may apply even in congestion control.

REFERENCES

[1] A. Sacco, F. Matteo, F. Esposito, and G. Marchetto, "Owl: Congestion control with partially invisible networks via reinforcement learning," in *IEEE INFOCOM-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '10)*, 2010, p. 63–74.

[9] C. Wilson *et al.*, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41. ACM, 2011, pp. 50–61.

[10] 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.

[11] 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 (BROADNETS)*. IEEE, 2005, pp. 484–493.

[12] 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.

[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 *Proceedings of the 36th International Conference on Machine Learning (ICML)*. PMLR, 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. Accessed: 2022-10-15. [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 *IEEE INFOCOM-IEEE 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-IEEE Conference on Computer Communications*, vol. 4. IEEE, 2004, pp. 2490–2501.

[29] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, and K. Lee, "Exll: An extremely low-latency congestion control for mobile cellular networks," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '18)*, 2018, pp. 307–319.

[30] R. Boutaba *et al.*, "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] 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.

[34] T. Meng, N. R. Schiff, P. B. Godfrey, and M. Schapira, "PCC proteus: Scavenger transport and beyond," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, 2020, pp. 615–631.

[35] 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 (SIGCOMM '20)*, 2020, p. 632–647.

[36] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 05)*, 2005, pp. 15–28.

[37] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '02)*. Association for Computing Machinery, 2002, pp. 89–102.

[38] S. Abbasloo, Y. Xu, H. J. Chao, H. Shi, U. C. Kozat, and Y. Ye, "Toward optimal performance with network assisted TCP at mobile edge," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2019.

[39] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "HPCC: High precision congestion control," in *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, 2019, pp. 44–58.

[40] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, 2020, p. 514–528.

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

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

[43] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in cognitive sciences*, vol. 3, no. 4, pp. 128–135, 1999.

[44] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[45] D. Kumaran *et al.*, "What learning systems do intelligent agents need? complementary learning systems theory updated," *Trends in cognitive sciences*, vol. 20, no. 7, pp. 512–534, 2016.

[46] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[47] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.

[48] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning (ICML)*. PMLR, 2016, pp. 1995–2003.

[49] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[50] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," *arXiv preprint arXiv:2006.14171*, 2020.

[51] V. Sivakumar, T. Rocktäschel, A. H. Miller, H. Küttler, N. Nardelli, M. Rabbat, J. Pineau, and S. Riedel, "Mvfst-rl: An asynchronous rl framework for congestion control with delayed actions," *arXiv preprint arXiv:1910.04054*, 2019.

[52] A. Narayan, F. Cangialosi, D. Raghavan *et al.*, "Restructuring endpoint congestion control," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, 2018, pp. 30–43.

[53] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, "Communicating between the kernel and user-space in linux using netlink sockets," *Software: Practice and Experience*, vol. 40, no. 9, pp. 797–810, 2010.

[54] 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.

[55] 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.

[56] Ryu controller. Accessed: 2022-6-7. [Online]. Available: https://ryu-sdn.org/

[57] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[58] 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.

[59] 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.

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

[61] I. Johansson, "Self-clocked rate adaptation for conversational video in lte," in *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop (CSWS '14)*, 2014, pp. 51–56.

[62] 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.

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

[64] A. Giessler, J. Haenle, A. König, and E. Pade, "Free buffer allocation—an investigation by simulation," *Computer Networks (1976)*, vol. 2, no. 3, pp. 191–208, 1978.

[65] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[66] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[67] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[68] J. D. Olden and D. A. Jackson, "Illuminating the "black box": a randomization approach for understanding variable contributions in artificial neural networks," *Ecological modelling*, vol. 154, no. 1-2, pp. 135–150, 2002.

[69] Z. Zhu, K. Lin, and J. Zhou, "Transfer learning in deep reinforcement learning: A survey," *arXiv preprint arXiv:2009.07888*, 2020.

[70] I. Higgins, A. Pal, A. Rusu, L. Matthey, C. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner, "DARLA: Improving zero-shot transfer in reinforcement learning," in *International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 1480–1490.

[71] Z. Allen-Zhu *et al.*, "Learning and generalization in overparameterized neural networks, going beyond two layers," in *Advances in neural information processing systems (NeurIPS)*, 2019, pp. 6158–6169.

[72] H. Wang, S. Zheng, C. Xiong, and R. Socher, "On the generalization gap in reparameterizable reinforcement learning," in *International Conference on Machine Learning (ICML)*. PMLR, 2019, pp. 6648–6658.

[73] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *International Conference on Machine Learning (ICML)*. PMLR, 2019, pp. 1282–1289.

Alessio Sacco received the received the M.Sc. degree (summa cum laude) and the Ph.D. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Torino, Italy, in 2018 and 2022, respectively, His research interests include architecture and protocols for network management; implementation and design of cloud computing applications; algorithms and protocols for service-based architecture, such as Software Defined Networks (SDN), used in conjunction with Machine Learning algorithms.

Matteo Flocco received the M.Sc. degree in Computer Science from Saint Louis University, where he worked as a research assistant in the networking group with Dr. Flavio Esposito. His research interests mainly involve computer networks, with a particular focus on Software-Defined Networking and congestion control algorithms, and machine learning applied to network management. Currently, he works as a Full Stack Developer for Blue Reply where he develops logistics softwares for large companies.

Flavio Esposito is an Associate Professor with the Department of Computer Science at Saint Louis University (SLU). He received an M.Sc. degree in Telecommunication Engineering from the University of Florence, Italy, and a Ph.D. in computer science from Boston University in 2013. Flavio's main research interests include network management, network virtualization, and distributed systems. Flavio is the recipient of several awards, including several National Science Foundation awards and the Comcast Innovation Award in 2021.

Guido Marchetto received the Ph.D. degree in computer engineering from the Politecnico di Torino, in 2008, where he is currently an Associate Professor with the Department of Control and Computer Engineering. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.