Mutant: Learning Congestion Control from Existing Protocols via Online Reinforcement Learning

Lorenzo Pappone
Computer Science Department
Saint Louis University

Alessio Sacco
DAUIN
Politecnico di Torino

Flavio Esposito
Computer Science Department
Saint Louis University

Abstract

Learning how to control congestion remains a challenge despite years of progress. Existing congestion control protocols have demonstrated efficacy within specific network conditions, inevitably behaving suboptimally or poorly in others. Machine learning solutions to congestion control have been proposed, though relying on extensive training and specific network configurations. In this paper, we loosen such dependencies by proposing *Mutant*, an online reinforcement learning algorithm for congestion control that adapts to the behavior of the best-performing schemes, outperforming them in most network conditions. Design challenges included determining the best protocols to learn from, given a network scenario, and creating a system able to evolve to accommodate future protocols with minimal changes. Our evaluation on real-world and emulated scenarios shows that Mutant achieves lower delays and higher throughput than prior learning-based schemes while maintaining fairness by exhibiting negligible harm to competing flows, making it robust across diverse and dynamic network conditions.

1 Introduction

Years of research on congestion control algorithms for TCP [44] have produced creative solutions departing from TCP Vegas [13], Fast [30], Compound [47], and the more recent BBR [15], to name a few, with improvements in performance results of more than 15% compared to Cubic [25], the default in most Linux systems. BBR, rather than traditionally relying on packet loss to adjust the congestion window, (*cwnd*), uses round-trip time (*rtt*) and throughput data to decide how fast to transmit data packets. TCP Vegas, a delay-based protocol, only looks at the *rtt* to adjust the congestion window. An increasing *rtt*, due, for example, to a packet drop, causes Vegas to reduce its packet sending rate, while a steady *rtt* causes Vegas to increase its sending rate until the *rtt* increases again. These examples highlight that traditional congestion control solutions rely on current network stats such as *rtt*, throughput, and packet loss. As such, they are mostly reactive (not proactive) to sudden network changes. In addition, they rarely look at the past behaviors of the network or their own historical behaviors in their analysis.

With the surge of machine learning performance in recent years, promising approaches using Deep Learning (DL) or Reinforcement Learning (RL) techniques have been proposed to adjust performance, mostly by tuning the congestion window [29]. As an example, Aurora [28], a variant of the Performance-oriented Congestion Control (PCC) [21], uses Deep Reinforcement Learning to adjust the congestion control actions empirically by observing the connection state and the experienced performance. Owl [43], Orca, [2], and Marten [39], three other RL-based congestion control protocols, use Deep Q-Learning to determine the next congestion window using metrics retrieved from the network stack. These approaches share the idea of solving the congestion control problem by instructing an agent to dynamically learn the optimal *congestion window (cwnd)* via the RL framework. Despite their learning ability as the network environments change, their design is inherently limited by the training phase. Congestion control parameters often exhibit complex interactions and dependencies with various network factors, including traffic load, topology changes, and routing dynamics. Even when enough samples of the significant states are collected *offline*, i.e., ahead of time, data drift problems remain, and protocols experience performance degradations for their lack of adaptability under new, unseen, changing conditions.

The good news is that years of research on congestion control protocols have shown that some protocols can learn how to perform well by observing specific network input signals, and even though they fail in scenarios for which they were not optimized, there is often another (existing) protocol that does perform better in such scenarios. *What if we could learn from a team of protocols, each performing well in a few complementary scenarios, without requiring exhaustive and impractical data collections and without requiring the one-size-fits-all protocol?*

In this paper, we propose *Mutant*. While such a design is inspired by the so-called imitation learning [20], deciding

which protocol Mutant should “imitate" or “mutate" into is also a computationally hard problem even if we choose the best-performing k protocols of the team offline. To solve this problem bounding the optimality gap, we use the team selection theory [31]. In particular, our algorithm solves the following problem: *Given a set of congestion control algorithms available, which subset should we input to our online reinforcement learning algorithm, so that our selection is most likely to bring the best performance, i.e., smaller delays, higher throughput, or both, for each network environment and congestion state?* We reduce this problem from the top-*k* players in cooperating games and discuss the team selection algorithm used by Mutant in Section 5.2. There are tens of CC protocols to choose from and we show that having as many protocols as possible leads to inefficiencies, not only overhead. Similar problems have been applied in the context of supervised machine learning, where individual features are modeled as players and feature subsets as coalitions that yield the best predictive performance [19].

In Section 4 we justify the chosen subsets of protocols per each network scenario and discuss our lessons learned. Among those, (1) the datasets available and the choice of protocols to learn from are more important than the learning algorithm, and (2) learning from too many protocols is counterproductive: the performance degrades because, intuitively, we explore too much and we do not exploit enough.

Aside from algorithmic challenges, to deploy a protocol that mutates its behavior based on the underlying available user-space and kernel implementations, we needed to solve several system design challenges as well (Section 3). For example, our kernel module implementation should be able to quickly mutate into one of the congestion control algorithms used to learn with minimal overhead. For performance reasons, we focused on learning from congestion control protocols already implemented in the kernel, such as Cubic, Hybla, and TCP Vegas, but Mutant can be easily adapted to learn from any transport protocol that uses sockets and returns congestion window updates, including ML-based protocols. Our evaluation results (Section 6), on real network traces and real-world traffic, show that *Mutant* consistently performs among the best protocols in all scenarios analyzed, for both delay-sensitive and bandwidth-sensitive applications. We also analyzed *Mutant*'s fairness using the notion of harm [50], showing promising results. From our evaluation, it is also clear that the choice of which protocol set *Mutant* is programmed to emulate influences its performance, just as humans should not learn from anyone in any scenario. We review existing literature in Section 7 and we present our conclusion in Section 8.

2 Mutant: Motivation and Design Overview

Learning From Existing Protocols. The main insight behind our protocol design is the idea that there is no one-size-fits-all congestion control for all scenarios, as confirmed by our results in Figure 1. These results show the ratio between throughput and delay, and the fairness performance of four widely used TCP protocols. We consider the following six network environments sampled from our experimental sets: *fixed bandwidth*, *low capacity*, *high rtt*, *large queue* i.e., buffer of the bottleneck router, *mixed conditions* i.e., large queue with high RTT, *step* i.e., with regular and abrupt channel bandwidth fluctuations. In Section 6.1, we detail these testing scenarios. We observe that no protocol is dominant in all the cases for single-flow settings. Cubic, for example, performs well in stable conditions but fails across fluctuating bandwidth conditions (step scenario). BBR2 [16], on the contrary, handles these abrupt changes but is suboptimal for networks with low capacity. When we instead consider a multi-flow scenario, focusing on the fairness of these protocols, expressed in terms of *harm*, a notion introduced in [50] and defined in Section 6.4, we can see that, e.g., BBR2 harms others in many use cases but not in all scenarios.

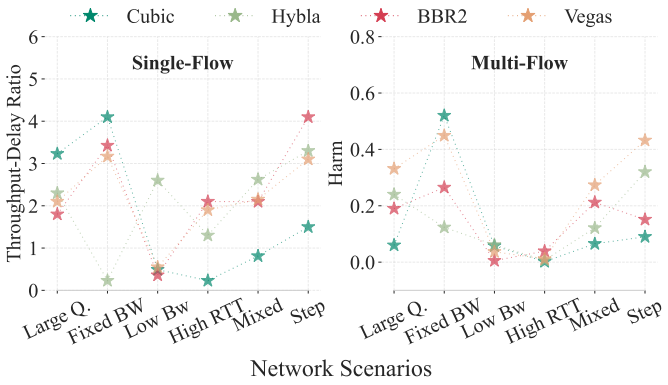


Figure 1: Motivation for our study: Performance of a few protocols in six different network environments, tput/delay on a single-flow (left) and harm [50] in multi-flow scenarios (right): low values translate to high fairness. Every protocol performs well in only a selected group of network scenarios.

The Challenge of Generalization. While adaptive and learning-based solutions have been explored, we believe that learning from past congestion window values is insufficient and poorly adapted to unseen network conditions. Current approaches attempting to achieve adequate model generalization require a long offline training phase, e.g., [42, 43, 55]. In Figure 2, we compare one state-of-the-art model as Sage [55] against our online model. The offline learning process requires vast data collection, expensive resources, and long training sessions. Mutant quickly adapts to the network scenario while achieving comparable or superior performance to the data-driven model. Motivated by these results, we propose a system that quickly adapts to diverse conditions and learns in real-time from ongoing interactions with the underlying network environment. In addition, our design principle is to offer the community a system that, rather than autonomously adapting

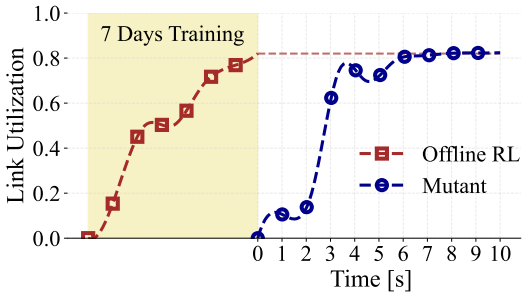


Figure 2: Mutant reduces the learning time overhead compared to pretrained, data-driven ML models for congestion control.

the best *cwnd* update strategy, considers other features to learn what would be the best protocol in a given network state.

