PolicyCache: Intra-flow Learning in Congestion Control

Han Tian[1], Han Wang[1], Wenbo Li[1], Xudong Liao[2], Decang Sun[2], Wenxue Li[2]
Donghui Chen[3], Bin Huang[3], Senbo Fu[3], Junxue Zhang[1], Dian Shen[4], Kai Chen[2]
[1]USTC [2]iSING Lab, HKUST [3]Huawei [4]Southeast University

Abstract

TCP congestion control (CC) schemes must balance fast responsiveness, adaptability to diverse network conditions, and low computational overhead. Existing approaches fall short: heuristic-based algorithms are lightweight but brittle, learning-based schemes provide high responsiveness yet struggle with generalization, and exploration-based methods adapt well but converge slowly. We present PolicyCache, the first CC algorithm based on *intra-flow learning*, where both training and execution of the policy are confined to a single flow. Unlike prior inter-flow learning, this paradigm avoids cross-environment generalization pitfalls while maintaining high responsiveness. PolicyCache leverages a lightweight, non-parametric tree-based model coupled with online exploration and dynamic model switching to enable rapid and robust adaptation. We provide convergence analysis of PolicyCache and have built a fully functional Linux prototype. Extensive evaluations demonstrate that PolicyCache consistently achieves high throughput, low latency, and fairness across diverse emulated and real-world networks, while incurring minimal overhead. These results establish intra-flow learning as a practical and effective new direction for congestion control.

1 Introduction

With decades of development in TCP congestion control (CC), researchers have proposed numerous variants to accommodate diverse network architectures and path characteristics. For instance, schemes have been designed for highly variable cellular networks [28,35,40] as well as ultra-high-bandwidth, low-latency data center environments [3,20,24,33,36]. Classic TCP variants such as Reno [18], NewReno [14], Cubic [16], and Vegas [7] further exemplify how different congestion signals, such as packet loss or delay inflation, have been exploited over time. While these approaches achieve strong performance in their targeted settings, they often lack adaptability and degrade significantly under other conditions. Consequently, the pursuit of a universal congestion control solution has largely given way to specialized, scenario-specific designs.

To address the adaptability limitations of TCP congestion control, recent efforts focus on designing CC algorithms that perform consistently across diverse and complex real-world network conditions. Broadly, two categories of methods have been proposed:

- **Learning-based methods** [2,19,21,31,32] adopt a data-driven approach. They collect training trajectories to guide optimal sending rate adjustments given observed network signals and use these trajectories to train a model (typically a multi-layer neural network) that approximates the signal-to-action mapping.
- **Exploration-based methods** [10,11] rely on online trial-and-error to probe the best rate adjustment in the current network conditions, making minimal assumptions about the network.

While these approaches have shown promising results under controlled settings, they suffer from inherent limitations, as validated in our motivation experiments (§2). Learning-based methods often face *generalizability issues*: a policy learned in one network environment may perform perfectly there but unpredictably in unseen scenarios [21,31]. In contrast, exploration-based methods achieve decent performance across various networks but suffer from low responsiveness, performing poorly in highly dynamic networks with high RTT. Table 1 summarizes representative methods along with their characteristics and limitations.

The core challenge lies in how existing methods handle learning: where to learn from and where to apply the learned policy. Most prior learning-based methods follow a paradigm we term *inter-flow learning*, applying a policy learned from one set of flows to others. This assumes the learned state-action mapping is consistent across flows, which often fails in real networks due to variations in link characteristics, bottleneck locations, and in-network mechanisms such as rate limiters, trace shapers, ACK aggregation, and delayed ACKs, making generalization difficult. In contrast, exploration-based methods rely solely on online probing, making minimal assumptions about network conditions[1]. However, this comes at the cost of long control loops: each probe requires several RTTs to generate the next action, resulting in slow convergence, especially in high-RTT or highly dynamic networks.

Therefore, we pose the following question: *How can we leverage the proactive responsiveness of learning-based methods while avoiding their generalizability issues?* Addressing this question requires a new learning paradigm that relies on a weaker assumption.

From Inter-flow to Intra-flow Learning. We propose *intra-flow learning*, a paradigm in which both the learning and

[1]Or equivalently, they assume a probed action remains valid for a few RTTs.

execution of the learned policy are confined to a single flow (and its corresponding path). This design rests on a weaker assumption: the learned policy remains effective within a given flow under its current network conditions. Intra-flow learning operates as follows: when a new flow starts, no prior policy is assumed. The flow collects state-action pairs[2] in real time as training data, immediately updates the learning model, and, once the policy is sufficiently learned, uses it to guide subsequent transmissions within the same flow. By solely relying on trajectory data from the current flow, intra-flow learning reduces the need for cross-environment generalization while achieving consistent performance across diverse network conditions with strong responsiveness.

Challenges: Transitioning from inter-flow to intra-flow learning is non-trivial for two main reasons: i) Existing learning-based CCAs typically rely on parametric models (e.g., deep neural networks) trained via stochastic gradient descent (SGD). Such methods require thousands of samples and hours of training to converge on new data, often exceeding the lifetime of a flow, making intra-flow learning impractical; ii) The computational overhead of online SGD prevents its use on devices with constrained resources or in high-frequency control scenarios.

