Learn-to-Probe: Achieving Signal Distinguishability in Learning-based Congestion Control

Han Tian[1], Wenbo Li[1], Junxue Zhang[1], Xudong Liao[2], Decang Sun[2], Donghui Chen[3], Bin Huang[3], Wenxue Li[2], Yong Wang[2], Kai Chen[2]
[1]University of Science and Technology of China
[2]iSING Lab, Hong Kong University of Science and Technology
[3]Huawei Technologies Co., Ltd.

Abstract

Internet congestion control remains a fundamental challenge, and recent learning-based congestion control algorithms (CCAs) have shown potential in optimizing network performance. However, their reliance on heuristically chosen input signals often leads to suboptimal behavior across diverse network conditions. In this paper, we identify the root cause as the lack of signal distinguishability–the ability of signals to reflect meaningful differences in network states. To address this, we propose Learn-to-Probe (LTP), a novel signal engineering paradigm that actively generates distinguishable network signals to improve the learning process. LTP (i) employs Bayesian filtering to accurately estimate network states from historical signals, and (ii) introduces an intrinsic reinforcement learning reward that encourages the flow to probe the network, inducing signal sequences that minimize the estimation uncertainty in (i). This probing behavior naturally enhances signal distinguishability, enabling the learning model to make more informed decisions. Extensive evaluations show that LTP consistently achieves high link utilization, low queuing delay, and stable convergence across diverse environments. Our results underscore the importance of signal distinguishability and offer a new direction for robust, adaptive congestion control.

CCS Concepts: • **Networks → Transport protocols**; • **Computing methodologies → Machine learning approaches**.

Keywords: Congestion Control, Machine Learning, Transport Protocol

ACM Reference Format:
Han Tian, Wenbo Li, Junxue Zhang, Xudong Liao, Decang Sun, Donghui Chen, Bin Huang, Wenxue Li, Yong Wang, Kai Chen. 2026. Learn-to-Probe: Achieving Signal Distinguishability in Learning-based Congestion Control. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3767295.3769317

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769317

1 Introduction

Internet congestion control (CC) remains one of the most fundamental and well-studied problems in networking. In recent years, the remarkable advances in learning-based algorithms across diverse domains such as games [27, 28, 32], computer systems [14, 44], and networking [25, 26, 42]—have motivated researchers to explore their potential for addressing the congestion control (CC) problem [1, 7, 8, 18, 22, 24, 38, 43].

Learning-based methods offer a data-driven approach: they automatically learn the mapping between observed signals and the optimal sending adjustments, thereby relieving network engineers and researchers from the need to handcraft rigid control rules. However, the effectiveness of these models heavily depends on how the learning components are defined—particularly the model inputs, which determine what the model observes. Choosing the right signals (e.g., RTT, loss rate, throughput) and performing effective feature transformation on them are key factors influencing the performance of learning-based congestion control algorithms (CCAs).

Many prior learning-based CCAs rely on trial-and-error approaches to handcraft their input signals, often lacking principled design guidelines. Typically, network operators collect training data from their specific deployment environments and use it to train a learning-based CCA in the hope of deriving an effective policy. However, we find that when trained across a wide range of diverse network conditions, this approach suffers from two key limitations (§2.2):

- Intra-env performance degradation: Though normalization is a necessary step for training learning-based CCAs to ensure the model's convergence, we observe that over-normalized input signals can cause the model to lose information about network states[1], hindering its ability to make differentiated control decisions and leading to poor performance.

[1]Network states encompass both static parameters–such as bottleneck link bandwidth and base RTT–and dynamic conditions, including queuing delays and the sending rates of competing flows at the bottleneck.

- Inter-env policy conflict: The learned policies in one environment degrade in another environment with different link characteristics (e.g., link capacity and base delay), and vice versa. This results in an "average" policy that is mediocre over all the training network environments with limited generalization.

In this paper, we argue that effective signals for learning-based CCAs must exhibit signal distinguishability. Specifically, a flow should receive distinct network signals under different network states, both within a single environment (intra-environment) and across different environments (inter-environment). Such distinguishable signals are essential for enabling the model to map signals to differentiated control actions, allowing for flexible and optimal policy adaptation. The core focus of this work is to identify and construct signals that possess strong signal distinguishability, enabling learning-based CCAs to consistently deliver high performance across diverse network conditions.

This leads us to the question: How can we engineer signals for learning-based CCAs to ensure they are distinguishable? In this paper, we answer it by *encouraging the flow to learn to probe network conditions.*

A probe policy in congestion control actively adjusts the sending rate and infers the network state based on the received signal responses. Our key insight is that a probe policy that aims to eliminate ambiguous network states based on signal responses naturally generates signals that exhibit superior signal distinguishability. By designing an effective probe policy, the signals it generates can directly serve as distinguishable input features for the learning model, improving the overall performance.

Building on this concept, we introduce Learn-to-Probe (LTP), a novel reinforcement learning-based paradigm for learning-based CCAs. LTP incorporates a signal engineering technique that generates distinguishable signals, enabling the training of performant policies. LTP uses a model-based probe method to extract informative signals, helping the model better identify the current network state. In addition to the standard reinforcement learning (RL) process, we introduce an estimation module that refines the reward signal by estimating the possible range of the network state. The narrower the estimated range, the higher the reward. This estimated range reward is combined with the congestion control performance reward to guide the model's actions. During training, the model learns to perform probing actions to gather distinguishable signals and use them to make optimal decisions.

We implemented and trained LTP on Linux servers with customized kernel modules (§6). Extensive evaluations (§7) show that LTP's estimation module achieves more accurate network state estimation. With high-quality signals as input, the policies learned by LTP consistently perform well across various network environments, including real-world Internet settings. Furthermore, LTP demonstrates strong fairness properties. We also highlight how the learned probe phase generates informative signals that guide the sending rate adjustment decision-making process. The code of LTP is available at https://github.com/tianhan4/learn-to-probe.

Our key contributions are:

- We develop a novel and accurate network state range estimation technique based on Bayesian filtering, enabling the estimation of network states based on observable signals at the end-hosts;
- We introduce the Learn-to-Probe paradigm, which leverages reinforcement learning to actively probe the network and generate signals that are distinguishable across both intra- and inter-environment scenarios;
- We present a comprehensive learning-based CCA learning framework, which, supported by the generated signals, achieves consistently high performance across various network environments and demonstrates good convergence properties.

2 Design Philosophy

2.1 Learning-based CCAs

Internet congestion control algorithms are essential for managing network traffic, adjusting flow sending rates to avoid congestion and ensure fair resource allocation. Traditional CCAs rely on signals like round-trip time (RTT), packet loss rate, and throughput, typically sampled through acknowledgements (Acks) at end-hosts. These signals are then used to infer the current state of the network path. For example, in heuristic-based CCAs, such as Cubic [13], the loss event is treated as a strong indicator of congestion, triggering a predefined adjustment to the congestion window.

In contrast, recent learning-based CCAs [1, 18, 22, 38, 39, 43] take a more data-driven approach. Instead of relying on predefined rules, these CCAs automatically learn the mapping between signals and optimal actions using machine learning algorithms. While this shift reduces the need for manual rule crafting, it places greater emphasis on signal design—particularly the choice of input states and how they are pre-processed. Table 1 illustrates the input signals used by some recent learning-based CCAs. Although some of these methods use normalization techniques (e.g., min-max scaling or ratio-based transformations) to improve generalization, the signal selection and normalization processes are often ad hoc, lacking solid guidelines.

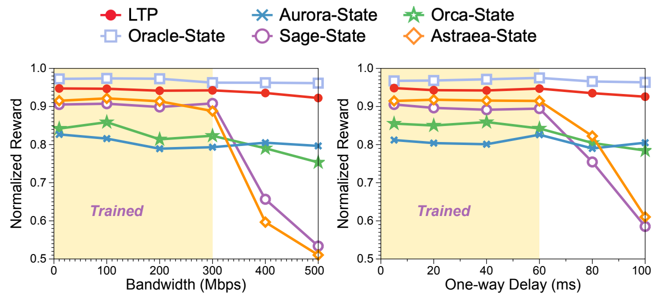
2.2 Why Distinguishability?

To explore the impact of signal selection and normalization on learning-based CCAs, we conduct experiments comparing the performance of algorithms from Table 1. For this comparison, we retain their signal definitions and use the same reward function, action function, and training paradigm as described in this paper[2]. We call these CCA variants

[2]Orca is a hybrid control algorithm with Cubic and deep reinforcement learning (DRL) components. We retrain only the DRL part.

DRL-based CCA	**Aurora** [18]	**Orca** [1]	**Sage** [43]	**Astraea** [22]
Input signals	Latency gradient, latency ratio, sending ratio	Throughput ratio, sending rate ratio, packet loss ratio, inflight packet ratio, latency ratio	69 features including i) avg, min, and max values of measurements; and ii) hand-crafted normalized states.	Throughput ratio, sending rate/window ratio, packet loss ratio, inflight packet ratio, latency ratio, maximum throughput, minimum latency.
All signals normalized?	Y	Y	N	N

Table 1. A comparison of the input signals of recent learning-based CC baselines.



(a) Bandwidth generalization. (b) Delay generalization.

Figure 1. The control performance of DRL-based CCAs varies with different state inputs.

CCA-State. The models are trained in emulated links with bandwidth ranging from 0 to 300 Mbps and base delay from 0 to 60 ms, and evaluated in both training and testing environments[3]. To demonstrate the performance ceiling of a policy with perfect network knowledge, we modified Astraea to incorporate global network information– bottleneck link utilization and queuing delay–into the model input, which we call Oracle-State. In this setting, flows with different link utilizations and queuing delays are mapped to clearly distinguishable feature vectors.

Fig. 1 shows the rewards collected by these baselines across network environments (with the yellow region indicates the training region). We observe that all prior schemes suffer from performance degradation outside the training region, while Oracle-State significantly outperforms the rest. This suggests that existing signal sets fail to capture critical distinctions between network states. Upon closer analysis, we identify two key causes of this degradation:

Intra-env performance degradation. Models relying solely on normalized signals (e.g., Aurora-State and Orca-State) exhibit stable but suboptimal performance. This is because over-normalization obscures key network information—distinct network states may appear identical to the model, even when they demand different control actions. For example, normalizing throughput by the maximum observed value can yield a constant signal (e.g., always 1), even as the actual sending rate continues to increase. This lack of signal variation, especially under competing flows or network jitter, hinders the model's ability to adapt effectively.