Mutant Overview. At the core of our solution, *Mutant*, lies on an online Reinforcement Learning (RL) algorithm that *(1)* quickly learns from existing congestion control schemes by a real-time policy switching mechanism at fine-grained intervals implemented in kernel, *(2)* minimizes the reaction time to network changes, and *(3)* relies mostly on online data with minimal offline adjustments. We present the overview of Mutant in Figure 3, highlighting its main components. Our online learning logic decides on the best CC scheme in real-time during the communication. Two main components dictate the selection of the best scheme at each time step: the *Protocol Manager* and the *Learning Module*. The *Protocol Manager* is a pluggable kernel module implemented following the structure of the existing CC schemes in the Linux kernel. Indeed, Mutant is a kernel module that can be loaded as another kernel-based TCP congestion control protocol (i.e., Cubic, BBR2, Hybla, etc.). This component supports the key low-level functionalities of Mutant's system: *(1)* runs the real-time CC scheme switching logic, *(2)* maintains the state of each protocol during the execution, and *(3)* communicates the network statistics to the user-space *Learning Module* module.

The *Learning Module*, instead, is a user-space component that handles the algorithmic operations within our system. This module is responsible for executing *(1)* the online reinforcement learning process and *(2)* the best starting protocols using a *top-k* selection schema (Section 4). At each step of the RL loop, Mutant switches the CC scheme in real-time, by sampling from a pool of different protocols in the kernel. The user-space module will learn the relationship between the scheme selected and the features collected at each time step, with the primary goal of maximizing the throughput and minimizing the delay.

While designing our solution, we attempted to answer two key questions: *(1)* What pool of CC protocols should we consider in our deployment? *(2)* Which schema should we choose at runtime? To answer the first question, we questioned if all the CC schemes should be kept in the pool or if selecting only a subset of them would benefit the overall performance of our

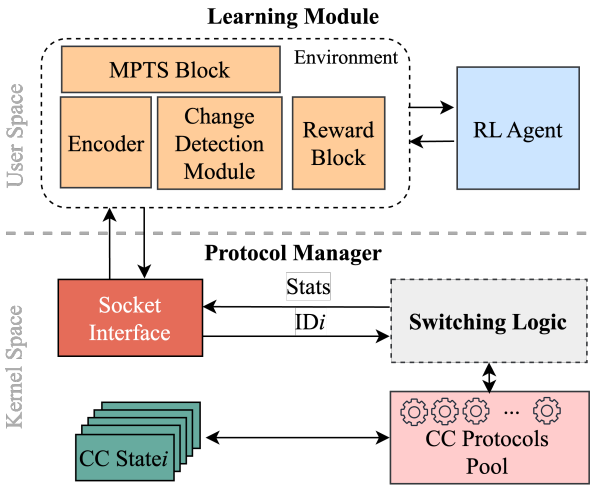


Figure 3: Overall architecture of *Mutant*. Two main components are a kernel module hosting a set of TCP protocols and a user-level module with the learning logic to decide when it is convenient to mutate into another protocol.

solution. Our results show that a subset of the best-performing policies for the specific network scenario leads to higher performance and faster convergence (see Section 6.2). We designed an algorithm that selects the top-*k* policies *per-flow* before running the online learning process (see Section 5.2). Secondly, Mutant runs a reinforcement learning procedure that can quickly learn *its own CC schema over time, based on the pool of existing protocols* (i.e., the CC heuristics in the kernel). By continuously adapting and exploring different protocols, Mutant can dynamically learn the strengths of each protocol. For instance, it might recognize that BBR2 excels in specific scenarios, such as when delays become particularly high or bandwidths change abruptly, while Cubic performs better in different conditions, for example, when packet loss events occur. In such a way, our solution can learn to adapt and refine its own congestion control policy over time, continually assessing and adjusting its strategies based on the evolving network conditions and performance metrics.

3 Design and Implementation

This section details the two main components overviewed in the previous section. First, we highlight the main challenges that originate from the kernel components and operations of the Protocol Manager. Then, we describe the Learning Module with the RL-based algorithm.

3.1 Protocol Manager

Our Protocol Manager seamlessly integrates in-kernel functionalities with the Linux kernel's TCP source code. The Mutant kernel module acts as a versatile meta-protocol for congestion control. The kernel utilizes a modular and extensible framework for implementing TCP congestion control

algorithms, with the struct `tcp_congestion_ops` at its core. This structure consists of function call pointers that define an interface for congestion control algorithms. Each function can be seen as a handler for a specific TCP event. Examples of events can be congestion avoidance (i.e., a congestion event has been triggered), receipt of a TCP ACK, or the adjustment of the TCP sender's state. Each function is designed to promptly respond to these events by executing the necessary actions that facilitate congestion management and update the internal state variables of the CC scheme. Each congestion control algorithm encapsulates its functionalities within this structure, enabling effortless pluggability into the system.

In our implementation of Mutant, we have created an independent and flexible *meta-protocol*. This module acts as a wrapper for the `tcp_congestion_ops` of the protocols in the pool and can dynamically plug and unplug them for live switching. Our live switching procedure functions as follows: when the Protocol Manager receives a new CC scheme to select from the Learning Module, it performs two important operations. Firstly, it stores the current state of the active CC scheme, which can be restored if selected again later. Secondly, it plugs the `tcp_congestion_ops` of the selected CC scheme, seamlessly replacing the previous one. Each CC scheme's state variables are maintained over time, and they are frozen and unfrozen when the scheme is selected. Additionally, upon every switch, the next protocol starts from the `cwnd` value of the preceding one. To communicate with the RL module in the user space, the protocol manager uses the *netlink* socket family. After receiving every ACK, it sends the network features to the Learning Module, which predicts the CC scheme to be selected for the next step.

3.2 Learning Module

This module hosts the online learning capability using a Reinforcement Learning (RL) approach. In RL, an *agent* performs an action, a, in a particular state, s, of an *environment* and receives a reward, r, for having performed this action [46]. An *agent* learns a *policy* to maximize its reward over time, where such a reward value indicates how successful the agent's action has been. This motivates the agent to seek for a series of state-action pairs that eventually maximizes its overall reward via a process of exploration and exploitation, with varying degrees.

In this work, we design an online learning algorithm based on contextual Multi-Arm Bandits (MAB) [36]. Similar to the more popular *Q-Learning*, *MAB* is an RL algorithm that makes decisions under uncertainty by balancing between exploration and exploitation of several actions, known as *bandits*. Simply, the algorithm observes a context, makes a decision by selecting one action from a set of alternative actions, and then observes the outcome of that decision. However, unlike traditional RL, bandit problems only observe the outcome of a selected action for a given state, and it does not have

Table 1: Raw input statistics used in the learning process. For each feature in **bold**, we also compute the mean, minimum, and maximum values across three observation windows of different sizes: *short* (10), *medium* (200), and *long* (1000).

`snd_cwnd`	Sender's congestion window size
`rtt_us`	Round-trip time in microseconds
`srtt_us`	Smoothed round-trip time
`mdev_us`	Mean deviation of round-trip time
`min_rtt`	Minimum round-trip time
`advmss`	Advertised maximum segment size
`delivered`	Number of packets delivered
`lost_out`	Rate of packets lost
`in_flight`	Number of packets sent
`retrans_out`	Number of retransmitted packets
`rate`	Delivery rate
`prev_proto`	ID of the previous protocol
`crt_proto`	ID of the selected protocol
`throughput`	Throughput
`loss_rate`	Loss rate

Q-tables to look up past performances and decide the best action to take [36]. By being simple in the learning nature, this approach can be used to learn *online* among possible choices and, thus, leads to faster adaptability to new network conditions.

Environment. It acts as the interface through which the agent interacts with its surroundings, offering feedback to the agent's actions and shaping its learning trajectory. The environment runs in our *Learning Module*, which collects the network parameters from the kernel at runtime and provides the observation at each time step. The duration of this time interval is customizable, but it impacts the switching frequency between different congestion control (CC) schemes within the pool. We explored various time intervals through empirical experimentation to understand their impact on model performance in Section 6.1.

Agent. The agent interacts with the Mutant environment by observing states, taking actions, and receiving rewards. The agent aims to learn the best mapping between states and actions to maximize cumulative rewards over time. In contextual MAB, the agent faces a set of actions (*arms*) and must choose the next action based on observed contexts or features. As each action yields an uncertain reward, the agent aims to learn which action maximizes cumulative rewards given different contexts. In multi-arm bandits algorithms, the agent employs a policy to balance the trade-off between exploration and exploitation. In Mutant, we implement the Upper Confidence Bound [32] as the exploration-exploitation policy (see Section 5.1).

State. In our work, we consider a diverse set of network statistics from the kernel during the communication between two hosts as our input features (Table 1). For a subset of these features, we additionally consider three observation windows to extract temporal dynamics varying the time granularity

across *short* (10 samples), *medium* (200 samples), and *long* (1000 samples) durations. This leads to a total of 55 input features collected at each step from the kernel. To efficiently manage this input space, we employ a pre-trained encoder to map these features in the latent space to achieve dimensionality reduction (see Section 5.1), resulting in a total of 16 embedding input signals.