Our approach: To overcome these challenges, we present POLICYCACHE, the first congestion control algorithm based on intra-flow learning, designed for high responsiveness, strong adaptability, and low overhead. Its core advantages arise from the synergy of two key components:

- **Non-parametric online learning:** Instead of heavy parametric models, POLICYCACHE employs an online tree-based structure for lightweight stream learning and control. Non-parametric tree models learn by memorizing observed samples, enabling immediate use of collected patterns and eliminating long convergence times (§3.2). Experiments show that control rules are quickly learned within the first few RTTs and deployed for the remainder of the flow, enabling rapid adaptation (§6.4).
- **Explore-Learn-Execute procedure:** POLICYCACHE integrates stream learning (§3.2), online exploration (§3.3), sample collection (§3.4), and model execution (§3.5) to achieve robust and adaptive control. It operates in two modes: i) *backup exploration mode* probes good control directions in a slow loop; and ii) *model execution mode* feeds network signals into the tree-based model and executes the resulting control actions in a fast loop. POLICYCACHE continuously evaluates the model performance, switching between modes as needed. This ensures a performance lower bound during early learning while enabling highly responsive control once the policy is learned (§6).

A key challenge in switching between a learned model and online exploration is ensuring fairness among flows that operate with different ratios of exploration and execution. Previous work [10, 11, 23] focus on convergence of pure exploration methods using complex social concave games [13], often assuming that network conditions remain unchanged during exploration. In contrast, we reconstruct the convergence analysis under a simpler probabilistic model that accounts for multiple flows exploring concurrently to improve pre-defined utilities. Our convergence proof holds under asynchronous exploration, ensuring both RTT-fairness and fairness in mixed inference modes (§4).

We implement a fully functional prototype of POLICYCACHE in Linux and conduct extensive evaluations (§5). Results show that POLICYCACHE achieves broader adaptability than both learning-based and heuristic approaches, and adapts more quickly than exploration-based schemes (§6.1). It also demonstrates the most stable convergence among baselines, while ensuring RTT fairness and maintaining good friendliness (§6.2). Furthermore, POLICYCACHE incurs low computational overhead (§6.3) and we offer intuitive interpretability of its operation (§6.4).

[2] The state-action pair consists of the observed network signals and the corresponding recommended control action.

CC Algorithm	Responsiveness	Adaptability	Overhead
Rule-based [7, 8, 16]	✓	✗	✓
DRL-based [21, 31, 32]	✓	✗	✗
Exploration-based [10, 11]	✗	✓	✓
POLICYCACHE	✓	✓	✓

Table 1: Comparison of congestion control algorithms across different categories.

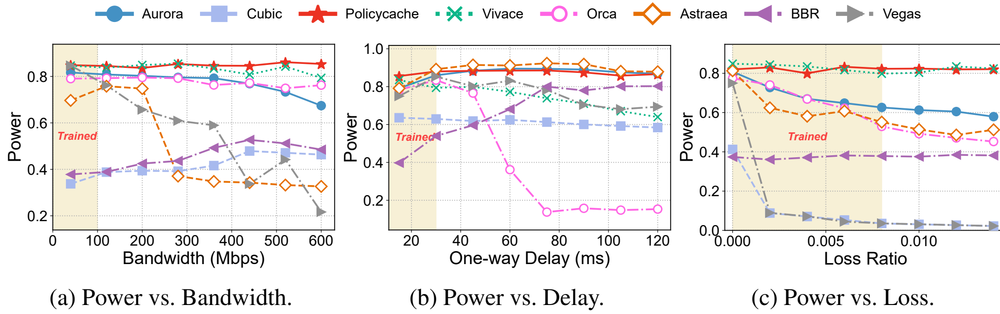
2 Background and Motivation

2.1 Existing Congestion Control Schemes

Internet congestion control (CC) regulates sender rates to prevent queue buildup, loss, and latency inflation. An effective CC should satisfy three key properties: (i) fast responsiveness, (ii) robust adaptation, and (iii) low overhead. We categorize existing approaches into three main lines and summarize their strengths and limitations in Table 1.

Heuristic-based CC: Classic rule-based algorithms, such as CUBIC [16], BBR [8], and Vegas [7], adjust the sending rate based on hand-crafted rules mapping network events (e.g., loss, delay) to window changes. These schemes are simple and lightweight, achieving low overhead and immediate responsiveness (good iii and i). However, they are often brittle across diverse network conditions (poor ii). For instance, CUBIC can create persistent queues on large buffers, while Reno underperforms on high-BDP/long-RTT paths and overreacts to non-congestion wireless losses [18, 27].

Exploration-based CC: Exploration-based schemes such as Allegro, Vivace and Proteus [10, 11, 23] eschew fixed rules



(a) Power vs. Bandwidth.

(b) Power vs. Delay.

(c) Power vs. Loss.

Figure 1: The control performance of CCAs varies with different network conditions.