[3]For evaluation, we fixed bandwidth at 100Mbps when changing delay, and fixed delay at 20ms when changing bandwidth.

Inter-environment Policy Conflicts. A naïve solution is to include more unnormalized features, as in Sage-State and Astraea-State. While this improves training performance slightly, it often degrades generalization to unseen environments and still falls short of Oracle-State. Simply expanding the training region, as shown in §7.2, exacerbates the problem. The issue arises because flows in different environments may produce similar signals despite requiring different control strategies. Consider a flow observing 110 Mbps sending rate and increased latency. This could mean:

- The flow is fully utilizing a 100 Mbps bottleneck, and the latency increase is self-induced. It should reduce its rate;
- The flow is using just 11% of a 1 Gbps link, and the latency increase is caused by a competing flow. It should increase its rate to seize more bandwidth.

In both cases, identical signals lead to contradictory control decisions. This creates a policy conflict, where a single model cannot resolve competing behaviors, resulting in the "whack-a-mole" effect—fixes in one environment cause regressions in others.

Good Signals for Learning-based CCAs From these observations, we argue that good signals for learning-based CCAs should exhibit distinguishability—the ability of signals to differentiate between distinct network states under different network environments, enabling the model to take appropriate control actions (e.g., adjusting sending rate). This paper aims to develop such signals.

2.3 Probe to be Distinguishable

Based on the success of introducing global network information, we propose using probing to improve signal distinguishability. A probe policy in congestion control actively adjusts the sending rate and infers the network state based on the resulting signal responses. For example, BBR periodically drains inflight packets and uses a min-filter on observed RTT signals to estimate the link's base RTT [6].

Our design is inspired by an insightful observation: *a probe policy aimed at eliminating unlikely network states naturally produces signal sequences with high distinguishability.* This occurs because probing helps rule out possible states—if the observed signal is inconsistent with a particular network state, that state can be excluded. As a result, an effective probe policy targeting a network state variable X induces distinct signal response patterns for different values of X. These patterns can then serve as discriminative features for the learning-based control policy, enabling it to make more

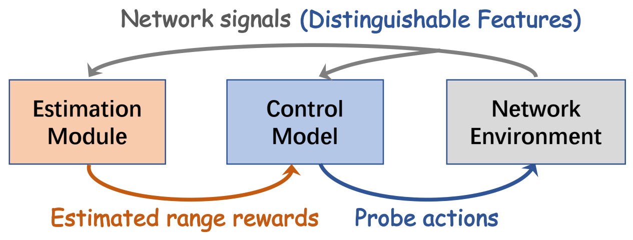


Figure 2. Overview of the learn-to-probe mechanism.

informed decisions by reacting differently to varying values of X.

Many previous CCAs have incorporated heuristic-based probing methods [4, 6]. A straightforward approach would be to adopt these traditional probing policies and use inferred network states as model input. However, heuristic-based probing can be suboptimal, often vulnerable to network jitters or specific network conditions. For example, BBR's probeRTT phase can be affected by non-congestive delays or competing flows that do not cooperate in draining their inflight packets, leading to inaccurate base delay estimations [15]. Moreover, manually balancing the trade-off between exploration and exploitation across diverse network environments is inherently challenging and inflexible.

Therefore, instead of inferring network parameters and flow states, we propose a fully end-to-end learning-based approach to learn to probe the network. Precisely inferring network states can be challenging and error-prone in practice. Therefore, rather than pinpointing exact values, we design our method to learn how to minimize the possible ranges of network states, focusing on improving the estimation confidence (§4). The probe policy in LTP is learned to probe the three most critical yet unobservable network states from an end-host's congestion control perspective: *the flow's own bottleneck bandwidth occupancy, the occupancy by competing flows, and the current level of congestion* (§5). Furthermore, by designing the learning process over normalized network states, the learned probe policy can also generalize to unseen network environments.

3 LTP Overview

To learn how to probe for distinguishable signals, we introduce LTP, a novel probabilistic reinforcement learning paradigm. The probe goal of LTP is to narrow down the possible network state ranges based on the received signals, enhancing the model's ability to distinguish between different network conditions.

We provide an overview of the LTP framework in Fig. 2. In addition to the standard reinforcement learning process for learning a control policy offline, LTP incorporates an estimation module that refines the reward signal received by the flow agent. During training, the estimation module and the flow agent operate in parallel: at each control interval (e.g., 30 ms), network signals are provided both to the flow agent as input and to the estimation module to infer the current network state. The estimation module estimates the possible range of the flow's status, with narrower ranges indicating higher rewards, and this estimated range reward is combined with the performance reward from congestion control to form the final reward input. While probing is not explicitly present in the deployed model, it implicitly shapes the learned control policy during training: the probe reward term encourages effective exploration of network dynamics and stabilizes learning under noisy conditions, after which the trained model directly generates rate control actions without additional probing overhead.

The model is trained using RL with the aim of maximizing the cumulative reward accumulated over the flow's lifetime. Through this process, the model learns to i) initiate probing actions when necessary to gather distinguishable network signals, and ii) leverage these signals to perform optimal congestion control actions. By using the distinguishable signal sequences generated through the probing policy, the learning-based control policy significantly improves its performance. Furthermore, both the estimation module and the CC model rely on normalized states, ensuring that learned policy can also generalize to unseen network environments. A key benefit of our explicit network modeling is that we can easily observe and interpret how the probe and performance rewards are generated during the training process. This helps us understand LTP's behavior and decisions in various network settings. The following sections provide a detailed description of the key components of the LTP framework.

4 Estimate the Network

The estimation module is the core component that determines the effectiveness of the learned policy. The primary objective is to design a model-based estimation approach that is both accurate and robust to network jitters and interference from competing flows.

Previous congestion control algorithms have introduced heuristic-based methods to infer target network states directly [4, 6]. For instance, CCmatic [2] employs a belief set to represent possible ranges of network states based on historical interactions. However, these methods often fail to fully leverage multiple sources of signal sequences or adequately address multi-flow scenarios. As a result, their estimations tend to be overly broad or inaccurate, limiting their ability to effectively narrow down potential network conditions.

Our Estimation Method Our estimation method draws inspiration from particle filtering–a sequential Monte Carlo technique widely used in Bayesian inference for state estimation and filtering tasks [9]. It is designed with the following key features:

- Models the network with multiple sources of variance, including jitter and interference from competing flows;

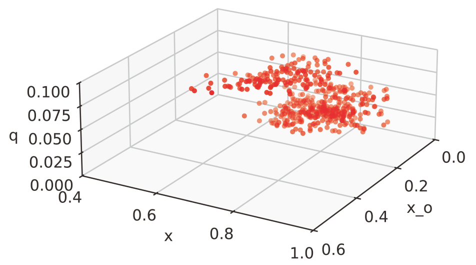


Figure 3. Network state particles.

- Incorporates a transition model to update state estimates over time using newly observed signals;
- Jointly considers multiple signals to produce tighter, more accurate estimations of the underlying network state;

This design enables precise and reliable inference of network conditions, enhancing the learning process of the probe policy. The estimation process begins with an initial set of particles representing possible network states and proceeds through four core steps at each time interval:

① **Transition Step:** Updates each particle's state based on the previous state and a predefined network transition model.
② **Observation Step:** Computes the expected signal values for each particle using the predefined network's measurement model.
③ **Filtering Step:** Eliminates particles whose predicted signals deviate significantly from the actual observations.
④ **Resampling Step:** Generates new particles by applying random mutations to the filtered set, ensuring diversity and continued exploration.

This iterative process enables the system to track the network state over time, with the particles converging toward the most probable state trajectories as more signals are observed. By representing the potential distribution of network state as a discrete set of particles, our method can handle complex, non-linear estimated ranges. Fig. 3 provides a snapshot of the particles maintained during the estimation process.

Noisy Model Specification We extend the CCAC model [3] with i) competing flows; ii) network state transition; and iii) multiple signal sources. Fig. 4 illustrates our probabilistic noisy model. To ensure robust estimation, the model incorporates four types of variances: i) link variance, such as rate limiting imposed by token bucket filters; ii) non-congestive loss jitter, e.g., packet loss caused by factors like wireless link instability; iii) non-congestive delay jitter, e.g., extra random delays that occur due to network conditions; and iv) sending rate changes of competing flows. We assume the introduced noises are i.i.d., meaning each step's noise is independent of previous steps, which allows our model to account for a wider range of possible signal variations. In contrast, highly correlated or adversarial noise could further constrain the inferred network state ranges, which we leave as future work.

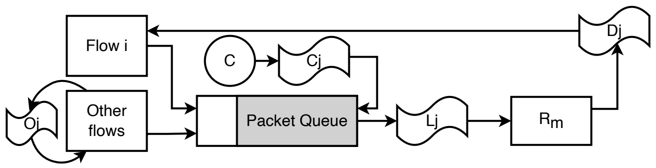


Figure 4. Noisy network model from a sender's view.

c - bottleneck link capacity	d - queuing delay
l - loss rate	x - sending rate
R_m - propagation delay	x_o - other flows' sending rate
L_j - loss jitter range	D_j - delay jitter range
C_j - bandwidth jitter range	O_j - gradient range

Table 2. Glossary of symbols in the network model.

Since we only explicitly model the bottleneck behavior of competing flows, their RTTs and loss rates can vary, allowing our model to generalize across diverse real-world network topologies. We parameterize the noise level of the model with L_j, D_j, C_j, and O_j representing the variances of loss, delay, bandwidth, and other flows' sending rates, respectively. Table 2 lists the symbols used in the network model. We note that the symbols are modeling constructs for simulating network state transitions, rather than features directly consumed by our RL model. They describe the network state and signals, enabling us to accurately capture the transition dynamics of the network.

We define two parts for the model: i) the network state transition function defines how the flow state transitions over time at the bottleneck queue[4], and ii) the signal measurement function defines what signal values the flow observes given the network state. We model the network state transition as a Markov chain, where the state at time t depends on the state at time $t-1$. The signal measurement process is a function of the network state and noise at time t. For simplicity, we assume that the noise is independent and identically distributed (i.i.d.) across time intervals. Future work will address highly correlated or adversarial noise.