Actions. The arm or action determines what an agent observing an environment can do to influence the environment. In our context, *Mutant* has the following set of actions represented as $A = tcp\_proto_i \; \forall \; i$, where i is the index of the CC scheme deployed on the host machine. According to the learning agent, Mutant selects the next scheme based on the network conditions it has seen. For every network scenario and before the agent starts learning, we run our MPTS protocol to determine the pool of *top-k* schemes, given a fixed k. However, k is a parameter of *Mutant*, and we designed our solution with the aim of providing the flexibility of including other (future) protocols in the kernel thanks to an extensible and versatile interface [1]. Our prototype tested in this paper supports 11 different in-kernel CC schemes: Cubic [25], Hybla [14], BBR2 [15], Westwood [17], Veno [24], Vegas [13], YeAH [9], Bic [53], HTCP [33], Highspeed [23] and Illinois [35]. Selecting the best scheme at each time interval is a non-trivial problem. We discuss such a problem and solution in Section 4.

Reward. A reward is a utility function that allows us to calculate the effectiveness of the action by measuring a reward value for an agent observing an environment. In our context, we want our agents to maximize the network throughput while minimizing the delay. We express this objective as an attempt to maximize the Power metric, a well-known quantity defined as $pw = \frac{Throughput}{Delay}$. At the same time, we also aim to minimize the number of packet losses. As such, our agents seek to maximize their overall reward at time step t, R_t defined as:

$$R_t = \frac{(thr_t - \zeta \cdot l_t)^{\kappa}}{d_t}, \tag{1}$$

where thr_t, d_t, l_t are the delivery rate, average delay, and loss rate observed at timestep t. ζ and κ are coefficients that determine the impact of the loss rate over the throughput and the importance of the throughput over the delay, respectively. We compute the throughput as the delivery rate during the communication directly in the kernel, defined as $thr_t = \frac{r_t}{s_t}$, where r_t is the number of packets delivered and s_t the time interval in *us*.

Algorithm 1: Mutant's Online Learning Algorithm

Let δ represent the step duration
Let N denote the total number of steps
Let x_t denote the raw network features
Compute top-k CC schemes via **MPTS (Algorithm 2)**.
Initialize thr_{max} and del_{min}
for *step in n_steps* **do**
    Predict id_i
    Switch current scheme to id_i
    while δ *has not expired* **do**
        Read x_t from kernel module
        Compute the embeddings $z_t = f_{\text{enc}}(x_t)$
        Update $state$ with z_t
        Update thr_{max} and del_{min}
    Compute $\mu_a = \theta^{*T} z_t$
    Compute $UCB_a = \mu_a + \sqrt{\frac{2\log(t)}{n_a}}$
    Update MAB *policy* parameters
$R_{max} = thr_{max}/del_{min}$
$R_t = \frac{(thr_t - \zeta \cdot l_t)^{\kappa}}{d_t} \cdot \frac{1}{R_{max}}$

4 Problem Statement: Selecting the Best Protocols to Learn From

A cooperative game is characterized by a pair (N, ν) containing a set of players $N = \{p_1, ..., p_n\}$ and a value function $\nu : \mathcal{P}(N) \to \mathbb{R}$, where $\nu(\emptyset) = 0$ by definition. The players can form coalitions $S \subseteq N$ and obtain a combined benefit given by $\nu(S)$, which is called the worth of S.

Our problem is characterized by a set of strategies, or arms $\mathcal{A} = \{a_1, ..., a_n\}$ and at each discrete time step t, the learner can *pull* an arm a_i. To each arm i there is an associated probability distribution $p_i \in [0, 1]$. The observed reward for arm i is drawn from p_i. We model our congestion control protocol set selection in Mutant as a multi-armed problem.

In particular, the resolution of the top-k arms identification problem determines the protocols to use. The Mutant agent selects the k arms with the highest mean rewards to form a coalition at the end of T evaluations. In the multi-arm bandit literature, the parameter T is denoted *fixed budget*. Such budget is given beforehand, and once exhausted, the learner returns its guess about the top-k arms. The learner's performance is hence measured by the probability of returning a correct output. Formally, the objective of the learner is to select the set $\{a_1, ..., a_k\}$ corresponding to the set of arms with the k highest mean rewards, $\mu_1, ..., \mu_k$. In the rest of this section, we assume that the final set of arms is ordered by the reward, i.e., $\mu_1 > ... > \mu_k$. This ordering assumption preserves the generality, while the assumption that the means are all distinct is made for the sake of notation (the complexity measures differ slightly if the top k means are ambiguous).

We assess the effectiveness of the Mutant agent's strategy by using the probability of misidentification, i.e.:

$$e_T = \mathbb{P}(\{a_1, ..., a_k\} \neq \{1, ..., k\}). \tag{2}$$

Considering the following complexity measure for the single

[1] Mutant code is publicly available at https://github.com/lorepap/mutant

best arm identification:

$$H_1 = \sum_{i=0}^{n} \frac{1}{\Delta_i^2} \quad \text{and} \quad H_2 = \max_i \frac{i}{\Delta_i^2}, \tag{3}$$

where $\Delta_i = \mu_1 - \mu_i$ for $i \neq 1$ and $\Delta_1 = \mu_1 - \mu_2$ (difference in mean rewards). Prior work [7] showed that these two complexity measures are equivalent up to a log factor, i.e.,

$$H_2 \leq H_1 \leq \log(2n) H_2. \tag{4}$$

Intuitively, H_1 represents a lower bound on the number of evaluations necessary to identify the best arm. To verify if an arm has mean $\mu^* = \max_i \mu_i$ or μ_i, and agent needs to sample $\frac{1}{\Delta_i^2}$ times. The surprising fact shown in [7], that we reuse in our analytical result, is that the order of H_1 evaluations suffices to identify the best arm. As for upper bounds, the quantity H_2 proved to be a useful surrogate for H_1 to express the bounds on e_T.

To model our k-best arms identification problem, we then define the gaps and the complexity measures as follows:

$$\Delta_i^k = \begin{cases} \mu_i - \mu_{k+1} & \text{if } i \leq k \\ \mu_k - \mu_i & \text{otherwise} \end{cases}$$
$$H_1^k = \sum_{i=0}^{n} \frac{1}{(\Delta_i^k)^2}, \quad H_2^k = \max_i \frac{i}{(\Delta_{(i)}^k)^2}, \tag{5}$$

where the notation $(i) \in \{1, .., n\}$ is defined such that $\Delta_1^k \leq ... \leq \Delta_m^k$. In our case, the H_1^k replaces the H_1 as the lower bound of the k-best arms identification problem.

5 Mutant Online Learning Logic

After the team selection process, we oversee the ultimate lineup and dynamically choose the protocol responsible for controlling congestion. We formulate the online learning procedure in Mutant (Algorithm 1) - which changes the CC scheme in the pool based on the real-time collection of network features - as a Contextual Multi-Armed bandit problem (CMAB) [36]. Unlike the traditional multi-armed bandit problem, where the rewards of each arm are independent of the context, in CMAB, the reward of each arm (i.e., the CC schemes) depends not only on the arm chosen but also on a set of contextual features (i.e., the network parameters) associated with the current state of the environment. Formally, at each time step t, the agent observes a context vector $\mathbf{x}_t$ that describes the current state of the environment and selects an arm i based on a policy $\pi(\mathbf{x}_t)$. The policy maps each context vector to a probability distribution over the arms, indicating the likelihood of selecting each arm given the context. The agent then receives a reward $r_{t,i}$ associated with the chosen arm.

To address this trade-off problem, we leveraged the Linear Upper-Confidence Bound policy, which we combine with an algorithm to preliminary select the policies in the pool (Section 5.2). We design our algorithm as a cooperative game. Unlike competitive games, where players compete to maximize an often selfish utility, cooperative games foster teamwork and coordination among participants [41]. In our system, all players (protocols) aim to reach "high throughput and low delay" for all flows. An intriguing question in the context of cooperative games concerns the importance of a single player's contribution. In particular, one crucial question is: What are the top players performing in this game?

5.1 Challenges to Achieve an Online Learner

Exploration vs. Exploitation Trade-Off. One main challenge in online RL is balancing the trade-off between exploration, i.e., trying out new actions or strategies to gather information about the environment, and exploitation, i.e., leveraging known information to maximize short-term rewards. While too much exploration may lead to sub-optimal performance, excessive exploitation can result in missed opportunities for discovering better solutions, given the uncertainty surrounding the efficacy of the chosen protocol and the varying network conditions. This difficulty arises because achieving quick and cost-effective convergence becomes impractical without investing significant time and resources. As an online RL approach, our model must minimize the exploration time and ensure consistently good performance by selecting the best-performing protocol for the underlying network settings.

The key challenge in the CMAB problem is to learn an effective policy that balances the exploration of different arms with the exploitation of arms that are likely to yield high rewards, considering the contextual information. To this end, we use the LinUCB (Linear Upper Confidence Bound) algorithm [34], which extends the classic Upper Confidence Bound algorithm [8] to the contextual setting.