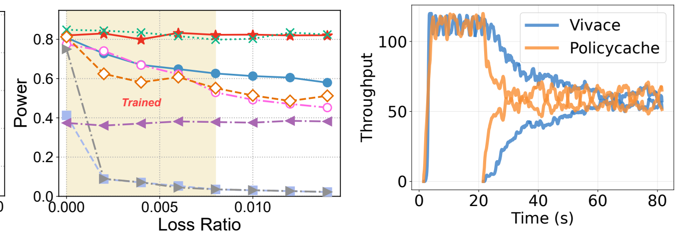


Figure 2: Convergence responsiveness.

and instead probe the network by perturbing rates over short intervals and comparing observed utilities. This trial-and-error design provides strong adaptability across diverse conditions (good ii) and maintains low overhead (good iii). Yet, exploration-based methods react slowly, as each decision requires multiple RTTs to evaluate, resulting in weak responsiveness (poor i) in high-latency or rapidly changing networks.

Learning-based CC: Learning-based approaches, e.g., Aurora [19], Spine [32], Astraea [21], and Orca [2], learn policies that map rich state vectors to rate control actions. They can capture complex interactions in multi-flow and dynamic networks, yielding good responsiveness (good i). However, generalization remains challenging: policies trained on specific network conditions often fail under unseen environments (partial ii) [31], and the computational cost of online inference and offline training is non-trivial (poor iii), limiting deployment on real-time or resource-constrained devices.

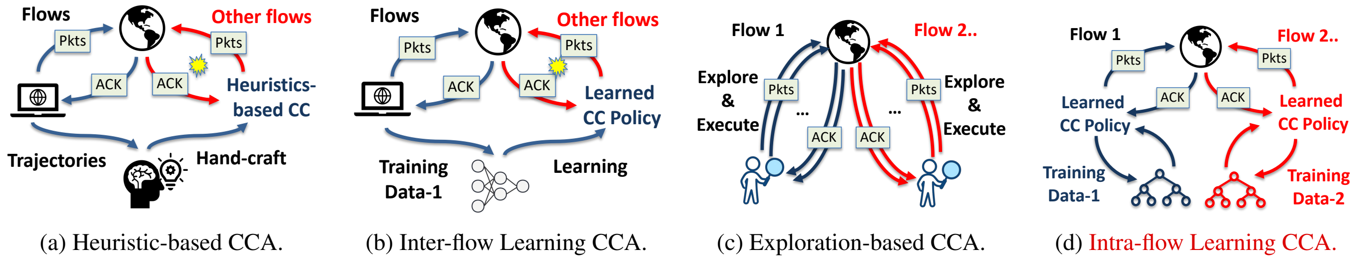
2.2 Control Responsiveness vs. Adaptability

To validate the limitations of three representative lines of CC approaches, we evaluate them in a uniform emulation environment: heuristic-based (CUBIC, Vegas), exploration-based (Vivace), and DRL-based (Aurora, Astraea, Orca). Using a dumbbell topology with a single long-lived flow, we vary link characteristics—capacity and one-way base delay—under three one-dimensional sweeps: (i) fixing capacity at 120 Mbps, random loss at 0, and varying one-way delay from 15–120 ms; (ii) fixing one-way delay at 15 ms, random loss at 0, and varying capacity from 40-600 Mbps; and (iii) fixing capacity at 120 Mbps, one-way delay at 15 ms, and varying random loss from 0-1.5%. Unless stated otherwise, the queue length is set to $1\times$BDP. DRL models were trained within 0-100 Mbps capacity, 0-30 ms one-way delay, and 0-0.8% random loss (the shaded "Trained" region in the figures). The evaluation metric is *power* = link utilization / real-time one-way delay, where higher is better.

Results in Fig. 1 show that both rule-based and learning-based approaches suffer from adaptation limitations. CUBIC's power remains low across the entire range and degrades further with random loss, due to its fixed signal-to-action mapping (e.g., shrinking cwnd upon loss). Vegas, on the other hand, performs reasonably at low speed but collapses as link bandwidth increases, because Vegas only maintains a small number of queuing packets. As the relative delay signal becomes vanishingly small compared to the number of in-flight packets at high capacity, Vegas becomes overly sensitive to delay variations and stabilizes at a much lower throughput than the link capacity. Learning-based methods (Aurora, Orca, Astraea) perform well within their training region but show directional generalization: they extrapolate reasonably along some dimensions but degrade sharply in others. This is consistent with well-known generalization issues in deep learning: once network conditions drift outside the training distribution, the learned policy's behavior becomes unpredictable or ineffective.

We argue that the root cause lies in the *inter-flow learning* paradigm, where policies are learned/handcrafted based on one set of flows and directly applied to new flows in different paths and environments. This paradigm implicitly assumes that the state-action mapping, i.e., the input network state and the corresponding optimal rate adjustment, remains consistent across environments, an assumption that does not hold in practice. Simply enlarging the training region is impractical: real-world networks involve highly dynamic link characteristics, heterogeneous topologies, and in-network mechanisms such as rate limiters, traffic shapers, ACK aggregation, and delayed ACKs. These dynamics are often hidden from the sender side, making them impossible to capture through offline training alone.