For each particle, we record the following normalized network states:

- The ratio of the sending rate of our target flow to the bottleneck link capacity $\overline{x} = \frac{x}{c}$.
- The ratio of the overall sending rate of competing flows to the bottleneck link capacity $\overline{x_o} = \frac{x_o}{c}$.
- The queuing delay d.
- A binary variable is_full, indicating whether the queue is full.

Unlike previous methods, we do not estimate static network states (e.g., bottleneck link capacity C or base RTT R_m), since the estimated rewards in network environments

[4]We focus on the timeline at the bottleneck in this section.

with different link characteristics would not be generalizable.

Why not normalize d? We adopt queue delay without normalization as changes in queuing time inherently represent the ratio between the sending rate and bottleneck link capacity, independent of the link capacity and the end-to-end delay. The change in RTT over a time interval Δt can be approximated as:

$$\Delta t \cdot \frac{x + x_o - c}{c}.$$

Transition Step. During the state transition from time t_1 to t_2, we assume that the time interval $\Delta t = t_2 - t_1$ is small enough that sending rates remain constant throughout the interval, and update the sending rates at the start of the interval.

For $\overline{x}$, we update it based on the multiplicative sending rate action performed:

$$\overline{x}(t_2) = \overline{x}(t_1) \cdot a(t_2).$$

For the sending rates of competing flows, we assume their continuity. We sample a set of gradients o_j evenly from $[0, O_j]$ that represent changes in the sending rates of competing flows at the bottleneck. The updated $\overline{x_o}$ is given by:

$$\overline{x_o}(t_2) = \max(0, \overline{x_o}(t_1) + \overline{x_o}(t_1) \cdot o_j(t_2) \cdot \Delta t).$$

The change in the packet queue size depends on the difference between the incoming and outgoing bytes. The incoming bytes are determined by the total sending rate $(\overline{x_o}(t_2) + \overline{x}(t_2)) \cdot \Delta t$, while the outgoing bytes depend on the bottleneck link capacity $c \cdot \Delta t$. Therefore, the queue size is updated as follows:

$$d(t_2) = \begin{cases} \min(d(t_1), d'(t_2)) & \text{if } is\_full(t_1) = 1, \\ d'(t_2) & \text{otherwise.} \end{cases}$$

where

$$d'(t_2) = \max(0, d(t_1) + (\overline{x_o}(t_2) + \overline{x}(t_2) - 1) \cdot \Delta t).$$

If $d(t_2)$ reaches the maximum recorded value, the queue may be saturated. In this case, we perform a particle division: removing the particle and creating two new particles–one with $is\_full(t_2) = 1$ and another with $is\_full(t_2) = 0$, keeping other states the same.

Observation Step. The network states are estimated through observed signals from end-host flows. In this step, we introduce the network measurement model that defines how a flow sender observes signals such as RTT, loss rate, and throughput based on its network state.

To ensure generalizability, we normalize the observed signals as follows:

- The additive change in RTT, denoted by $\Delta RTT = RTT(t_2) - RTT(t_1)$.
- The multiplicative change in throughput, $\Delta thr = \frac{thr(t_2)}{thr(t_1)}$.
- The change in loss rate, represented by the multiplicative change in the acknowledged ratio $\Delta L = \frac{1-l(t_2)}{1-l(t_1)}$.
- The throughput ratio, normalized by the maximum throughput observed so far for the sender: $\tilde{thr} = \frac{thr(t_2)}{thr_{\max}}$.
- The RTT difference, normalized by the minimum RTT observed so far for the sender: $\tilde{RTT} = RTT(t_2) - RTT_{\min}$.

To align with the timeline at the bottleneck used in previous steps, our signal collection ensures that packets sent during each interval have their responses correctly recorded, and adopts these responses as observed signals. Therefore, the difference values ΔRTT, Δthr, ΔL end-host flows observe are the changes in the last two intervals whose sent packets are acknowledged, and $\tilde{thr}$ and $\tilde{RTT}$ are calculated based on the last interval. Here we select the observed signals based on heuristics. One possible improvement is to use Principal Component Analysis (PCA) to select the most informative signals, which is left for future work.

Here we introduce how these signals are calculated based on the network state. For ΔRTT, it is equivalent to the change in the queuing delay:

$$\Delta RTT(t_2) = d(t_2) - d(t_1) \tag{1}$$

The throughput signal Δthr depends on several factors: i) the queue status, ii) the sending rate of the target flow, and iii) the sending rates of competing flows. When there are packets in the queue or the overall sending rate exceeds the bottleneck link capacity, $(\overline{x}(t_2) + \overline{x_o}(t_2) > 1)$, the overall throughput is capped at the bottleneck link capacity, distributed across flows based on their sending rates. Therefore, the observed throughput for the target sender will be:

$$thr(t) = c \cdot \frac{\overline{x}(t)}{\overline{x}(t) + \overline{x_o}(t)}$$

When the queue is empty and the overall sending rate is less than or equal to the bottleneck link capacity, $(\overline{x}(t) + \overline{x_o}(t) \leq 1)$, the throughput equals the sending rate:

$$thr(t) = c \cdot \overline{x}(t)$$

This leads to the following piecewise definition for throughput:

$$thr(t) = \begin{cases} c \cdot \overline{x}(t) & \text{if } \overline{x}(t) + \overline{x_o}(t) \leq 1 \text{ and } q(t) = 0, \\ c \cdot \frac{\overline{x}(t)}{\overline{x}(t) + \overline{x_o}(t)} & \text{otherwise.} \end{cases} \tag{2}$$

It is then straightforward to calculate the change in throughput, $\Delta thr = \frac{thr(t_2)}{thr(t_1)}$, where the bottleneck link capacity c cancels out.

The loss signal ΔL depends on whether the packet queue is full. When $is\_full(t) = 1$, packets are dropped, and congestive packet loss occurs, which is modeled as:

$$l(t) = 1 - \frac{1}{\overline{x}(t) + \overline{x_o}(t)}$$

Thus, ΔL can be calculated as follows. Given the loss rate $l(t)$ at time t as:

$$l(t) = \begin{cases} \eta_l \cdot (1 - \frac{1}{\overline{x}(t) + \overline{x_o}(t)}) & \text{if } is\_full(t) = 1 \text{ and } \overline{x}(t) + \overline{x_o}(t) > 1, \\ \eta_l & \text{otherwise.} \end{cases} \quad (3)$$

where the loss jitter is sampled $\eta_l \sim \mathcal{U}(0, L_j)$. Then, it is then straightforward to calculate $\Delta L = \frac{1-l(t_2)}{1-l(t_1)}$ based on Equation 3.

The normalized signals $\tilde{thr}(t_2)$ and $\tilde{RTT}(t_2)$ can be directly used in the filtering step with the following constraints:

$$\tilde{thr}(t) \geq \frac{thr(t)}{c} \quad \text{and} \quad \tilde{RTT}(t) \leq q(t), \quad (4)$$

where the maximum throughput observed must be less than or equal to the bottleneck link capacity, and the minimum RTT observed must be greater than or equal to the base RTT.

Filtering step. In the filtering step, we collect the observed signals for ΔRTT, Δthr, ΔL, $\tilde{thr}$, and $\tilde{RTT}$, comparing them with the expected results for each particle, and filter out mismatched particles. We design a mismatch function to score the divergence between the network states represented by particles and the observed signals:

$$mismatch\_score = \prod_{s \in signals} \frac{gap_s}{1 + gap_s}, \quad (5)$$

where $gap_s = \sigma_s \times |observed_s - expected_s|$. The scaling factor σ_s represents the sensitivity of different signals to measurement error. The mismatch score lies in the range $[0, 1)$. When all the observed signals perfectly match the expected signals, the mismatch score equals 0. To achieve robust estimation, we utilize the mismatch score as the filter probability to discard particles with large gaps. In this way, some mismatched particles may still survive to account for measurement noise. The mismatch score is in the range $[0, 1]$ and we use it directly as the filtering probability. For example, a particle with a mismatch score of 0.5 has a 50% chance of being discarded.

If no particle survives, we assume that the estimation process has diverged from the actual state. In this case, the module re-initializes the particle pool and recalculates the estimation range from scratch.

Resampling step. After the filtering step, the resampling step is invoked to replenish the particle pool and introduce network variances, maintaining a fine-grained estimation. During resampling, survived particles are selected, mutated, and appended to the particle pool. Since our noisy network model considers network jitters affecting link bandwidth, delay, and the loss rate, we also add sampled noise to the network states $\overline{x}$, $\overline{x_o}$, and d.

Specifically, for the normalized sending rates $\overline{x}$ and $\overline{x_o}$, we add multiplicative noise sampled from a uniform distribution $\mathcal{U}(-C_j, C_j)$:

$$\begin{aligned} \overline{x^{new}} &= \overline{x} * (1 + \eta_c), \\ \overline{x_o^{new}} &= \overline{x_o} * (1 + \eta_c), \\ \eta_c &\sim \mathcal{U}(-C_j, C_j). \end{aligned} \quad (6)$$

For the queuing delay d, we add additive noise sampled from a uniform distribution $\eta_d \sim \mathcal{U}(-D_j, D_j)$:

$$d^{new} = \max(0, d + \eta_d), \quad \eta_d \sim \mathcal{U}(-D_j, D_j). \quad (7)$$

Since the is_full indicator is binary and determined during the transition step, no further mutation is performed on it. By introducing variance into the particle pool, LTP has the opportunity to restore correct estimations, even if the estimation direction diverges, thus ensuring robustness to network jitters.

5 Learn to Probe

The learning process of the control model is conducted using deep reinforcement learning (DRL), which optimizes a deep neural network through interactions between the flow agent and the network environment. During training, the flow sender interacts with the network environment to gather training data: For the t-th time interval, the sender observes network signals (e.g., throughput, latency, and loss) as its input state $s_t \in \mathcal{S}$, and generates an action $a_t \in \mathcal{A}$ to adjust the sending rate. The link state changes correspondingly after the agent's action, and the flow receives new signals s_{t+1}. The sender then obtains a reward r_t from the environment and updates its policy based on r_t to build a mapping between the input state and the output action. The DRL agent updates the control policy embedded in the deep neural network model to maximize the discounted cumulative expected reward:$\mathcal{J} = \mathbb{E}\left(\sum_{t=0}^{T} \gamma^t r_t\right)$, where γ is a discount factor. The well-learned policy drives the network states to actions that lead to effective probing or high control performance, depending on the reward definition.