LinUCB assumes a linear relationship between the expected reward of an arm and associated contextual information. Specifically, the expected reward for pulling arm a at round t, given a context feature vector x_t with dimensionality d, can be expressed as $\mu_a = \theta^{*T} x_t$, where θ^{*T} captures the weights associated with each feature. However, since we do not have direct access to θ^*, LinUCB employs an estimate $\hat{\theta}_t \in \mathbb{R}^d$ that is updated over time based on observations. Our approach entails Bayesian linear regression [12] to model such a relationship between the contextual features and the expected rewards for each action, thus forecasting the weights θ^{*T}. This linear relaxation simplifies the traditional RL process, making the learning policy a more lightweight process suitable for our purpose of learning online.

In addition to predicting the reward, LinUCB computes an upper confidence bound (UCB) for each action's expected reward. This bound represents a trade-off between the estimated mean reward and the uncertainty in that estimate. In conclusion, by selecting actions with higher UCB values, LinUCB can balance exploration (selecting actions with uncertain re-

wards) and exploitation (selecting actions with potentially high rewards), aiming to maximize cumulative rewards over time.

Reduce the Input Representation. From the raw network features indicated in Table 1, we compute maximum, minimum, and mean values for some metrics over short, medium, and long-size sliding windows. While this approach allows to extract temporal dynamics at runtime, it also leads to a total of 55 different network features. Such a large state space is hardly managed by LinUCB, which approximate a linear relationship between the context (i.e., network features) and the reward of each arm (i.e., the protocols in the pool). If, on the one hand, such a linear algorithm enables a more lightweight policy, on the other hand, this simplistic assumption poses new challenges as it needs more representational power, especially with a large context vector.

Inspired by [40], to overcome this limitation, we design an *encoder* network to represent a large network feature vector into a smaller latent space. On top of that, this operation should be done in an online fashion. At each step, the high-dimensional raw observation x_t is processed by a pre-trained encoder that outputs the low-dimensional embeddings z_t. We use a Gated-Recurrent Unit [18] layer with two fully connected (FC) layers as our encoding network architecture to capture the sequence-level information from the online network data. Thus, the expected reward for each arm becomes $\mu_a = \theta^{*T} z_t$, where z_t is the output of the last hidden layer of the encoding network for the context x_t.

Dynamic Reward Normalization. Since the reward defined in Equation 1 captures the ratio between the throughput and the round-trip time, as the bandwidth can change over time, the most-rewarded protocol can change accordingly. Because of this, we need to normalize the reward according to the maximum achievable reward for the particular network setting. We adopt an online change detection algorithm based on ADWIN [11]. This algorithm can detect the throughput and the round-trip time shifts during the communication for each selected CC scheme. Every time a change is detected, we recalculate the maximum reward by replacing the throughput and round-trip time values in Equation 1 with each adaptive window's maximum throughput and minimum round-trip time. This approach ensures that the normalized reward reflects the best-performing CC scheme at each step, even when the underlying network setting changes.

5.2 Top-*k* selection with MPTS

We present an algorithm — *Mutant Protocol Team Selection (MPTS)* – to find the best CC algorithms to learn from, and we show that such an algorithm has a bounded error. Before proceeding, we introduce the following notation. For each arm i and for all rounds $t \geq 1$, we have $W_i(t) = \sum_{s=1}^{t} \mathbb{1}_{I_t=1}$ to be the number of times arm i was pulled from rounds 1 to t, and $X_{i,1}, X_{i,2}, ..., X_{i,W_i(t)}$ the sequence of associated rewards. Then,

Algorithm 2: Mutant Protocol Team Selection (MPTS) algorithm for k-best protocol identification.

Let $\mathcal{A}_1 = \{a_1, ..., a_n\}, k(1) = k, n_0 = 0$, $\overline{\log}(T) = \frac{1}{2} + \sum_{i=2}^{T} \frac{1}{i}$, and for $j \in \{1, ..., n-1\}$ $n_j = [\frac{1}{\overline{\log}(T)} \frac{T-n}{n+1-j}]$;

Initialize ;

for *each phase* $j \in \{1, ..., n-1\}$ **do**

1. For each active arm $a_i \in \mathcal{A}_j$ select arm a_i for $n_j - n_{j-1}$;
2. Let $\sigma_j : \{1, ..., n+1-j\} \to \mathcal{A}_j$ be the bijection that orders the empirical means by $\hat{\mu}_{\sigma_j(1),n_j} \geq \hat{\mu}_{\sigma_j(2),n_j} \geq ... \geq \hat{\mu}_{\sigma_j(n+1-j),n_j}$. For $1 \leq r \leq n+1-j$, we have these empirical gaps $\hat{\Delta}_{\sigma_j(r),n_j} =$ $\begin{cases} \hat{\mu}_{\sigma_j(r),n_j} - \hat{\mu}_{\sigma_j(k(j)+1),n_j} & \text{if } r \leq k(j) \\ \hat{\mu}_{\sigma_j(k(j)),n_j} - \hat{\mu}_{\sigma_j(r),n_j} & \text{if } r \geq k(j)+1 \end{cases}$;
3. Let $i_j \in argmax \hat{\Delta}_{i,n_j}$ and ties are broken arbitrarily. Deactivate arm a_{i_j} and $\mathcal{A}_{j+1} = \mathcal{A}_j \backslash \{a_{i_j}\}$;
4. If $\hat{\mu}_{i_j,n_j} > \hat{\mu}_{\sigma_j(k(j)+1),n_j}$, then arm a_{i_j} is accepted, i.e., $k(j+1) = k(j) - 1$ and $a_{k-k(j+1)} = i_j$;

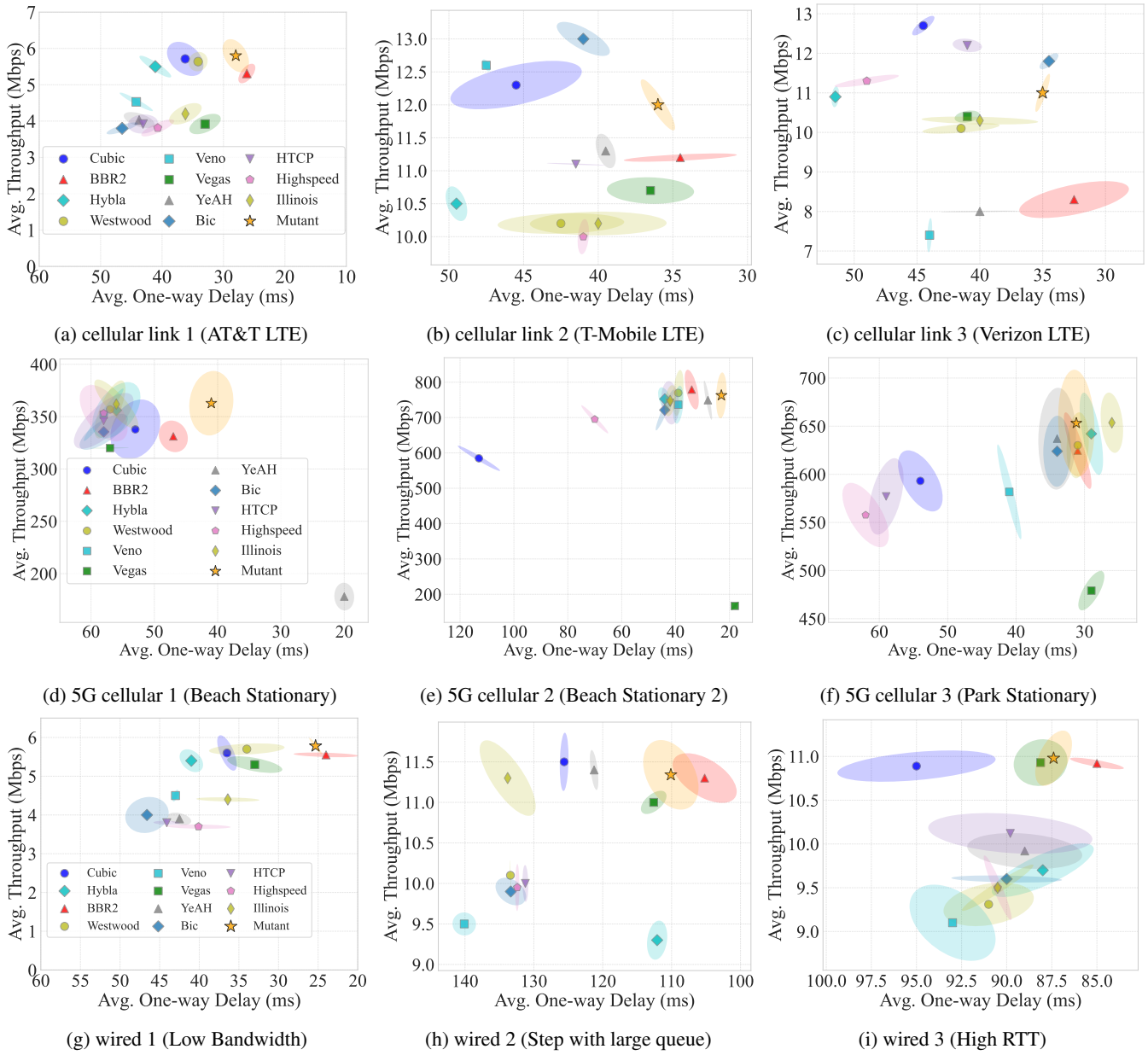
return the k accepted arms $a_1, ..., a_k$;

let $\hat{\mu}_{i,s} = \frac{1}{s} \sum_{t=1}^{s} X_{i,t}$ the empirical mean of arm i after s evaluations and $X_{i,s}(k)$ and $\hat{\mu}_{i,s}(k)$ the corresponding quantities in the multi-bandit problem.