Exploration-based methods (e.g., Vivace) adopt a different philosophy: they probe the network online via trial-and-error. By incrementally increasing or decreasing the sending rate and observing responses, they discover control actions without assuming any prior mapping. This approach provides robustness across diverse network conditions, as shown in Fig. 1. However, the cost of discarding the learning process is a slower control loop: probing takes several RTTs, delaying reaction to changes. Consequently, exploration-based methods converge slowly, especially under high-RTT or rapidly varying conditions. As seen in Fig. 1, Vivace's performance gradually degrades with increasing RTT. To further highlight its responsiveness limitation, we conduct a two-flow staggered-start experiment (capacity 120 Mbps, one-way delay 60 ms,



(a) Heuristic-based CCA. (b) Inter-flow Learning CCA. (c) Exploration-based CCA. (d) Intra-flow Learning CCA.

Figure 3: An illustration of existing CCAs and intra-flow learning paradigm. Heuristic-based (3a) and inter-flow learning CCAs (3b) apply hand-crafted/ learned policies to new flows, leading to risks of poor generalization, denoted by the yellow starbursts. Exploration-based methods (3c) suffer from slow responsiveness due to long control loops. Intra-flow learning approach (3d) confines the learning process within flow, thereby avoiding generalization issues while maintaining high responsiveness.

no loss, queue length 1×BDP). As shown in Fig. 2, PolicyCache quickly redistributes bandwidth and reaches fairness, while Vivace requires a long transition before flows converge, underscoring its sluggish responsiveness.

2.3 Intra-flow Learning

The above discussion motivates a central question: *How can we harness the proactive responsiveness of learning-based methods while avoiding their generalization pitfalls?* We argue that the key lies in a more relaxed assumption. Empirically, learning-based schemes perform well within their training region, implying that a policy learned under a given condition is effective as long as it is applied within that same condition (Fig. 1). This insight naturally leads to our proposal: *intra-flow learning*, where each flow learns its own policy from its own observations and applies it only to itself.

Intra-flow Learning We propose *intra-flow learning*, a paradigm where both the training and execution of the congestion control (CC) policy are confined to a single flow along its current path. The key assumption is weaker than traditional cross-flow generalization: as long as the state–action mapping remains valid within a flow's own conditions, the learned policy can be effective.

In practice, when a new flow starts, no prior policy is assumed. The flow incrementally collects state–action pairs, updates the model online, and once the policy is sufficiently learned, applies it to guide subsequent transmissions. This enables each flow to drive its own CC decisions directly from its observations, without relying on pre-trained models or cross-environment adaptation. As a result, intra-flow learning provides consistent and responsive performance across diverse network conditions (Fig. 3). Additionally, rather than reacting to instantaneous noisy measurements in exploration-based schemes, intra-flow learning learns statistically from accumulated state-action tuples. This averaging effect dampens the influence of measurement noise and enables stable policy execution.

However, enabling intra-flow learning is challenging. Existing learning-based CCAs often rely on deep reinforcement

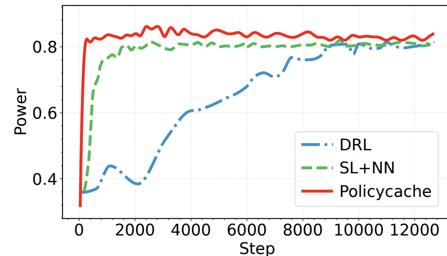


Figure 4: Power vs. update steps.

learning (DRL), which requires hours of training and thousands of update steps to adapt—far longer than the lifetime of most flows. To illustrate this limitation, we evaluate three designs under the same environment: (i) a DRL agent, representative of prior work; (ii) our tree-based system POLICYCACHE; and (iii) a variant of POLICYCACHE that substitutes the tree model with a neural network (SL+NN). The experiments are conducted in a dumbbell topology with 120 Mbps link capacity and 60 ms one-way delay. We launch three concurrent long-lived flows, each governed by a different controller. The evaluation metric is power (link utilization divided by real-time one-way delay).

As shown in Fig. 4, POLICYCACHE converges within only a few update steps, while both the DRL agent and the NN model adapt much more slowly, leaving flows underutilized for a significant portion of their duration. Their slow convergence originates from the reliance on *parametric models*: Neural networks, optimized via stochastic gradient descent (SGD), need a large number of training samples and iterative updates, incurring high online overhead. Such constraints make parametric approaches ill-suited for intra-flow learning, where responsiveness and efficiency are critical.

Non-parametric vs. Parametric Models To overcome these limitations, POLICYCACHE adopts a lightweight *non-parametric* design based on online tree structures. Parametric models, such as DNNs, assume a fixed structure with a predetermined number of parameters. Their learning involves repeatedly tuning these parameters with SGD, which not only requires significant training time but also slows adaptation to new conditions. In contrast, non-parametric models such as decision trees effectively "memorize" observed patterns, al-

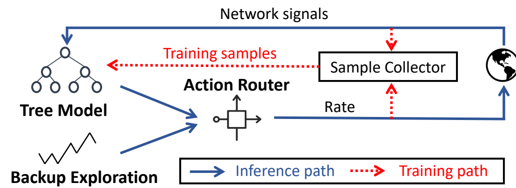


Figure 5: Overview of POLICYCACHE. In model execution mode, the tree model determines the control rate. In exploration mode, rates are generated via trial-and-error. In both modes, the sample collector gathers training samples, which are used to update the model as well as its performance score.

lowing immediate utilization of new samples without lengthy convergence. This property makes the tree-based model architecture well-suited for intra-flow learning: it learns incrementally from the flow's observations, adapts within only a few steps, and sustains high performance under the learned flow condition, as shown in Fig. 4.