Here, we introduce the RL components used in the model, including the input state, action, and reward definitions.

State. The input state consists of the same observable and generalizable signals—ΔRTT, Δthr, ΔL, $\tilde{thr}$, and $\tilde{RTT}$—used in the observation step in estimation module for two key reasons. First, by using normalized signals as input, we ensure the generalizability of the learned probe policy. Second, since the model will probe to make the input signal for the estimation module distinguishable, it is crucial to use these signals to provide rich information about the network status for performant control. Similar signals have been used in previous learning-based CCAs [37] and achieve good model performance.

Action. For the action policy, we apply the sending rate adjustment used in [22]. The model outputs an action value between $[-1, 1]$, and the sending rate change is defined as follows:

$$cwnd_{t+1} = \begin{cases} cwnd_t \times (1 + \alpha \times a_t) & \text{if } a_t \geq 0, \\ cwnd_t / (1 - \alpha \times a_t) & \text{otherwise.} \end{cases} \quad (8)$$

where the hyperparameter α controls the policy's aggressiveness. The datapath then adopts a uniform pacing rate based on the updated congestion window and the average RTT observed in the last interval:

$$x_{t+1} = \frac{cwnd_{t+1}}{RTT_t}.$$

Reward. The reward function consists of both the probe goal and the performance goal.

① **Probe Reward.** As discussed earlier, the probe goal is related to the estimated ranges derived from the estimation module. The reward is computed based on three terms corresponding to the ranges of $\overline{x}$, $\overline{x_o}$, and d, respectively. To avoid outliers, we use the 10% and 90% quantiles for each range. For example, the reward term for the queue length range is given by:

$$\text{reward}_q = -\beta_q \cdot (q_{90} - q_{10}), \quad (9)$$

where q_{10} and q_{90} are the 10% and 90% quantiles of the queue length values, and β_q is a scaling factor. Similarly, we have:

$$\begin{aligned} \text{reward}_{\overline{x}} &= -\beta_{\overline{x}} \cdot \log\left(\frac{\overline{x}_{90}}{\overline{x}_{10}}\right), \\ \text{reward}_{\overline{x_o}} &= -\beta_{\overline{x_o}} \cdot \log\left(\frac{\overline{x_o}_{90}}{\overline{x_o}_{10}}\right). \end{aligned} \quad (10)$$

We use logarithms on the sending rate ratios because the sending rate action defined in Equation 8 is multiplicative. This ensures that performing transitions on normalized sending rates does not change the reward without additional constraints. We can also interpret the rewards for rate ranges as indicators of how many control actions are needed for the flow to change its rate from the lower bound to the upper bound.

Finally, the total estimation reward is the sum of these individual rewards:

$$\text{reward}_{esti} = \text{reward}_q + \text{reward}_{\overline{x}} + \text{reward}_{\overline{x_o}} + \beta_r. \quad (11)$$

We set the constant β_r to ensure the average probe reward is zero, which accelerates the convergence of the reinforcement learning process by penalizing control behaviors that lead to indistinguishable network estimation. This value was determined through trial-and-error, and without it, the learning process struggles to converge. We do not include the range reward for the is_full indicator, since, similar to other modern CCAs [4, 6, 18], our CCA operates at the equilibrium point where the link is fully utilized and the queue is nearly empty. Therefore, actively probing the upper bound of the queue (like in Cubic [13]) is unnecessary and may lead to additional packet loss.

② **Performance Reward.** We define the performance reward to quantify the performance criterion of the control task. Inspired by the power-based reward in Orca [1] and the fairness metric in Astraea [22], we define the performance reward function as follows:

$$R_{perf} = \left(\frac{thr - \zeta \times loss}{lat'}\right) \bigg/ \left(\frac{thr_{\max}}{lat_{\min}}\right) + \beta_{fair} \times R_{fair}, \quad (12)$$

where

$$lat' = \begin{cases} lat_{\min} & \text{if } lat_{\min} \leq lat \leq \beta_{lat} \times lat_{\min}, \\ lat & \text{otherwise.} \end{cases} \quad (13)$$

and

$$R_{fair} = \sqrt{\frac{\sum_i \left(avg\_thr_i - \frac{1}{n}\sum_i avg\_thr_i\right)^2}{n\left(\sum_i avg\_thr_i\right)^2}}, \quad (14)$$

where avg_thr_i is the average throughput of the i-th flow, and n is the number of competing flows at the link bottleneck.

The first term in Equation 12 originates from Orca, denoting the ratio of normalized throughput to normalized latency, with a penalty on lost packets. It also tolerates small queuing delay (as defined in Equation 13) to achieve maximum bandwidth. The second term is inspired by Astraea and explicitly represents the fairness metric across flows. With this reward function, the RL algorithm rewards high throughput, low latency, and good fairness.

The final reward combines the two components:

$$R = \beta_{esti} \times R_{esti} + \beta_{perf} \times R_{perf}. \quad (15)$$

RL Approach. Similar to previous DRL-based CCAs [1, 22, 38], we adopt the model-free off-policy DRL training algorithm Deep Deterministic Policy Gradient (DDPG) [23] to update the policy model. To achieve intelligent decision-making, we employ recurrent neural networks (RNNs) as the building blocks of our model, which can embed the history of signals as input. Several optimization techniques, such as TD3 [12] and R2D2 [19], are used to improve the learning process. For further details on the training algorithm, please refer to Appendix B.

6 Implementation

We implement a fully functional LTP prototype on Linux, which consists of two main components: a CC kernel module and a DRL agent running in userspace. The kernel module collects observable network signals on the sender side and sends them to the RL agent, which is responsible for processing the signals and returning control actions. The kernel module then uses these actions to adjust the congestion window and pacing rate. To facilitate communication between the kernel module and the RL agent, we implement a cross-space communication channel using Netlink [31], allowing efficient interaction between the kernel and userspace. For the DRL model design and training, we leverage the DI-engine framework [10], which is built on PyTorch [30]. This

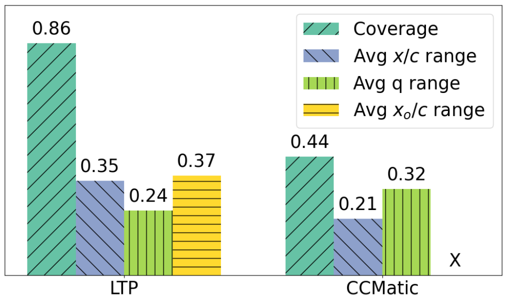


Figure 5. Range estimation of LTP and CCmatic.

framework supports a wide range of deep reinforcement learning algorithms and enables the creation of custom policies. Our DRL model consists of three fully connected layers, each with 128 dimensions, forming the basis of our recurrent neural network (RNN) architecture.

We use Mahimahi [29] for network emulation and Pantheon-tunnel [41] to simulate network tunnels. We train and evaluate LTP on a Linux server with 80 CPU cores, 256GB of RAM. To increase fairness and friendliness, we collect the training data by running various number of competing flows (2-10) concurrently controlled by either Cubic or LTP, with various extra delays and loss rates to simulate complex network topologies.

To stabilize the learning process of LTP, we carefully select and fine-tune all RL training hyperparameters. Detailed information about the hyperparameters, including their specific values and additional training details, can be found in Appendix D.

7 Evaluation

We evaluate LTP through extensive emulation and testbed experiments, which reveal the following key results:

- We demonstrate that the estimation module of LTP provides accurate and consistent estimations of network states, such as queue length and rate occupancy ratio, particularly when the system is contending with other flows (§7.1).
- We demonstrate the effectiveness of the probe reward in guiding LTP's learning process. When trained across a wide range of network environments, LTP successfully learns distinct policies tailored to different conditions, enabling robust and adaptive performance (§7.2).
- We show that LTP, leveraging distinguishable signals, consistently achieves high performance and exhibits strong convergence properties (§7.3) across diverse network environments.
- We highlight how LTP distinguishes itself from previous learning-based algorithms by incorporating probing actions, which help explore the network environment and enhance the performance of the control process (§7.5).

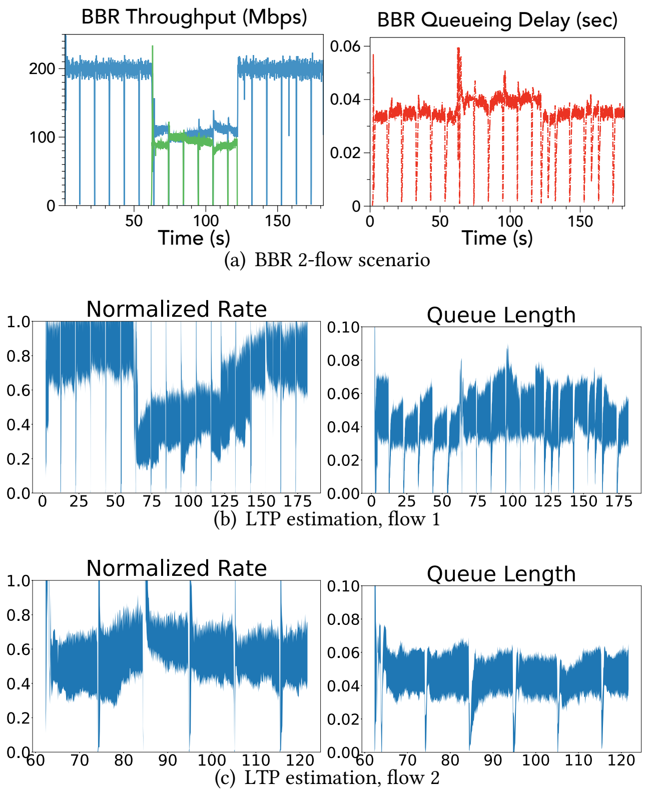


Figure 6. Estimation results of LTP in BBR 2-flow scenario. Two metrics are estimated for the two competing flows respectively: the queuing delay and the sending rate normalized by the link bandwidth (x/c).

7.1 Range Estimation

The performance of the estimation module in LTP directly influences the ability to generate distinguishable features for DRL-based CCAs. In this section, we evaluate the accuracy of LTP's network state range estimation, comparing it to the baseline estimation method, CCmatic [2]. Specifically, we apply both estimation methods to traces from multi-flow scenarios involving various congestion control protocols, including Cubic, Vegas, BBR, Copa, and our approach, LTP. For each time interval (0.1 seconds), we record the necessary signals and update the estimation ranges for each method.