The concept behind MPTS is similar to *Successive Rejects*, which was developed to address the single best arm identification problem [7]. However, MPTS has the additional feature of occasionally accepting an arm if it can be conclusively shown to be one of the top k arms.

Informally MPTS evolves following these steps. The time (the T rounds) is first divided into $K-1$ phases. At the end of each phase, the algorithm either accepts the protocol (arm) with the highest empirical mean or dismisses the protocol with the lowest empirical mean. In any of these two cases, the corresponding protocol is deactivated. During the following phase, it pulls each active protocol with the same frequency. During a specific phase j, the key to choosing whether to accept or reject is to rely on estimates for the gaps Δ_i^k.

Assuming, for example, that the algorithm has already accepted $k - k(j)$ protocols, $a_1, ..., a_{k-k(j)}$, it means that $k(j)$ must be still found. At the end of the phase j, MPTS computes for the $k(j)$ empirical best arms (among the active arms) the distance in terms of empirical mean to the $(k(j)+1)^{th}$ empirical best arm among the active arms. For the active arms not among the $k(j)$ empirical best arms, instead, MPTS computes the distance to the $k(j)^{th}$ empirical best arm. MPTS deactivates the arm i_j that maximizes these empirical distances. If i_j is the currently best (empirically) arm, it is accepted and



(a) cellular link 1 (AT&T LTE)

(b) cellular link 2 (T-Mobile LTE)

(c) cellular link 3 (Verizon LTE)

(d) 5G cellular 1 (Beach Stationary)

(e) 5G cellular 2 (Beach Stationary 2)

(f) 5G cellular 3 (Park Stationary)

(g) wired 1 (Low Bandwidth)

(h) wired 2 (Step with large queue)

(i) wired 3 (High RTT)

Figure 4: Average throughput and one-way delay over different emulated network environments.

variables are updated as $k(j+1) = k(j) - 1$, $a_{k-k(j+1)} = i_j$. Otherwise, the arm i_j is rejected. The formal description of MPTS is reported in Algorithm 2. We now show that the error probability of MPTS is bounded, with the following Theorem.

Theorem 5.1. *Let n be the number of available protocols, T the number of evaluations, and H_2^k the complexity measure defined in Equation 5. The error probability of the MPTS algorithm in the k-best protocol identification problem has the following error bound:*

$$e_T \leq 2n^2 \exp\left(-\frac{T-n}{8\overline{\log}(T)H_2^k}\right). \tag{6}$$

Proof. See appendix. □

6 Performance Evaluation

In what follows, we describe Mutant general evaluation using simulated and real-world traffic. In particular, in Section 6.1, we give details about the environments used for our experiments. In Section 6.2, we present Mutant's performance and compare them to state-of-art ML-based schemes. In Section 6.3, we analyze the importance of selecting the best-k protocols in the pool. We elaborate more about Mutant's fairness in Section 6.4. Finally, in Section 6.5 we present an

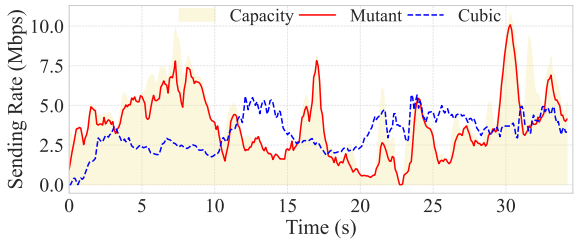


Figure 5: Mutant quickly adapts to highly variable network conditions compared to Cubic, the default congestion control in most Linux implementations. An optimal protocol would follow exactly the available capacity.

ablation study on Mutant's main components.

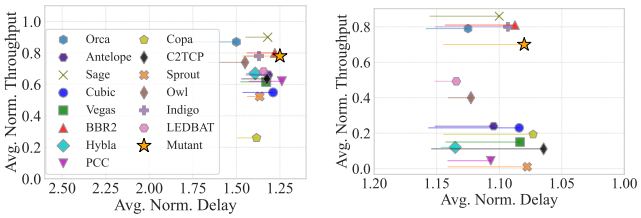
6.1 Evaluation Settings

Across our experiments, we test our protocol on publicly available realistic network conditions [2, 6, 37] to emulate *cellular* links even high bandwidths. Additionally, we consider a combination of emulated *wired* network scenarios, i.e., a typical Internet network environment consisting of a fixed bandwidth, delay, and queue size. We range the bandwidth, minimum rtt, and buffer queue size values within [12, 96] Mbps, [10, 120] ms, and [1, 20] $\times$ BDP, respectively.

We collect the performance of other CC baselines using Pantheon testbed [54] and Mahimahi emulator [37]. An experiment involves establishing an `iperf3` session between a single client and a Linux-based host machine. This setup allows us to capture and analyze various network metrics and evaluate the performance of different congestion control algorithms in a controlled experimental environment. For our real-world scenarios, we test Mutant on *(i)* intra-continental and *(i)* inter-continental network environments using Fabric [10] testbed. In our experimental setup, we establish the default quantity of kernel protocols within the pool, denoted as k, to be 6. This decision followed a comprehensive evaluation via a greedy search approach, as detailed in Section 6.3. Additionally, we set $\delta = 10^{-}2$s, i.e., the interval for alternation among CC schemes, (see Section 6.5).

6.2 Throughput-Delay Performance

In our experiments we compared Mutant with the following baseline, divided into four categories: (i) end-to-end TCP designs, e.g., Cubic [25], Vegas [13], Hybla [14] and the schemes in the pool, PCC [21], Copa [5]; (ii) end-to-end cellular, i.e., LTE protocols, e.g., C2TCP [1], Sprout [51]; (iii) ML-based protocols, such as Owl [43], Indigo [54], Orca [2], Sage [55], Antelope [56], and (iv) mixed schemes, e.g., LEDBAT [45]. Figure 4 shows the average throughput and delay across the cellular and wired sets of experiments, where the shaded ellipses represent the standard deviation across experiments from the mean value (i.e., the center of the ellipses). In particular, the *step* scenario (Figure 4h) is obtained by doubling the bandwidth after a defined time period. We compare Mutant against the pool of schemes it learns from. We select different emulated network environments for wired and cellular experiments and run each protocol for 2 minutes. We selected a diverse set of network scenarios with different ranges of bandwidth, from low (i.e., LTE and wired), to high[2] (i.e., 5G cellular). For each environment, we preliminary compute the protocols in the pool by running MPTS with $T = 100$, then running Mutant with the resulting pool. Results prove that Mutant is consistently among the best protocols across all tested scenarios and learns how to adapt to the specific environment. Somehow surprisingly, these findings show that Mutant does not depend on the best-performing protocol but can learn its policy by switching among the protocols pool and adapting quickly to the network scenario to maximize its reward. Indeed, Mutant can learn from the pool of protocols and outperform them. In Figure 5, we present how Mutant's best-k setting can quickly adapt to changing network conditions, while Cubic clearly shows the inability to achieve full link utilization. This shows that Mutant learns to select the protocols based on when they perform at their best over time, even in highly variable network scenarios. We explore further the reward suboptimality in Appendix B.



(a) Intra-Continental (b) Inter-Continental

Figure 6: Average normalized throughput, average normalized delay and 95^{th} percentile (end of lines) on real-world experiments.

We then validate Mutant on real-world network environments. We consider intra-continental and inter-continental experiments by setting up different hosts across the US and Europe using Fabric testbed [10]. To make a fair comparison with the ML baselines, we preliminary run the MPTS algorithm for the top-k protocols on every scenario. For each baseline, we run one flow for 30 seconds and repeat the experiment 10 times. As demonstrated in Figure 6, Mutant consistently achieves high performance levels when compared to other baselines. Most notably, Mutant can surprisingly match or even outperform pre-trained ML-based congestion control protocols (such as Orca, Sage, Antelope, and Indigo), even without extensive offline training. Mutant can also outperform online learning schemes, such as PCC, which show a competitive delay but a low throughput. We also highlight

[2]Due to Mahimahi limitations on traces with bandwidths larger than $\sim$ 200 Mbps, we leveraged a modified version of Mahimahi [6] to test Mutant on real 5G cellular traces with high bandwidth.

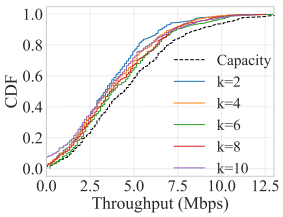


Figure 7: CDF of the throughput over a cellular link varying k.

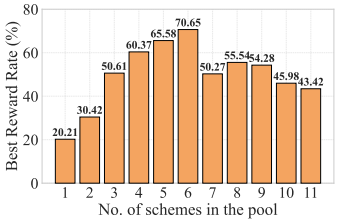


Figure 8: Mutant's performance varying the number of protocols in the pool (k).