3 Design

3.1 Overview

Inspired by the above motivations, we present POLICYCACHE, the first congestion control algorithm built on the *intra-flow learning* paradigm, designed for high responsiveness, strong adaptability, and low overhead. Rather than relying on heavy parametric models, POLICYCACHE employs an online tree-based structure for lightweight stream learning and control. In addition, it integrates an exploration algorithm as a backup policy to collect ground-truth samples for training the tree model[3]. This design ensures a performance lower bound during early learning while enabling highly responsive control once the policy has been learned.

Figure 5 shows the overall workflow of our congestion control algorithm. POLICYCACHE operates in fixed time intervals (e.g., 30ms). At each interval, it (i) observes network signals, (ii) decides a rate control action, (iii) records the performance of the tree-based model, and (iv) updates the model when training samples are available. To achieve this, POLICYCACHE integrates a tree-based model with an exploration algorithm, and operates in two modes:

- **Model execution mode:** The tree model processes statistical signals collected from ACKs as input and outputs the rate control action.
- **Exploration mode:** A backup policy explores rate control actions via trial-and-error and updates the explored rate.

For each interval, the *action router* switches between the two modes based on the performance of the tree model. Regardless of the mode, POLICYCACHE ensures that the *sample collector* provides training samples for updating the tree model; the model's performance score is updated by comparing its predictions with ground-truth actions obtained from exploration. This raises two challenges:

- How to obtain ground-truth samples in the model execution mode?
- How to update the model performance score in the exploration mode?

We address these challenges with two key designs. First, a *micro-exploration mechanism* introduces small oscillations in the execution mode, enabling ground-truth samples similar to those in exploration (§3.3). Second, in exploration mode, the tree model remains in a *recommendation role*: although its outputs are not enforced, they are compared against the explored ground-truth actions to update the model's performance score (§3.5).

POLICYCACHE works as follows: upon flow initiation, a tree model is initialized for each flow. Since the model is untrained, POLICYCACHE begins in exploration mode. During transmission, the sample collector gathers network signals and ground-truth rates to update the model. Once the model has learned sufficiently, i.e., its predictions largely agree with the collected ground-truth, POLICYCACHE switches to model execution mode, allowing fast control decisions. If the network changes (e.g., a new bottleneck), the model's outputs may again diverge from the ground-truth, lowering its performance score. In this case, the action router reverts to exploration mode until the model adapts.

We name our CC scheme POLICYCACHE because it functions like a cache of control policies during flow transmission, but it is *not* a key–value cache with exact matching. Each explored control action, together with the corresponding network signals, forms a rule that is stored in the tree model. The tree generalizes across nearby states and learns branching logic (e.g., "if RTT trend rises and throughput drops, decrease rate") rather than memorizing exact keys. Once the model has learned sufficient rules, POLICYCACHE switches to model execution mode, extracting control actions from the model using the observed network signals as input. This enables fast and effective control.

The rest of this section details the design components of POLICYCACHE: §3.2 introduces the tree-based model, §3.3 describes the exploration mechanisms, §3.4 explains how the sample collector generates training samples, and §3.5 presents the action router and its switching policy.

3.2 Tree-based Model

The tree-based model collects network statistical signals from Acks as model input, then outputs the rate control action. During the transmission, it incrementally updates itself to adapt to dynamic network conditions. Furthermore, it should be able to adapt to concept drift, which means the model should support link changing events. For example, if a flow changes its path, the model should be able to adapt to the new

[3] We define the ground-truth action as the explored increase/decrease decision that yields higher utility during probing.

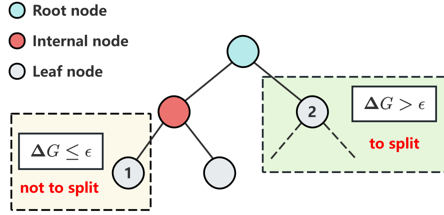


Figure 6: The Hoeffding tree splits based on information gain and Hoeffding bound.

path and re-learn the policy swiftly.

We adopt the Hoeffding Adaptive Tree (HAT) [6] as our tree-based model architecture. HAT is a lightweight incremental learning algorithm specifically designed for streaming data. Unlike traditional decision trees that must be trained offline on a fixed dataset, HAT can continuously update itself as new samples arrive, making decisions in real time. This incremental learning approach enables rapid adaptation in highly dynamic network environments.

Incremental splitting with the Hoeffding bound. In batch settings, a decision tree can compute information gain for all attributes over the entire dataset. But in a stream, data arrive one by one, so we must decide whether we have seen "enough" samples to trust the observed best attribute. As a variant of Hoeffding Tree, HAT relies on the *Hoeffding bound* to make the splitting decision: if the difference in information gain ΔG between the top two candidate attributes is larger than a small tolerance ε, then with probability $1-\delta$ the top attribute is truly the best split—even without seeing all future data.