We evaluate the following average metrics: i) coverage, which represents the proportion of intervals where the actual network state falls within the estimated range; and ii) estimated range width, which includes metrics such as the normalized sending rate $\frac{x}{c}$, the queuing delay q, and the normalized overall sending rate of competing flows $\frac{x_o}{c}$. Since CCmatic estimates link bandwidth C rather than the sending rate, we compute the range $\frac{x}{c}$ based on the actual sending rate x and the estimated link bandwidth.

As shown in Figure 5, LTP significantly outperforms CC-Matic in terms of estimation coverage, even with comparable range widths. This performance gap arises because CCmatic

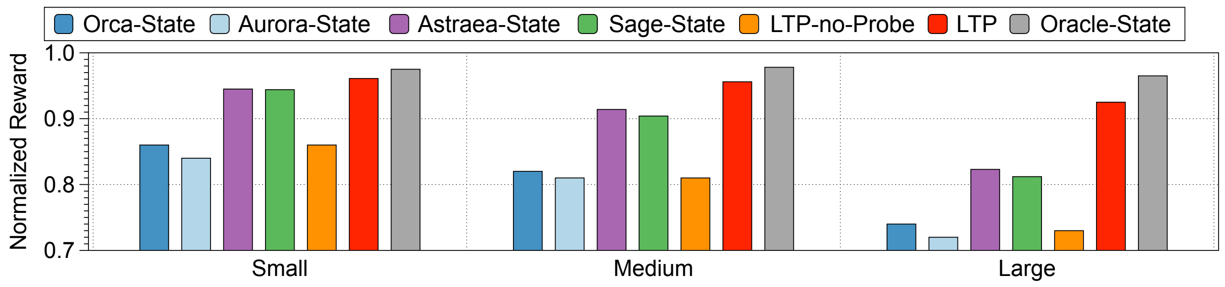


Figure 7. The average normalized reward of LTP and baselines under various training regions.

Env	Bandwidth	Base delay	Buffer size	Loss rate
Small	10-100 Mbps	5-30 ms	0.8-1 BDP	0-0.1%
Medium	10-300 Mbps	5-60 ms	0.8-1.5 BDP	0-0.5%
Large	10-600 Mbps	5-120 ms	0.8-2 BDP	0-1%

Table 3. Training environments.

does not account for the impact of competing flows, limiting its ability to handle multi-flow scenarios. Figure 6 provides a closer look at the estimation results for LTP in a two-flow BBR setup. We observe that LTP, by explicitly modeling the sending rates of competing flows, correctly adjusts its estimation, recognizing that the bandwidth is now shared among multiple flows. LTP benefits from several techniques, such as transition mechanisms, multi-signal constraints, and particle-based flexible estimation, as outlined in §4, which enhance its estimation accuracy. Also, we observe that the convergence process requires only a few control intervals to collect enough signals. We attribute it to the efficiency of our network modeling and state elimination.

7.2 Enhancing Learning with LTP

In this section, we evaluate the performance of LTP under different training region sizes. As in the motivation experiment in §2.2, we prepare baselines by retaining with only their original input states while adopting the same reward function, action space, and training paradigm as LTP. This ensures a fair comparison focused solely on the impact of input signal design. To analyze the contribution of the probe reward, we introduce an ablated variant of our method, LTP-no-probe, which disables the probe reward during training. We train LTP, LTP-no-probe, and all baselines across three training region sizes—small, medium, and large (defined in Table 3)—and evaluate their performance using the average normalized reward inside each training region, as defined in Equation 15.

Figure 7 shows the average normalized reward across 100 sampled environments from each training region. Variance across runs is within ±5%. Overall, LTP consistently achieves the highest reward among all learning-based approaches and performs close to the Oracle-State, which leverages full knowledge of link capacity and base RTT, highlighting the value of the probe reward in learning to infer environment-specific states. As the training region expands, the performance of all baselines declines significantly due to increased signal ambiguity across diverse environments. In contrast, LTP maintains stable performance. In smaller training regions, baselines using unnormalized signals (e.g., Astraea-State and Sage-State) outperform normalized ones (e.g., Orca-State and Aurora-State). The reason is that they overfits to the training region, as shown in Figure 1, where they fail to generalize to unseen environments. In the following section, we use LTP trained in the large training region to evaluate its performance across various network conditions.

7.3 Consistent High Performance

We show LTP's consistent high performance across a wide range of network environments through emulated experiments and real-world. We compare LTP with various learning-based and heuristic-based CCAs including Astraea [23], Orca [1], Aurora [18], Vivace [8], Cubic [15], BBR [4], Copa [3]. For learning-based schemes, we all use the published code and model provided by the authors.

7.3.1 Extensive Emulations. We first evaluate LTP against other congestion control algorithms across a diverse set of emulated network environments. Specifically, we compare link utilization and delay ratio by varying key link characteristics: bandwidth, base delay, random loss rate, and buffer size. The emulations use a dumbbell topology with a single flow, where we vary one link characteristic at a time while keeping the others constant. The parameters span bandwidths from 10 to 600 Mbps, base delays from 15 to 120 ms, random loss rates from 0% to 1%, and buffer sizes ranging from 0.2× to 2.2× the Bandwidth-Delay Product (BDP). For constant values, we use a 100 Mbps bandwidth, a base round-trip time (RTT) of 30 ms, a buffer size of 1 BDP, and no random loss.

The results, averaged over 10 trials, are shown in Figure 8[5]. From the results, we observe that LTP consistently maintains high link utilization and low latency inflation as it learns distinct policies in various network conditions, thanks to its use of distinguishable signals.

In contrast, heuristic-based algorithms such as BBR suffer from high queuing latency across network environments.

[5]The variance of the repeated results is within ±5%.

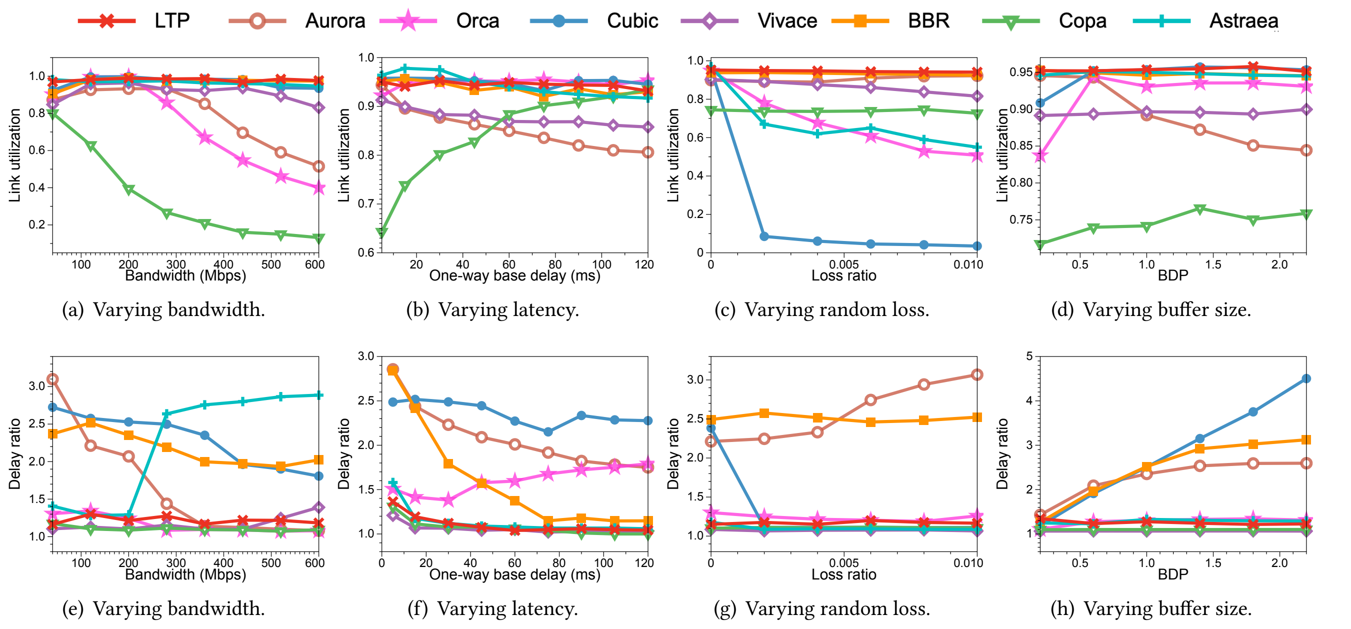


Figure 8. The performance of LTP and baselines in terms of link utilization and delay ratio (one-way delay normalized by link base delay) under various bandwidths, base delays, loss rates, and bottleneck buffer sizes.

Cubic underutilizes the link when the loss rate is high, as it relies on a hand-crafted rule to reduce the congestion window in response to packet loss events (see Figure 8(c)). The performance of previous DRL-based algorithms, such as Aurora and Orca, degrades significantly when the bandwidth or base delay exceeds the ranges seen during training. This is likely due to their limited training on links with large BDPs. Additionally, the online learning algorithm Vivace struggles with high latency scenarios (Figure 8(b)), as it relies on trial and error to adjust sending rates, resulting in slow responsiveness when RTT is high.

CCA League. Inspired by the concept of winning rate from [43], we further evaluate LTP and baseline schemes in both single-flow and multi-flow scenarios. We calculate each scheme's performance and fairness scores across varioue network conditions as follows. For performance, we calculate the score based on the Power metric and the loss rate:

$$\text{Perf-Score} = \frac{\text{throughput}}{\text{delay}} \times (1 - 40 \times \text{loss}).$$

For the fairness score in multi-flow scenarios, we adopt Jain's Fairness Index, calculated based on per-flow throughput:

$$\text{Jain Index} = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \times \sum_{i=1}^{n} x_i^2}$$

where x_i is the throughput of the i-th flow, and n is the total number of flows. A scheme is considered a winner if its score reaches at least 90% of the highest score in that scenario. We then compute each scheme's winning rate as the proportion of scenarios in which it is identified as a winner. These winning rates in terms of performance and fairness are used to derive the overall performance and fairness ranking.