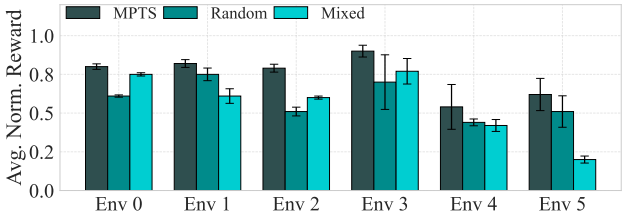


Figure 9: Impact of different protocol selection strategies.

that Mutant is capable of achieving competitive performance even within a short duration of the flow transmission. Overall, Mutant shows a 3.85% lower delay than BBR2 (appearing as the best-performing protocol in the pool) and a 3.60% lower delay than the average delay of other machine learning-based solutions like Sage, Orca, Indigo, and Antelope. Notably, it can achieve this result while maintaining a high throughput. This analysis yields an important result: *it is possible to learn from a pool of protocols in an online fashion and achieve competitive performance compared to offline ML algorithms.*

6.3 MPTS: Top-k Evaluation

To identify the optimal k for our Mutant Congestion Control protocol, we employ our MPTS Algorithm 2 while varying the number of protocol k in the pool, which also defines the action space in the RL model. Empirically, we observe that the inclusion or exclusion of certain protocols significantly influences the performance of the model, leading to either performance improvement or deterioration. Consequently, it is crucial to determine the best-performing configuration, where the k-size "team" in the algorithm effectively represents the protocols that positively contribute to the overall performance of Mutant.

We run an experiment to empirically find the value of k to be set for each network scenario. We run MPTS for a total of $T = 100$ rounds, varying k between 2 and 11. During each round of the MPTS algorithm, the available protocols are selected based on a uniform distribution. This selection process ensures that each protocol has an equal chance of being evaluated. Upon completing the T total rounds of the MPTS algorithm, we compute the best k protocols based on their cumulative mean reward, as defined in Equation 1. Intuitively, the higher the number of protocols in the pool, the more time Mutant must explore to find the best protocol at a given step of the algorithm. We evaluate how the number of protocols in the pool impacts the performance of Mutant on a cellular link in terms of throughput over time, reporting results in Figure 7. While it benefits the exploration of more protocols, we found that with more than $k = 8$, performance degrades without improving, even with more time to explore.

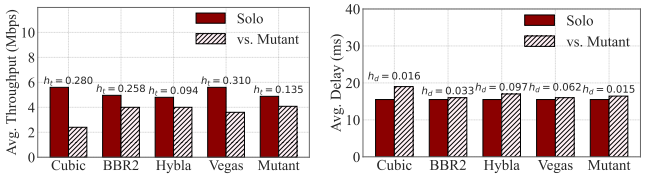
We then test each resulting configuration across all network scenarios and collect the average throughput and delay. We present the overall performance in Figure 8 in terms of *best reward rate*, namely the number of times Mutant results in the top 10% of the best-performing protocols for a given network environment.

Finally, to underline the importance of selecting the best-k protocol preliminary (via MPTS), we run the following experiment: we consider six different network environments sampled from the sets in Section 6.1, and three protocol selection strategies: (i) MPTS with $k = 6$, (ii) a random selection of $k = 6$ initial CC schemes and (iii) a mixed selection of CC schemes for each of the following categories: delay-based (Vegas, BBR2), loss-based (Cubic, Westwood), hybrid (Illinois) and specialized (Highspeed, Hybla). Subsequently, we run Mutant with the resulting pool of CC schemes for each selection strategy for 2 minutes on the six network scenarios. In Figure 9, we present the average normalized reward (Equation 1) for the three selection strategies. Interestingly, the selection of the initial pool of schemes significantly impacts the performance of our model. When a diverse set of CC schemes (i.e., protocols of different natures) is considered, Mutant shows a degradation of performance. The takeaway message is that *the efficacy of Mutant is significantly enhanced by an adequate selection of the starting protocols for the specific network environment, and our MPTS algorithm enables consistently high performance across the test scenarios.*

6.4 Fairness and Harm Analysis

In this section, we evaluate the *harm* [50] that a *Mutant* flow, α, causes another flow, β, considered as another congestion control algorithm, e.g., Cubic. As mentioned by Ware et al. [50], *harm* instead of *fairness* is a better performance metric. If the amount of harm caused by flows using a new algorithm γ on flows using an algorithm ω is within a bound derived from how much harm ω flows cause other ω flows, we can consider γ deployable alongside ω. Therefore, we use this metric to show that *Mutant* can be deployed to real-world systems.

Harm, unlike *fairness*, takes into account that different flows can have different demands. As such, for example, a new CC scheme γ competing against a *Reno* flow on a 10Gbps link would not be flagged as *unfair* because it took advantage of the remainder of the link capacity due to the well-known *Reno's* slow additive increase and aggressive reaction to loss [26], which prevents *Reno* from fully utilizing the link. Also, it is



(a) Throughput. (b) Delay.

Figure 10: Mutant's *harm* [50] to other protocols and *self-harm* (right-most bars) is consistently low, meaning that the protocol *fairness* and *friendliness* are high when competing with other protocols or against other Mutant instances.

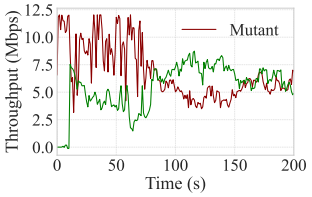


Figure 11: Friendliness competing with a Cubic flow.

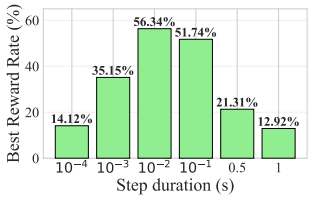


Figure 12: Impact of the step duration δ.

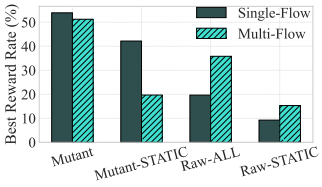


Figure 13: Impact of different groups of input features.

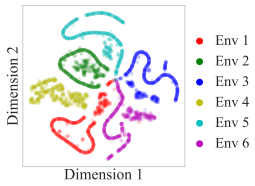


Figure 14: t-SNE [49] of the embeddings extracted from the last layer of the encoder.

multi-metric. It is not based solely on throughput unlike Jain's Fairness Index (JFI) [27]. On the contrary, it also considers performance metrics like latency, flow completion time, etc.

Much like *fairness*, *harm* ranges from [0 - 1] where 1 is maximally harmful and 0 is harmless. We use the following equations to calculate the *harm*, h, caused by *Mutant's* on another CC scheme. For metrics where "more is better" (like throughput), we use $h_t = \frac{x-y}{x}$, where x is the solo performance of β without α and y is the performance of β after the introduction of α. While for metrics where "less is better" (like latency or loss), we use $h_d = \frac{y-x}{y}$ to calculate *harm*.

We evaluated the harm of *Mutant* when competing with other flows. In Figure 10, we report the performance in terms of throughput and delay, where Mutant is used in competition with different CCAs. We also assess the protocol's self-harm. In particular, we tested some protocols in Mutant's pool on an emulated cellular link. First, we run each CC scheme individually in a single-flow scenario (i.e., without interference from Mutant's flow). Then, we run Mutant concurrently with a flow using a different CC scheme to observe their competition. Our results reveal that *Mutant* is friendly (i.e., causes minimal harm) to all the other protocols, with a negligible influence on the flows of other protocols, by just looking at the h_t and h_d values. From these experiments, we can see how, in addition to being friendly, Mutant also has high self-fairness i.e., it minimally harms other Mutant instances.

Additionally, we provide a sample of the Mutant's behavior when competing with a Cubic's flow in Figure 11 on a cellular link. It is worth noticing how Mutant learns to adapt to the new flow and converge to a fair share over time. This is due to the fact that even though Equation 1 does not explicitly include a fairness term, Mutant has the capability to penalize any unfair protocol behavior and promptly switch to an alternative one in response to a sudden surge in packet loss - which is accounted for in the formulation. Over time, Mutant converges to the fair share of the link capacity to maximize the reward.

6.5 Ablation Study

In this section, we validate some of the main Mutant's components and their impact on its overall behavior. First, as Mutant's switching logic is a core feature of the system, we study the impact of different values of δ (i.e., the switching frequency). Second, we evaluate the importance of input features in the online learning process. In particular, we validate the impact of mapping the network features into the latent space. Third, we study the encoder's ability to identify different network conditions.

Impact of switching frequency. We start analyzing the performance of different values of the step duration δ, i.e. the time interval a CC scheme in the pool runs before the next action of the RL algorithm is performed. We report in Figure 12 the *best reward rate* across all our network test environments - i.e., the number of times Mutant results in the top 10% of the best-performing protocols for a given environment. As emerging from the graph, *Mutant benefits from high-frequency transitions between schemes in the pool, instead of selecting a single protocol for extended duration.*

Input Features. To evaluate the effect of different input feature strategies, we run an experiment for single and multi-flow environments to examine the following input feature configurations: Mutant-STATIC, Raw-ALL, and Raw-STATIC. Mutant-STATIC excludes all mean, minimum, and maximum statistics from the input feature set. Raw-ALL removes the encoder from Mutant's workflow and directly feeds the network features into the learning model. Raw-STATIC eliminates both the encoder and the mean, minimum, and maximum statistics. Figure 13 exhibits that Mutant benefits from input features represented in the latent space. These results also reflect Mutant's behavior when only a partial set of features is input to the encoder: indeed, without any historical information given by the different observation windows, there is a clear performance degradation.