As illustrated in Figure 6, HAT uses this probabilistic guarantee to test at each leaf whether the computed difference of information gains $\Delta\bar{G}$ between the attributes X_a and X_b with the highest information gains respectively, $\bar{G}(X_a)-\bar{G}(X_b)$, is positive and non-zero. Given a specified tolerance δ, if $\Delta G > \varepsilon$, the leaf is split immediately (node 2). If $\Delta G < \varepsilon$, the node waits until more data arrive (node 1). This process enables the tree to grow continuously while keeping statistical guarantees.

HAT differs from the original Hoeffding Tree in its ability to handle *concept drift*—situations where the underlying traffic pattern changes over time. HAT achieves this through the ADWIN algorithm that detects input pattern changes and replaces existing subtrees with new learned ones. This localized replacement strategy allows HAT to adapt quickly and precisely where link change occurs, without retraining the entire model. Appendix A provides more details about HAT.

Model Input State To improve generalizability across diverse network conditions, prior learning-based CCAs [2, 19, 21, 31] often normalize network statistics so that features appear similar under different link capacities, base RTTs, and random loss rates. This allows the learned policy to transfer to new flows. In contrast, POLICYCACHE follows the intra-flow learning paradigm and thus does not rely on cross-environment generalization. As a result, POLICYCACHE does not require explicit feature normalization or engineering; instead, it can directly leverage whatever signals are most informative for decision-making.

In practice, we use a mixed set of raw statistics and normalized features that have been widely adopted in prior work. The raw statistics include: pacing rate (Mbps), throughput (Mbps), RTT (ms), loss ratio, and the current congestion window (in MSS segments). For normalized features, similar to TCP's rate sampling, POLICYCACHE records the enforced CWND/rate in each time interval and the corresponding feedback signals from ACKs. Specifically, for the t-th interval, we use:

- Sending rate decision a_t.
- Throughput change $\frac{thr_t - thr_{t-1}}{thr_{t-1}}$, where thr_t is the average throughput in interval t.
- RTT change $RTT_t - RTT_{t-1}$, where RTT_t is the average RTT in interval t.
- Loss ratio change $\frac{1-L_t}{1-L_{t-1}}$, where L_t is the average loss rate in interval t.

To better capture temporal dynamics, we stack the above features over a sliding history window (e.g., 3 intervals) to form the final model input.

Model Output Action Similar to prior learning-based CC schemes [2,19,22,31,32], POLICYCACHE adjusts the sending rate multiplicatively. Unlike continuous action spaces, POLICYCACHE employs a discrete action space: it either increases or decreases the sending rate by a fixed step size. This design matches the exploration strategy in §3.3.

Concretely, the model outputs a probability vector over actions. For example, $[0.2, 0.8]$ indicates a 20% chance to increase and an 80% chance to decrease the rate by the fixed step size. POLICYCACHE samples a decision a_t from this distribution and updates the congestion window as:

$$cwnd_{t+1} = \begin{cases} cwnd_t \times (1+\alpha) & a_t = 0, \\ cwnd_t \times (1-\alpha) & a_t = 1. \end{cases} \tag{1}$$

We adopt probabilistic sampling so that execution matches the exploration distribution. Statistically, this yields an effect equivalent to continuous action space across states and flows. The pacing rate is then computed from the updated CWND and the observed RTT:

$$rate_{t+1} = \frac{cwnd_{t+1}}{\mathrm{RTT}_t}. \tag{2}$$

Compared with rate-based adjustment, CWND-based control naturally tolerates transient variance and traffic bursts, making it better suited to discrete-interval decision-making than per-ACK updates.

3.3 Online Exploration

Backup Exploration Mode POLICYCACHE employs a backup exploration mode to ensure stable performance when the tree-based model has not yet accumulated sufficient samples to support reliable control. This module operates in a slow loop and explores actions through a simple trial-and-error process.

In each exploration round, POLICYCACHE groups $2k$ consecutive monitoring intervals into k probing pairs. In each pair, the flow perturbs its congestion window (and the corresponding sending rate) around the current baseline $cwnd_t$ by alternately decreasing and increasing it, i.e., $cwnd_t \times (1-\alpha)$ and $cwnd_t \times (1+\alpha)$, with the starting direction chosen uniformly at random. From the feedback collected in each interval, POLICYCACHE computes the utility function, adapted from Vivace [11], as:

$$\text{utility} = x\left(1 - b \times \frac{d\text{RTT}}{dt} - c \times Loss\right), \tag{3}$$

where x is the pacing rate, $\frac{d\text{RTT}}{dt}$ is the gradient of RTT, and $Loss$ is the loss ratio. Parameters b and c weight the sensitivity to RTT variation and packet loss, respectively. This utility function has shown good performance, convergence, and TCP friendliness in previous works [11].