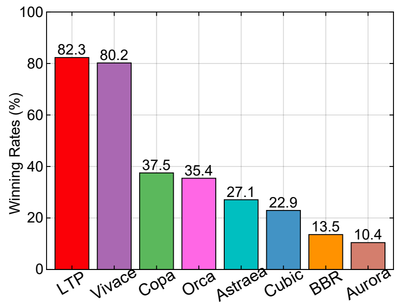
The experiments are repeated 1000 times with varying bandwidth, base delay, random loss rate, and number of flows in the training region. We show the winning rates of LTP and baselines in Figure 9[6]. We observe that LTP achieves outperforms all baselines in terms of performance in scenarios with one or multiple flows. Vivace also achieve high winning rates due to its online learning capability across various network conditions. On the other hand, BBR and Cubic exhibit low winning rates in terms of performance as we add loss penalty to the performance score. For fairness, although LTP does not rely on global or unnormalized features (e.g., $\frac{thr}{c}$ or raw throughput signals) that can directly support fair bandwidth allocation among competing flows, it still achieves strong fairness—only slightly behind BBR. This is enabled by our probing mechanism and the fairness metric (Equation 14) integrated into the reward function.

Fairness Showcase Figure 10 further illustrates the throughput dynamics of competing flows under different network conditions. As shown in Figures 10(a) and 10(b), LTP exhibits consistent fairness behaviors across various link capacities, delays, and loss rates.

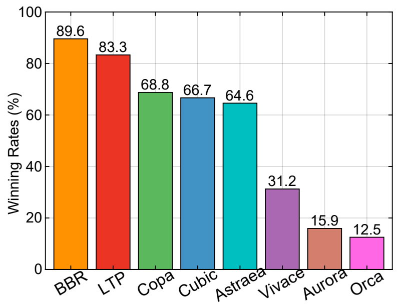
7.3.2 Friendliness.

To inspect the friendliness of LTP towards Cubic, we set up a 100Mbps link with a 15ms base delay. During the experiment, a LTP flow and a Cubic flow are run concurrently for 120 seconds, and their throughput ratio is recorded. We repeat the experiment across various base RTTs, where the buffer size is also tuned to be 1 BDP.

[6]Fairness match is conducted only with multi-flow scenarios.

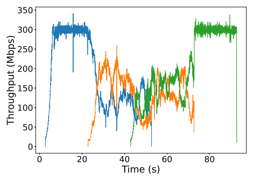


(a) Performance ranking.

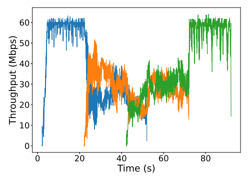


(b) Fairness ranking.

Figure 9. The ranking and winning rates of LTP and baselines.

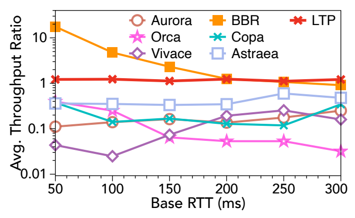


(a) 300Mbps, 30ms base RTT, 0% loss.

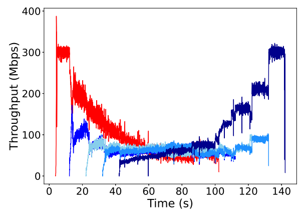


(b) 60Mbps, 120ms base RTT, 5% loss.

Figure 10. LTP's fairness illustration.

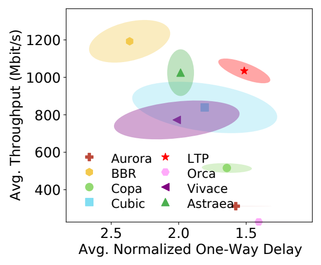


(a) LTP friendliness to Cubic.

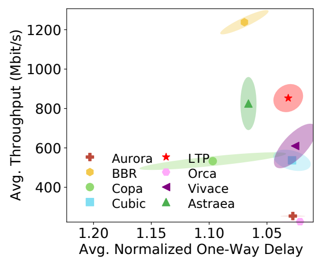


(b) LTP (Red) with 5 Cubic flows.

Figure 11. LTP friendliness.



(a) Intra-Continental Experiments



(b) Inter-Continental Experiments

Figure 12. Real-world experiments.

Figure 11(a) shows the results. We observe that compared to other learning-based solutions, LTP achieves a throughput ratio close to 1. We attribute this to our training process, where heterogeneous competing flows are introduced. Therefore, LTP will probe to find that it is competing with Cubic, and behave more aggressively. Figure 11(b) illustrates how LTP competes with 5 Cubic flows, achieving similar bandwidth.

7.3.3 Real-world Experiments. To further evaluate LTP's control performance, we conduct real-world experiments on the wild Internet. Specifically, we deploy the sender at AWS Seoul and place the receiver at AWS Singapore and London, testing LTP in both inter-continental and intra-continental scenarios. The bandwidths of both links are up to 10 Gbps. For each network condition, we run a flow for 60 seconds, repeating each trial 10 times and calculate overall average normalized throughput and one-way delay.

The evaluation results are presented in Figure 12. We observe that LTP sets a new benchmark in terms of high throughput and low latency. It achieves better link utilization and lower latency than most congestion control algorithms, including Cubic. This is primarily because LTP uses a DRL model to adjust the sending rate at a fine-grained level, allowing it to adapt rapidly to Internet bandwidth fluctuations without causing bufferbloat.

In contrast, while other learning-based algorithms Orca and Aurora perform well in emulated environments, they fail to achieve high utilization in real-world scenarios. We attribute this difference to the reward structure of these algorithms, which primarily focus on control performance. In contrast, our reward structure incorporates both probing and performance metrics, which encourages LTP to perform probing actions in wild and dynamic network conditions to gather network information, rather than blindly pursuing maximum utilization. This design choice accounts for LTP's superior performance among learning-based algorithms.

We also note that BBR achieves relatively high throughput in the inter-continental evaluation. This behavior, however, largely stems from BBR's specialized handling of rate policing at network edges, where it detects and models token-bucket policers to avoid excessive packet drops. Since our current work does not incorporate a handcrafted module for this mechanism, nor do our training and emulated environments include policers, BBR benefits disproportionately in such settings. We believe that extending LTP's training to encompass a wider range of network conditions, including various forms of rate policing, would enable it to learn similar resilience.

Further evaluation of LTP under challenging network conditions, including 10 Gbps high-speed networks and satellite networks, can be found in Appendix C. We find that though not trained over those network conditions, LTP's policy generalizes elegantly due to its well-normalized probe reward and input states.

7.4 Overhead

CPU Utilization. We evaluate the inference overhead of LTP and other CC baselines by measuring their CPU utilization during flow transmissions. Specifically, we emulate a link with 120 Mbps bandwidth, 20 ms RTT, and a buffer size

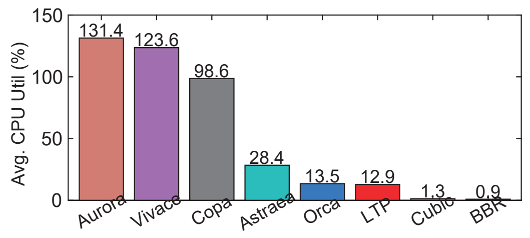


Figure 13. Average CPU utilization of LTP and CC baselines.

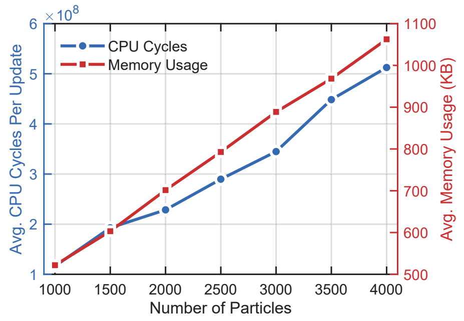


Figure 14. CPU cycles per update and memory usage across varying numbers of particles.

of 1 bandwidth-delay product (BDP). LTP is configured to infer a new sending rate every 30 ms. For each CC algorithm, we run the flow transmission 10 times for 60 seconds each and record the CPU utilization across all transmissions. The CPU utilization results are then averaged and presented in Figure 13. The results show that LTP achieves lower computational overhead than Orca, ranking just behind Cubic and BBR. This advantage mainly stems from its efficient C++ implementation. We note that the estimation module in §4 is functioning only in the offline training stage. Once the model is trained, the online, on-the-fly deployment phase only involves the DRL model, which receives signals and outputs control actions. We thus evaluate its overhead independently below.

Overhead of Estimation Module. To quantify the computational overhead introduced by LTP's particle update mechanism, we evaluate the module under different particle numbers. Figure 14 shows the CPU cycles required per update and the corresponding memory usage. Each configuration is repeated 10 times, and we report the average across all runs. The results show that both computation and storage overhead scale proportionally with the number of particles, while remaining within acceptable absolute values.

7.5 Under the Hood

In this section, we examine LTP's model by visualizing its traces to understand how it learns to probe the network status and enhance control performance. We present a trace with three homogeneous flows competing on a link with 120 Mbps bandwidth, 80 ms RTT, and 0% loss rate. For comparison, we define a baseline model—referred to as the Normal model—which uses the same state input, action, and RL training algorithm, but omits the estimation reward in the estimation module. The traces of the congestion window and RTT are plotted in Figure 15.

Our observations reveal that, compared to the Normal model, LTP (+Probe) learns to actively probe the network status by frequently applying small, aggressive oscillations to the congestion window and sending rate, as evidenced by rapid fluctuations in both CWND and RTT values. These fluctuations reflect LTP's efforts to explore the network's capacity and responsiveness. As noted in [36], when the link is fully utilized, the flow's sending rate and throughput variations provide insights into the flow's bandwidth occupancy. This suggests that LTP uses these small oscillations to probe both the link's bandwidth and competing flows' bandwidth.

8 Related Work

Congestion Control. Congestion control has been a long-standing research topic in networking. Traditional heuristic-based CC schemes [4–6, 13, 16, 39] rely on predefined rules, such as loss-based [13, 17] or delay-based approaches [4, 5]. Hybrid methods [11, 20, 35] attempt to combine multiple congestion signals but still require careful manual tuning, which limits their adaptability to varying network conditions. Recent research has leveraged machine learning for CC to replace manually crafted policies with data-driven control. Remy [39] precomputes an optimal mapping from network signals to actions, while PCC Allegro [7] and Vivace [8] use online learning for adaptive rate control. Aurora [18] applies deep reinforcement learning but faces challenges in fairness and overhead, which Orca [1] addresses by integrating classic CC schemes. Spine [38] employs hierarchical control for efficiency, and MOCC [24] incorporates multi-objective reinforcement learning to balance different application demands. None of them focus on the analysis and improvement of the signal engineering part of the learning process.