Quality of the embeddings. We present a t-Distributed Stochastic Neighbor Embedding (t-SNE) [49] analysis (a

popular technique employed to visualize high-dimensional data) to validate further the importance of encoding the input network features in Mutant. Figure 14 shows such a two-dimensional t-SNE representation of the embeddings in six random network scenarios sampled from the test set indicated in Section 6.1. This visualization highlights that our encoder sharply separates different network scenarios in the latent space, helping Mutant detect diversities among states.

7 Related Work

The problem of congestion control has been extensively discussed in the literature, given its importance for reliable data transmissions. While congestion control has been a popular research topic for over 30 years, new recent solutions have employed machine learning-based algorithms to overcome the limitations of traditional approaches. In the following, we describe how machine learning (ML) has been helpful and how we differ from these recent solutions.

Since TCP congestion control is a fundamental service, it is not surprising that significant improvements and variations have been proposed over the years. Among them, we can mention TCP Vegas [13], Fast [30], BBR [15], and Compound [47]. Several other protocols focus on data centers, e.g. Data Center TCP (DCTCP) [3], but those are not the focus of our approaches. Data centers have been recently departing from TPC-based solutions, moving towards service meshes. Past congestion control protocols have been classified according to the metric they consider. Delay-based protocols consider *rtt* and/or average delivery rate measurements to decide how fast data should be sent over the network. BBR [15], e.g., belongs to such class and is considered to be resistant to the bufferbloat problem, but it frequently exceeds the link capacity, causing excessive queuing delays. On the other hand, loss-based protocols rely on indications of lost packets, and most of the traditional approaches, e.g., Compound [47] and Cubic [25], rely on some predefined functions or rules to handle network conditions.

However, all these solutions are limited by their nature of fixed-rule strategies, and their performance is limited in networks that require rapid adaptations. To address this problem, a plethora of online learning congestion control protocols have appeared, e.g., Remy [52], Performance-oriented Congestion Control (PCC) [21], PCC-Vivace [22], Copa [5]. After defining an objective function to be optimized, the process consists of learning the best actions to take either periodically or upon receiving ACKs.

This learning process can be further optimized by the advances in ML, and in particular in reinforcement learning (RL). RL has permeated many congestion control mechanisms, such as Orca [2], Owl [43], SPINE [48], and Aurora [28]. While Aurora is built upon the previous PCC protocol and then extended with a Deep-RL approach, Owl and Orca extend the Cubic implementation by adding an RL learning model that can work even in unexplored conditions. SPINE, instead, follows the additive-increase/ multiplicative-decrease (AIMD) approach of Cubic, but the parameters of the increment and decrement are the output of the RL model. As these papers, our protocol also leverages past solutions to combine the classical approach with advanced DRL techniques, but with a novel twist. Prior learning solutions suffer from design suboptimalities. If, on the one hand, delay-based protocols cannot avoid starvation, an extreme form of unfairness, in paths with excessive delay variations [4], loss-based ones suffer the bufferbloat problems. To solve this tussle, we bank on the idea there is no one-size-fits-all protocol, as learned from decades of designing congestion control protocols. Our design is based on the notion of learning from existing protocols, and predicting which protocol (set) would perform best in future seen network and congestion scenarios.

The idea of switching among available CCs has been floated before [38, 55, 56]. With them, we share the idea of learning from existing protocols; however, we differ in several ways from these solutions. Firstly, Mutant is not reliant on a data-driven offline procedure, meaning that its learned congestion control policy can adjust to different network conditions over time. Secondly, Mutant can adapt quickly to changes in the network environment with minimal offline adaptations, as its main functions run online. Finally, Mutant can smoothly transition between different CC protocols in the kernel space, regardless of the flow duration, while preserving past TCP information and states for each protocol during the change.

8 Conclusion

In this paper, we presented the design, implementation, and evaluation of *Mutant*, a congestion control scheme that learns from a pool of existing congestion control algorithms online. *Mutant*'s online learning algorithm understands which current state-of-the-art congestion control algorithm can perform well in the near future. Our evaluation shows that our solution can adapt to different network congestion scenarios efficiently if it has the correct pool of CC schemes to learn from. Perhaps surprisingly, we found that more knowledge in terms of protocol choices may hurt performance, given the explorative nature of the protocol. Mutant obtained high throughput and low delays more consistently than our benchmark CC protocols across different network conditions. Additionally, we evaluated *Mutant* fairness by measuring the harm index. Our results confirmed that *Mutant* is fair and causes negligent harm and self-harm to other flows.

Acknowledgment

This work has been supported by NSF awards #2201536 and #2133407. We would like to thank Princewill Okorie for his initial work on this project, our shepherd Dr. Yasir Zaki, the anonymous reviewers, and Dr. Gianni Antichi for his feedback on a preliminary version of this manuscript.

References

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

[2] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. Classic meets modern: A pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '20)*, page 632–647, 2020.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '10)*, page 63–74, 2010.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '22)*, pages 177–192, 2022.

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

[6] Rohail Asim, Muhammad Khan, Luis Diez, Shiva Iyer, Ramon Aguero, Lakshmi Subramanian, and Yasir Zaki. ZEUS: An Experimental Toolkit for Evaluating Congestion Control Algorithms in 5G Environments. *arXiv preprint arXiv:2208.13985*, 2022.

[7] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best arm identification in multi-armed bandits. In *Proceedings of the 23rd Annual Conference on Learning Theory (COLT)*, pages 41–53, 2010.

[8] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[9] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. Yeah-tcp: yet another highspeed tcp. In *Proc. PFLDnet*, volume 7, pages 37–42, 2007.

[10] Ilya Baldin, Anita Nikolich, James Griffioen, Indermohan Inder S Monga, Kuang-Ching Wang, Tom Lehman, and Paul Ruth. FABRIC: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 23(6):38–47, 2019.

[11] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

[12] Christopher M Bishop. Pattern recognition and machine learning. *Springer google schola*, 2:645–678, 2006.

[13] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[14] Carlo Caini and Rosario Firrincieli. Tcp hybla: a tcp enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22(5):547–566, 2004.

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

[16] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. TCP BBR v2 Alpha/Preview Release. https://github.com/google/bbr/blob/v2alpha/README.md, 2019.

[17] Claudio Casetti, Mario Gerla, Saverio Mascolo, Medy Y Sanadidi, and Ren Wang. Tcp westwood: end-to-end congestion control for wired/wireless networks. *Wireless Networks*, 8:467–479, 2002.

[18] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[19] Shay B. Cohen, Gideon Dror, and Eytan Ruppin. Feature selection via coalitional game theory. *Neural Computation*, 19:1939–1961, 2007.

[20] Zihan Ding. Imitation learning. In Shanghang Zhang Hao Dong, Zihan Ding, editor, *Deep Reinforcement Learning: Fundamentals, Research, and Applications*, chapter 8, pages 273–306. Springer Nature, 2020.

[21] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, 2015.

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

[23] Sally Floyd. Highspeed tcp for large congestion windows. Technical report, 2003.

[24] Cheng Peng Fu and Soung C Liew. Tcp veno: Tcp enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications*, 21(2):216–228, 2003.

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

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

[27] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[28] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 3050–3059. PMLR, 2019.

[29] Huiling Jiang, Qing Li, Yong Jiang, GengBiao Shen, Richard Sinnott, Chen Tian, and Mingwei Xu. When machine learning meets congestion control: A survey and comparison. *Computer Networks*, 192:108033, 2021.

[30] Cheng Jin, David X Wei, and Steven H Low. FAST TCP: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4. IEEE, 2004.

[31] Mehdi Kargar and Aijun An. Discovering Top-k Teams of Experts with/without a Leader in Social Networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM '11)*, pages 985–994. ACM, 2011.

[32] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On bayesian upper confidence bounds for bandit problems. In *Artificial intelligence and statistics*, pages 592–600. PMLR, 2012.

[33] Douglas Leith. H-TCP protocol for High-speed Long Distance Networks. In *Proc. International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet '04)*, 2004.

[34] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670, 2010.

[35] Shao Liu, Tamer Başar, and Ravi Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools*, pages 55–es, 2006.

[36] Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. PMLR, 2010.

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

[38] Xiaohui Nie, Youjian Zhao, Zhihan Li, Guo Chen, Kaixin Sui, Jiyang Zhang, Zijie Ye, and Dan Pei. Dynamic TCP Initial Windows and Congestion Control Schemes Through Reinforcement Learning. *IEEE Journal on Selected Areas in Communications*, 37(6):1231–1247, 2019.

[39] Zhiyuan Pan, Jianer Zhou, XinYi Qiu, Weichao Li, Heng Pan, and Wei Zhang. Marten: A built-in security drl-based congestion control framework by polishing the expert. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 2023.

[40] Carlos Riquelme, George Tucker, and Jasper Snoek. Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127*, 2018.

[41] Walid Saad, Zhu Han, Mérouane Debbah, Are Hjorungnes, and Tamer Basar. Coalitional game theory for communication networks. *IEEE signal processing magazine*, 26(5):77–97, 2009.