After k probing pairs, POLICYCACHE aggregates the results by comparing the utilities of all increases versus decreases. The final action is chosen by majority vote: if more pairs favor increase (resp. decrease), the congestion window is updated upwards (resp. downwards); if neither direction dominates, the baseline remains unchanged. This voting mechanism makes exploration more robust against noise and short-term fluctuations, allowing POLICYCACHE to approach the optimal operating point, as shown in §4.

While adopting Vivace's utility definition, POLICYCACHE uses a far simpler exploration strategy: rather than performing gradient-based search with complex heuristics, it relies on repeated perturbations and direct utility comparison. This lightweight design suffices because backup exploration mode is only needed during the early phase of sample accumulation, after which control quickly transitions to the learned policy for proactive decision-making. Also, the action space remains consistent across modes: both exploration and model execution adjust the sending rate discretely by the same fixed step size α (as in Equation 1).

Micro-exploration Mechanism POLICYCACHE requires exploration to collect groundtruth rates as labels for training samples, which are unavailable during plain model inference because the procedure directly exploits the model's output rate control action. To address this, we design a micro-exploration mechanism in the model execution mode. Specifically, given a congestion window $cwnd_t$ generated by HAT, POLICYCACHE perturbs the rate in a manner similar to the backup exploration mode: it alternately decreases and increases the congestion window in the next two intervals, i.e., $cwnd_t \times (1-\alpha)$ and $cwnd_t \times (1+\alpha)$, with the starting direction chosen uniformly at random. From the feedback collected in each interval, POLICYCACHE computes the utility function according to Equation 3 and derives the groundtruth rate by comparing the utilities of the two perturbations.

Figure 7 highlights the difference in exploration behavior between the two modes. In backup exploration mode, POLICYCACHE perturbs the rate in two directions and must wait for feedback from both intervals to compute utilities and update the rate, requiring an additional RTT per round. In model execution mode, however, POLICYCACHE perturbs the rate similarly but immediately utilizes the model output for the next pair of intervals. When feedback arrives later, POLICYCACHE asynchronously computes the utilities of the two perturbations to determine the groundtruth rate. Importantly, this rate is not enforced on the current congestion window—it solely serves as training label for the tree model to update its policy and performance score as shown in the following two sections.

Due to POLICYCACHE's consistent exploration design, every rate adjustment in both modes consists of exploration pairs, ensuring uniform sample collection, continuous score updates, and guaranteed convergence (§4). The total number of transmitted packets in an exploration pair is equal to that of two non-exploratory intervals at the baseline rate. Therefore, no abrupt latency inflation occurs. Also, repeatedly executing paired actions leads the model to learn statistically and prevent starvation of delay-based CC schemes mentioned in [4].

3.4 Sample Collection

POLICYCACHE collects training samples in both modes. As discussed in §3.3, both modes rely on exploratory control to obtain ground-truth actions. The next step is to decide how to construct reliable training samples, in particular, how to align the observed states with the corresponding actions.

State–action Alignment To build the training data for the Hoeffding tree, we introduce a state–action alignment mechanism to handle the inherent delay between action execution and utility observation in network control. Directly pairing the most recent state with the immediately explored action can cause label noise, as the explored action should be enforced at the time point when exploration starts.

The sample collector maintains a sliding window of length k in exploration mode to record consecutive state-action pairs $(s_0, a_0), (s_1, a_1), \ldots, (s_k, a_k)$. Each sender state s_i corresponds to the model input defined in §3.2, while each action a_i represents an exploratory adjustment to the sending rate (increase/decrease). Since the effect of an action is only observable after at least one RTT, utilities $(u_0, u_1, \ldots, u_k)$ are later obtained from ACK feedback. Once the ground-truth action a_0^* is determined, it is aligned with the earliest state s_0 to construct the causally aligned sample (s_0, a_0^*). This align-

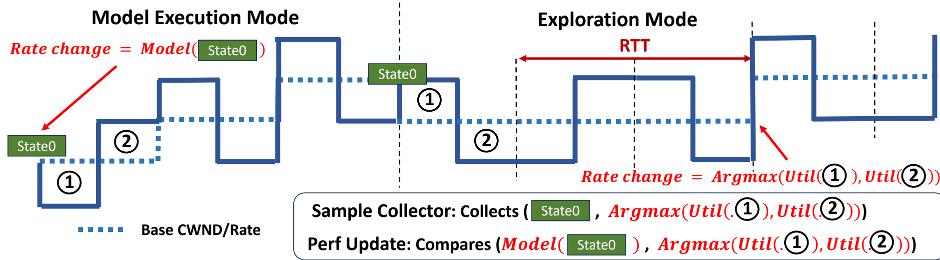


Figure 7: The workflow of POLICYCACHE in both modes, including the sample collection and performance update.

ment is implemented using a buffer that stores historical states indexed by interval id, allowing actions and subsequent feedback to be matched with the correct state to form reliable training samples. This approach ensures that each training sample reflects a high-confidence state-action mapping, reduces label errors introduced by RTT delays, and improves the reliability of the training process. Figure 7 also illustrates how the sample collector collects samples in both modes.