Estimate the Network state. Most existing CCAs focus on adjusting the sending rate based on observed signals but often overlook the accuracy of network condition estimation. Many approaches implicitly equate raw network signals (e.g., RTT, loss) with network status, disregarding noise and uncertainty in signal interpretation. Some methods attempt to refine signal processing, such as BBR's probeRTT phase [6], TCP's exponentially weighted moving average of RTT, range estimation in CCmatic [2] and Vivace's trial-and-error exploration [8]. However, these heuristic-based tricks lack a holistic framework for modeling network status transitions and the interactions between multiple signals.

9 Conclusion

In this paper, we highlight the importance of signal engineering in learning-based congestion control and propose Learn-to-Probe (LTP), a novel paradigm that improves both the distinguishability of input signals for DRL-based congestion

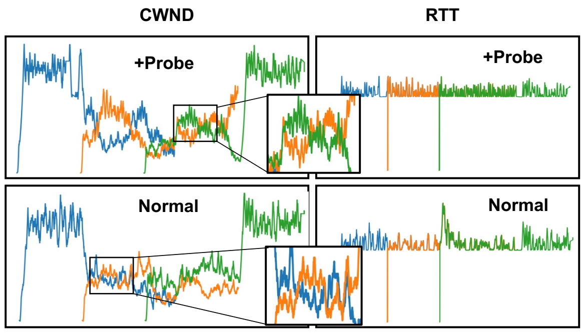


Figure 15. LTP learns to aggressively apply small oscillations to the CWND to probe the network status.

control algorithms. Our extensive evaluations demonstrate that LTP achieves consistently high performance across a diverse range of network environments, including real-world Internet deployments. Our findings suggest that explicitly designing signal engineering techniques can significantly enhance the effectiveness of DRL-based CCAs.

Acknowledgments

We thank the anonymous reviewers and our shepherd Shungeng Zhang for their constructive comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, NSFC 62402407, the TACC [40]. Kai Chen is the corresponding author.

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. Classic meets modern: a pragmatic learning-based congestion control for the internet. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication.* 632–647.

[2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2024. Towards provably performant congestion control. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 951–978. https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup

[3] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3452296.3472912

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

[5] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. 1994. *TCP Vegas: New techniques for congestion detection and avoidance.* Number 4. ACM.

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

[7] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 395–408.

[8] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 343–356.

[9] Jos Elfring, Elena Torta, and René Van De Molengraft. 2021. Particle filters: A hands-on tutorial. *Sensors* 21, 2 (2021), 438.

[10] DI engine Contributors. 2021. DI-engine: OpenDILab Decision Intelligence Engine. https://github.com/opendilab/DI-engine. (2021).

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

[12] Scott Fujimoto, Herke Hoof, and David Meger. 2018. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*. PMLR, 1587–1596.

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

[14] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. 2020. {LinnOS}: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 173–190.

[15] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th international conference on network protocols (ICNP)*. IEEE, 1–10.

[16] Jinbin Hu, Shuying Rao, Min Zhu, Jiawei Huang, Jianxin Wang, and Jin Wang. 2025. SRCC: Sub-RTT Congestion Control for Lossless Datacenter Networks. *IEEE Transactions on Industrial Informatics* 21, 4 (2025), 2799–2808. https://doi.org/10.1109/TII.2024.3495759

[17] Van Jacobson. 1988. Congestion avoidance and control. *ACM SIGCOMM computer communication review* 18, 4 (1988), 314–329.

[18] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *International Conference on Machine Learning ICML*. 3050–3059.

[19] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. 2018. Recurrent experience replay in distributed reinforcement learning. In *International conference on learning representations.*

[20] Ryan King, Richard Baraniuk, and Rudolf Riedi. 2005. TCP-Africa: An adaptive and fair rapid increase rule for scalable TCP. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 3. IEEE, 1838–1848.

[21] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[22] Xudong Liao, Han Tian, Chaoliang Zeng, Xinchen Wan, and Kai Chen. 2024. Astraea: Towards Fair and Efficient Learning-based Congestion Control. In *Proceedings of the Nineteenth European Conference on Computer Systems.* 99–114.

[23] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[24] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. 2022. Multi-objective congestion control. In *Proceedings of the Seventeenth European Conference on Computer Systems.* 218–235.

[25] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* 197–210.

[26] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[27] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1928–1937.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

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

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[31] J Salim, H Khosravi, Andi Kleen, and Alexey Kuznetsov. 2003. *Linux netlink as an ip services protocol.* Technical Report.

[32] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

[33] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Eric P. Xing and Tony Jebara (Eds.), Vol. 32. PMLR, Bejing, China, 387–395. https://proceedings.mlr.press/v32/silver14.html

[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

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

[36] The Google BBR Team. [n. d.]. BBR bandwidth based convergence. ([n. d.]). https://github.com/google/bbr/blob/master/Documentation/bbr_bandwidth_based_convergence.pdf.

[37] Han Tian, Xudong Liao, Decang Sun, Chaoliang Zeng, Yilun Jin, Junxue Zhang, Xinchen Wan, Zilong Wang, Yong Wang, and Kai Chen. 2025. Achieving Fairness Generalizability for Learning-based Congestion Control with Jury. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 413–427. https://doi.org/10.1145/3689031.3696065

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

[39] Keith Winstein and Hari Balakrishnan. 2013. TCP Ex Machina: Computer-Generated Congestion Control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/2486001.2486020

[40] Kaiqiang Xu, Decang Sun, Hao Wang, Zhenghang Ren, Xinchen Wan, Xudong Liao, Zilong Wang, Junxue Zhang, and Kai Chen. 2025. Design and Operation of Shared Machine Learning Clusters on Campus. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 295–310. https://doi.org/10.1145/3669940.3707266

[41] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[42] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 384–397. https://doi.org/10.1145/3452296.3472927

[43] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. 2023. Computers Can Learn from the Heuristic Designs and Master Internet Congestion Control. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*.

[44] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. 2022. Liteflow: towards high-performance adaptive neural networks for kernel datapath. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 414–427.

Appendix

A Bayesian Filtering Perspective of LTP

The transition estimation process in LTP is rooted in the concept of Bayesian filtering. Bayesian filtering is a general approach for estimating the state of a dynamic system in the presence of noisy observations. It is based on Bayesian inference, which combines prior knowledge about the system's evolution over time with the observed measurements that provide information about the current state. From a Bayesian filtering viewpoint, our state estimation problem is essentially a recursive signal estimation task. The objective is to track the hidden state sequence s_t of the network system, given the observed signals z_t. This process relies on two key components: the probabilistic model of the system's dynamics, $p(s_t|s_{t-1})$, which encodes prior knowledge about how the network state evolves, and the signal measurement model, $p(z_t|s_t)$, which describes how the signals are measured based on the current state of the system.

In this framework, the state estimate is represented as a probability density function (pdf), which not only quantifies the estimated state but also captures the uncertainty associated with it. Specifically, we refer to this estimate as the posterior state — the conditional probability distribution that represents the system's state after incorporating historical signal measurements. Our goal is to compute the posterior probability distribution of the network state at time t, conditioned on all available measurements up to that point:

$$p(s_t|z(0:t)), \tag{16}$$

where $z(t_1 : t_2)$ denotes the historical signal measurements between time t_1 and t_2, and $z(: t)$ represents the complete history of signal measurements up to time t. The agent's task is to estimate the network state based on the observed signals. This posterior probability distribution can be computed recursively as follows:

At time t_2, the method first updates the posterior distribution using the prior estimate from t_1 and the network's transition model:

$$p\left(s(t_2) \mid z(:t_1)\right) = \int p\left(s(t_2) \mid s(t_1)\right) p\left(s(t_1) \mid z(:t_1)\right)\, ds(t_1) \tag{17}$$

In this equation, $p(s(t_2) \mid s(t_1))$ is the network transition model, which encodes prior knowledge about how the system state evolves. The term $p(s(t_2) \mid z(: t_1))$ represents the prior state estimate at time t_2, which is based on the previous estimate and the system's transition behavior.

Next, using Bayes' theorem, we update the posterior distribution by incorporating the newly observed signals $z(t_2)$:

$$p\left(s(t_2) \mid z(0:t_2)\right) = \frac{p\left(z(t_2) \mid s(t_2)\right) p\left(s(t_2) \mid z(:t_1)\right)}{p\left(z(t_2) \mid z(:t_1)\right)}, \tag{18}$$

where $p(z(t_2) \mid s(t_2))$ is the signal measurement model, which describes the likelihood of observing the signal $z(t_2)$ given the network state $s(t_2)$. The denominator, $p(z(t_2) \mid z(: t_1))$, is the normalizing constant, which represents the probability of the observed measurement. It can be computed as follows:

$$p\left(z(t_2) \mid z(:t_1)\right) = \int p\left(z(t_2) \mid s(t_2)\right) p\left(s(t_2) \mid z(:t_1)\right)\, ds(t_2) \tag{19}$$

Thus, Bayesian filtering provides a systematic way to combine prior knowledge of the system's dynamics with new measurements, enabling an accurate estimation of the system's state over time.

Particle Filtering The integrals in the Bayesian filtering process can only be solved analytically under specific conditions, such as finite-dimensional discrete state variables or linear models with Gaussian pdfs. However, to avoid these restrictive assumptions, particle filtering provides an alternative by approximating the posterior pdf using a discrete set of samples, which allows minimal restrictions on the models involved. The optimal Bayesian solution is then approximated as a sum of weighted samples:

$$p\left(s_{0:t} \mid z_{1:t}\right) \approx \sum_{i=1}^{N_s} w_t^i \delta\left(s_{0:t} - s_{0:t}^i\right) \tag{20}$$

Here, $\left\{w_t^i, s_{0:t}^i\right\}_{i=1}^{N_s}$ represents a set of N_s particles, where each sample $s_{0:t}^i$ corresponds to a potential realization of the state sequence. The weight w_t^i indicates the relative importance of each particle $s_{0:t}^i$, with the normalization condition $\sum_{i=1}^{N_s} w_t^i = 1$. Particles associated with higher weights are considered to be closer to the true state, while those with lower weights represent less likely state realizations. The Dirac delta function $\delta(\cdot)$ is used to represent this discrete approximation of the posterior.