[42] Alessio Sacco, Matteo Flocco, Flavio Esposito, and Guido Marchetto. Partially Oblivious Congestion Control for the Internet via Reinforcement Learning. *IEEE Transactions on Network and Service Management*, 20(2):1644–1659, 2022.

[43] Alessio Sacco, Flocco Matteo, Flavio Esposito, and Guido Marchetto. Owl: Congestion control with partially invisible networks via reinforcement learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[44] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[45] Stanislav Shalunov, Greg Hazel, Jana Iyengar, and Mirja Kühlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817, December 2012.

[46] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[47] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12. IEEE, 2006.

[48] Han Tian, Xudong Liao, Chaoliang Zeng, Junxue Zhang, and Kai Chen. Spine: an efficient drl-based congestion control with ultra-low overhead. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22)*, pages 261–275. ACM, 2022.

[49] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[50] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*, pages 17–24. ACM, 2019.

[51] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of the 10th USENIX NSDI*, page 459–472, USA, 2013.

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

[53] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *IEEE INFOCOM 2004*, volume 4, pages 2514–2524. IEEE, 2004.

[54] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-Control Research. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 731–743, USA, 2018. USENIX Association.

[55] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. Computers Can Learn from the Heuristic Designs and Master Internet Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 255–274. ACM, 2023.

[56] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. Antelope: A framework for dynamic selection of congestion control algorithms. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

Appendix

A MPTS Proof

Proof Sketch of Theorem 5.1.

Proof. Let's consider the event ξ defined as:

$$\xi = \{\forall i \in \{1,...,n\}, j \in \{1,...,n-1\}, \left|\frac{1}{n_j}\sum_{s=1}^{n_j} X_{i,s} - \mu_i\right| \leq \frac{1}{4}\Delta^i_{(n+1-j)}\}; \quad (7)$$

by Hoeffding's inequality and a union bound, the probability of the complementary event $\overline{\xi}$ is bounded by inequality:

$$\begin{aligned}\mathbb{P}(\overline{\xi}) &\leq \sum_{i=1}^{n}\sum_{j=1}^{n-1}\mathbb{P}\left(\left|\frac{1}{n_j}\sum_{s=1}^{n_j} X_{i,s} - \mu_i\right| > \frac{1}{4}\Delta^i_{(n+1-j)}\right) \\ &\leq \sum_{i=1}^{n}\sum_{j=1}^{n-1} 2\exp\left(-2n_j\left(\frac{\Delta^i_{(n+1-j)}}{4}\right)^2\right) \\ &\leq 2n^2\exp\left(-\frac{T-n}{8\overline{\log}(T)H_2^k}\right).\end{aligned} \quad (8)$$

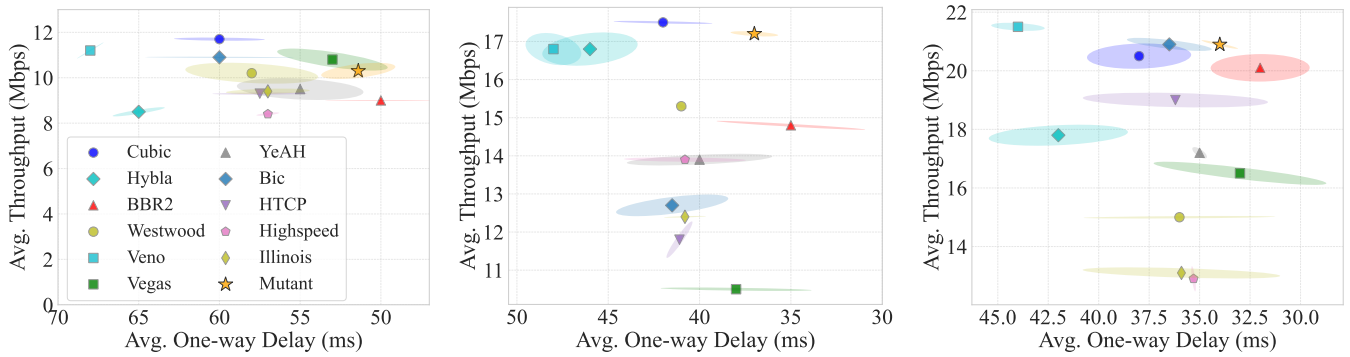
The last inequality is a direct conclusion from the fact that:

$$\begin{aligned}n_j\left(\Delta^i_{(n+1-j)}\right)^2 &\geq \frac{T-n}{\overline{\log}(T)(T+1-j)\left(\Delta^i_{(n+1-j)}\right)^{-2}} \\ &\geq \frac{T-n}{\overline{\log}(T)H_2^k}.\end{aligned} \quad (9)$$

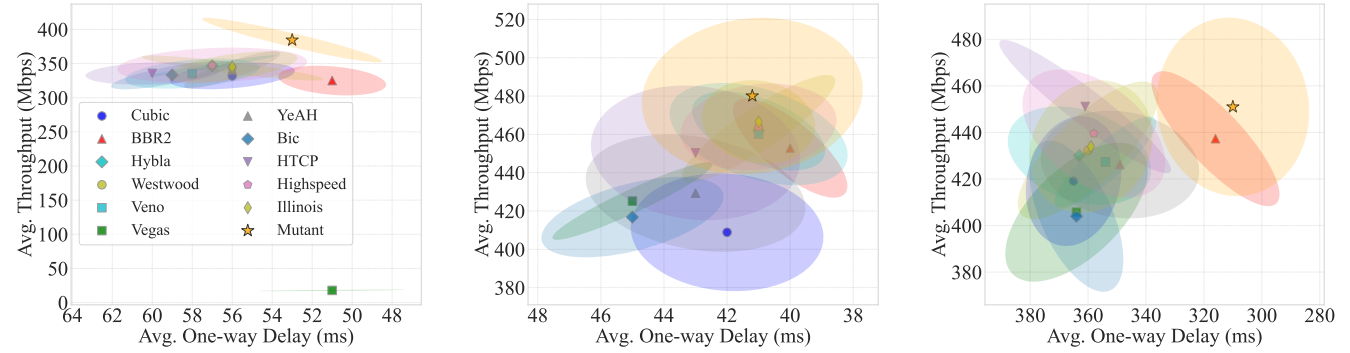
It is therefore sufficient to show that on event ξ, the algorithm does not make any error. This conclusion can be proved by induction on j. □

B More on Mutant's Performance

Mutant demonstrates higher performance across a wide range of network conditions in terms of both throughput and delay. We extended our evaluation to include additional cellular traces (Figure 15), encompassing both LTE and 5G networks with different ranges of bandwidth, where data are sampled from [2] and [6], respectively. Mutant consistently ranks among the top-performing protocols across diverse scenarios and shows its ability to adapt and maintain optimal performance.

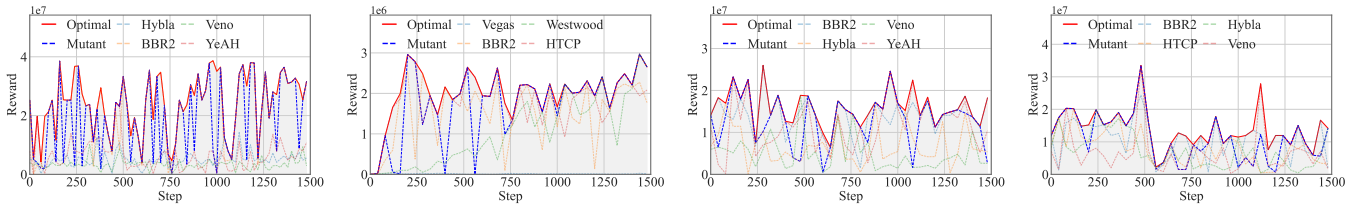


(a) cellular link 4 (Walking in Times Square) (b) cellular link 5 (Bus Riding) (c) cellular link 6 (Stationary Times Square)



(d) 5G cellular link 4 (Corniche) (e) 5G cellular link 5 (Corniche Walking) (f) 5G cellular link 6 (City Drive)

Figure 15: Average throughput and one-way delay over different real cellular network traces.



(a) 5G Cell. (Beach Stat.) (b) 5G Cell. (Beach Stat. 2) (c) 5G Cell. (City Drive) (d) 5G Cell. (Park Stationary)

Figure 16: Performance comparison of Mutant against individual protocols in four distinct 5G cellular scenarios. Mutant's reward performance is also against an idealized system that always chooses the "optimal" protocol.

In Figure 16, we report the reward of our Mutant protocol across four distinct high bandwidth 5G cellular scenarios. The reward function follows Equation 1. We compare Mutant's reward against an optimal system, which selects the best-performing protocol at each step of the run, and the reward of each of the CCAs in the pool. The pool of best-performing protocols is selected via the MPTS algorithm with $K = 4$. Our Mutant model was then trained to convergence and subsequently tested on network traces for 1500 steps and $\delta = 10^{-2}$ (see Section 6.1) in each scenario. We observe that Mutant, consistently achieves high rewards, demonstrating its adaptive capability by dynamically selecting the best-performing protocol at each step and for every network trace It is worth noting that the occasional dips in Mutant's performance, which correspond to intentional exploration (hence suboptimal) steps, are the only instances where it deviates from the optimal choice. However, these exploration phases enable Mutant to continually evaluate and adapt to potential changes in network conditions.