By approximating the continuous pdf with a set of discrete samples, particle filtering turns intractable integrals into summations over these N_s particles. This method is advantageous because it allows the posterior to be represented by arbitrarily shaped pdfs (assuming sufficient samples), and it imposes minimal restrictions on both the process and measurement models. These advantages explain why particle filtering is widely used in many applications.

However, a key challenge in particle filtering is that the posterior pdf itself is unknown, meaning direct sampling from it is impossible. Instead, samples must be drawn from an alternative distribution, called the importance density (or proposal density), denoted q. The weights are then used to compensate for the fact that the samples are drawn from this importance density rather than directly from the posterior pdf. The importance sampling formula is given by:

$$w_t^i \propto \frac{p\left(s_{0:t}^i \mid z_{1:t}\right)}{q\left(s_{0:t}^i \mid z_{1:t}\right)} \tag{21}$$

Given the importance density q and the fact that the true posterior is unknown, the weight update rule becomes:

$$w_t^i \propto w_{t-1}^i \frac{p\left(z_t \mid s_t^i\right) p\left(s_t^i \mid s_{t-1}^i\right)}{q\left(s_t^i \mid s_{t-1}^i, z_t\right)} \tag{22}$$

Finally, the posterior estimate of the state at time t is approximated by:

$$p\left(s_t \mid z_{1:t}\right) \approx \sum_{i=1}^{N_s} w_t^i \delta\left(s_t - s_t^i\right) \tag{23}$$

As the number of particles, N_s, increases, this estimate converges to the true posterior distribution.

While the importance sampling method is fundamental to particle filtering, it does come with a potential issue: as the weights evolve over time, the variance of the weights can grow. This leads to a situation called *weight degeneracy*, where most of the weights become very small, and only a few particles significantly contribute to the posterior estimate. This can cause the particle filter to become inefficient.

To mitigate weight degeneracy, a resampling step is generally introduced. In resampling, particles with higher weights are duplicated, while those with lower weights are discarded. After resampling, the weights are typically reset to a uniform value of $1/N_s$. This ensures that the particle population is concentrated in regions with higher posterior probability, improving the efficiency of the particle filter and making the state estimate more accurate.

B RL Training Algorithm of LTP

We employ the Deep Deterministic Policy Gradient (DDPG) algorithm [23], a well-known model-free off-policy reinforcement learning (RL) method to train sub-policies. The RL agent updates the parameters of our recurrent neural networks (RNNs) to refine the mapping from packet statistics to actions, with the goal of maximizing the accumulated rewards.

DDPG follows the actor-critic framework, involving two key models: the actor model and the critic model. Both models receive the current network state as input. The actor model represents the policy, i.e., it outputs actions, while the critic model evaluates the potential reward associated with a given state-action pair. The role of the critic model is to provide a value estimate that helps guide the actor model's learning process, as depicted in Figure 16. More specifically, the critic model estimates the action-value function $Q^{\pi_\theta}(s,a) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t | a, s]$, which predicts the cumulative reward an agent would collect after taking action a in state s, following the policy π_θ. Once the critic model is trained, the actor model uses the critic model's feedback to select the best action that maximize the expected reward. The use of a critic model helps mitigate the variance in reward estimates, making the learning process more stable. Both the actor and the critic are implemented using RNNs.

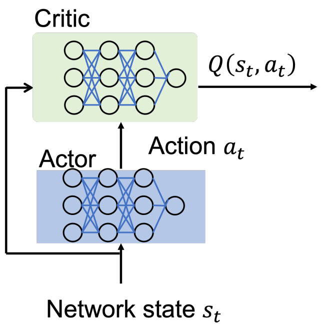


Figure 16. Actor-critic training algorithm in LTP.

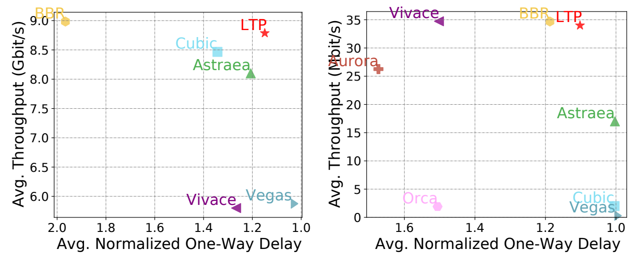
To make full use of the hidden states in the RNN structure, we incorporate strategies from [19]. Specifically, we store the hidden state throughout the training trajectory, which is later used to initialize the policy model during training. This ensures that historical context is incorporated when the policy model is updated. Furthermore, to stabilize the learning process, we include a 'burn-in' phase at the start of each sequence. During this phase, an additional segment of the trajectory is used exclusively for network forwarding to establish a stable hidden state before the training starts. These strategies help address the challenges associated with training RNN-based models and improve the overall training effectiveness.

The training procedure proceeds as follows: initially, the actor interacts with the environment and collects trajectories of state-action-reward sequences, denoted as (s, a, r, h), where h represents the hidden state from the RNN. For simplicity, we describe the process with a single trajectory, though in practice, we train in batches. In each training step, we sample a long trajectory that includes a burn-in prefix, such as $(s_{-b}, a_{-b}, r_{-b}, h_{-b})$ through (s_N, a_N, r_N, h_N), where b denotes the burn-in steps. We initialize our recurrent model with the first hidden state h_{-b} at the head of the trajectory and forward it to step 0 to get a warmed hidden state. Then, we unroll the recurrent network model on the trajectory from step 0 to the end to get the model outputs of s_{N-1} and s_N, which is the target state we will train on for this sample. We denote the sequence (s, a, r, s', a') as $(s_{N-1}, a_{N-1}, r_{N-1}, s_N, a_N)$ for convenience in the subsequent objective functions.

LTP updates the actor's policy π_θ by minimizing the following objective function:

$$\mathcal{J}(\theta) = \mathbb{E}\left[Q_\omega\left(s, \pi_\theta(s)\right)\right], \tag{24}$$

where $Q_\omega(s,a)$ represents the output of the critic that estimates the action-value function $Q^{\pi_\theta}(s,a)$ under the current policy. To optimize the policy distribution π_θ, we apply



(a) High-speed network. (b) Satellite network.

Figure 17. LTP in challenging network conditions.

the policy gradient theorem [33]. To accurately estimate the action-value function, the critic is trained using temporal difference learning [34], with the following objective function:

$$\mathcal{L}(\omega) = \mathbb{E}_{s,a,r,s'}\left[\left(Q_\omega(s,a) - r + \gamma Q_\omega(s',a')|_{a'=\pi_\theta(s')}\right)^2\right]. \tag{25}$$

For optimization, we use the Adam optimizer [21] with gradient-based updates. The back-propagation is truncated before the burn-in steps (i.e., until step 0). After calculating the gradients, the actor and critic models are updated with learning rates α and η, respectively, as follows:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{J}(\theta), \quad \omega \leftarrow \omega - \eta \nabla_\omega \mathcal{L}(\omega). \tag{26}$$

Additionally, LTP adopts several RL-related techniques from the TD3 algorithm [12], including clipped double Q-learning, delayed policy updates, and target policy smoothing regularization. These techniques help reduce the variance in the critic's estimates. For a detailed explanation of these techniques, we refer the readers to [12].

C LTP on Challenging Conditions

We further test LTP in following challenging network conditions to evaluate its performance generalizability:

High-speed Networks To evaluate the performance of LTP in high-speed network environments, we establish a connection between two end-hosts in our cluster, connected via a Mellanox SN2700 100G switch. To emulate a real-world high-speed WAN scenario, we limit the receiver's bandwidth to 10Gbps and introduce an additional one-way latency of 15ms, using traffic control (tc) for network emulation. We exclude previous learning-based schemes such as Aurora, Orca, and Vivace as they are constrained to a maximum bandwidth of 1Gbps due to inefficient implementation.

The throughput and latency results for LTP and other baseline algorithms are shown in Figure 17(a). We observe that LTP achieves throughput comparable to that of BBR, while maintaining lower latency than Cubic. These results highlight LTP's ability to generalize its performance effectively in high-speed network conditions, where the available bandwidth significantly exceeds the training region.

Name	Value
control time interval	30 ms
learning rate	0.005
discount factor (γ)	0.98
batch size	64
particle number	1000
model update frequency (per episode)	100
signal gap sensitivity ($\omega_q, \omega_{\overline{x}}, \omega_{\overline{x_o}}$)	50,10,10
other flow gradient range O_j	1.5
delay jitter range D_j	0.02
bandwidth jitter range C_j	0.05
loss jitter range L_j	0.001
estimation reward coefficient $\beta_q, \beta_{\overline{x}}, \beta_{\overline{x_o}}$	10, 0.2, 0.2
loss coefficient ζ	5
fairness coefficient $\beta_{fairness}$	1
latency tolerance β_{lat}	1.15
estimation weight β_{esti}	0.2
performance weight β_{perf}	1

Table 4. Training hyperparameters in LTP.

Satellite Networks In satellite communication, the long round-trip time (RTT) and random packet loss pose significant challenges for traditional congestion control schemes. To assess LTP in such a scenario, we conduct experiments on a simulated satellite link following the setup from [8], with a 42 Mbps bandwidth, 800ms RTT, and a 0.74% random loss rate. The results, averaged over 10 trials, are presented in Figure 17(b). We observe that by achieving both high throughput and low latency, LTP establishes a new point on the Pareto frontier. This performance is attributed to LTP's use of normalized features, particularly the ΔRTT and the normalized loss rate, which provide resilience against variations in base delay and random packet loss.

D Training Details

In order to stabilize the training process, LTP adopts episode-based training. Each episode consists of two phases: the collecting phase and the learning phase. In the collecting phase, several distributed collectors perceive states and enforce actions in various training environments to collect experiences for a period of time (100 seconds), which are stored in a data structure called replay memory [28]. In the learning phase, the centralized learner samples experiences from the replay memory for training. As a result, the learner updates the model only after the collectors have collected enough new experiences with sufficient length, and the experiences collected in one episode are from the static RL agent with no update. Therefore, the learning process will be less stochastic, and the model will converge faster. We use 4 actors to collect the training experience in parallel.

The training hyperparameters of LTP are given in Table 4.