3.5 Dynamic Model Switching

A key challenge in POLICYCACHE's multi-model design is deciding when the model is underperforming and switching to the backup exploration and when to switch back. After each explorative pair, the action router evaluates model reliability and selects the next mode accordingly. To track model reliability, POLICYCACHE maintains **performance score** $p \in [0,1]$, where $p = 1$ indicates that the model perfectly predicts the correct action (as later verified by trial-and-error). POLICYCACHE compares p against a threshold to determine whether to stay in *model execution mode* (letting the model directly generate rate updates) or fall back to *backup exploration*.

The score is updated with each collected training sample (s, a^*) (see §3.4). The router queries the tree-based model with the state s, compares the model's action against the groundtruth label a^*, and adjusts p accordingly:

$$p \leftarrow \beta \times p + (1-\beta) \times \mathbf{1}\{a^* = HAT(s)\}. \quad (4)$$

This update rule is an *exponential moving average* with smoothing factor β, emphasizing recent performance. In model execution mode, the model output $HAT(s)$ is cached until the groundtruth action a^* becomes available, at which point the score is updated. In backup exploration mode, the router explicitly queries the model to compare its prediction with the newly derived groundtruth. Figure 7 shows which actions are compared during score updates.

Slow Start Phase To accelerate ramp-up when a flow begins transmission and the HAT is empty, POLICYCACHE employs a tailored slow start procedure:

1. Seed the model with three dummy samples labeled "increase rate," biasing initial decisions toward rate growth.
2. Initialize the performance score to $p = 1$ to start in model execution mode.
3. Temporarily triple the step size α for faster rate climbing.

The slow start phase ends when the performance score p first drops below the predefined threshold, at which point the step size is restored. This approach allows a flow to quickly ramp up its sending rate until enough exploration disagrees with the model, after which POLICYCACHE resumes standard exploration to refine policy learning.

4 Convergence Analysis

In this section, we outline the convergence properties of POLICYCACHE. Previous exploration-based scheme Vivace [11] has provided a proof for convergence under its gradient-based exploration using the framework of social concave games [13]. However, Vivace assumes that gradients are computed accurately while fixing competing flows' rates, which does not hold in realistic exploration: all flows change their sending rates concurrently, and noisy RTT measurements introduce variance in feedback signals. Here we provide an intuitive explanation for the convergence property of POLICYCACHE. The complete, formal proof is provided in Appendix B.

Theorem 1 (Convergence to Equilibrium)**.** *When n POLICYCACHE flows share a bottleneck link, the sending rates converge to a unique configuration with all flows having the same rate, $x_1^* = x_2^* = \cdots = x_n^*$.*

The intuition for this result stems from the design of the utility function. When the link is loaded, each flow makes its decision to increase or decrease its rate based on utility measurements from its exploration phase. The utility function, contains a negative term $-bx_i\frac{dRTT}{dt}$ that is proportional to the flow's own rate, x_i. This means that for a given increase in RTT, a flow with a higher sending rate will experience a greater reduction in its utility. This establishes a key *monotonicity property*: a flow with a higher sending rate has a strictly lower probability of increasing its rate compared to a flow with a lower sending rate. Over time, this difference in probabilities causes higher-rate flows to decrease and lower-rate flows to increase, leading to a convergence point where all rates are equal. This mechanism is based on long-term probabilities and therefore holds even when flows update their rates at different, asynchronous intervals.

We then show that the fair equilibrium point does not induce persistent queuing delay when the latency parameter b in

the utility function is sufficiently large. We note that [11] arrives at the same conclusion using a different proof approach.

Lemma 1 (Convergence with No Latency Inflation)**.** *If the utility parameter $b > n$ (where n is the number of flows), then the equilibrium point in Theorem 1 coincides with the optimal equilibrium point with no queuing delay.*

5 Implementation

We implement a fully functional POLICYCACHE prototype on Linux, consisting of two main components:

- **CC Kernel Module:** Responsible for collecting observable network signals on the sender side and executing the backup exploration algorithm. The kernel module sends the collected signals and explored groundtruth actions to the userspace agent and receives control actions generated by the model when in execution mode.
- **Userspace Tree-based Model Agent:** Includes the action router and sample collector. It receives network signals and groundtruth from the kernel module, handles model inference, online training, and performance scoring.

To enable efficient interaction between the kernel and userspace components, we implement a cross-space communication channel using Netlink [30]. For the tree-based model, we leverage the streaming data processing library `scikit-multiflow` [25] to implement the Hoeffding Adaptive Tree model. For network emulation during evaluation, we use Mahimahi [26] and Pantheon tunnel [1] to simulate realistic network conditions and tunnels. The implementation hyperparameters are listed in Appendix C.

6 Evaluation

6.1 Consistent Performance

We validate POLICYCACHE 's performance adaptability through extensive emulations and real-world evaluations, demonstrating that it consistently delivers stable, high performance across diverse network environments, even with unreliable signal feedback. We compare POLICYCACHE with various learning-based and heuristic-based CCAs including Astraea [21], Orca [2], Aurora [19], Vivace [11], CUBIC [16], BBR [8], and Vegas [7]. For learning-based schemes, we all use the published code and model provided by their authors.

6.1.1 Extensive Emulations

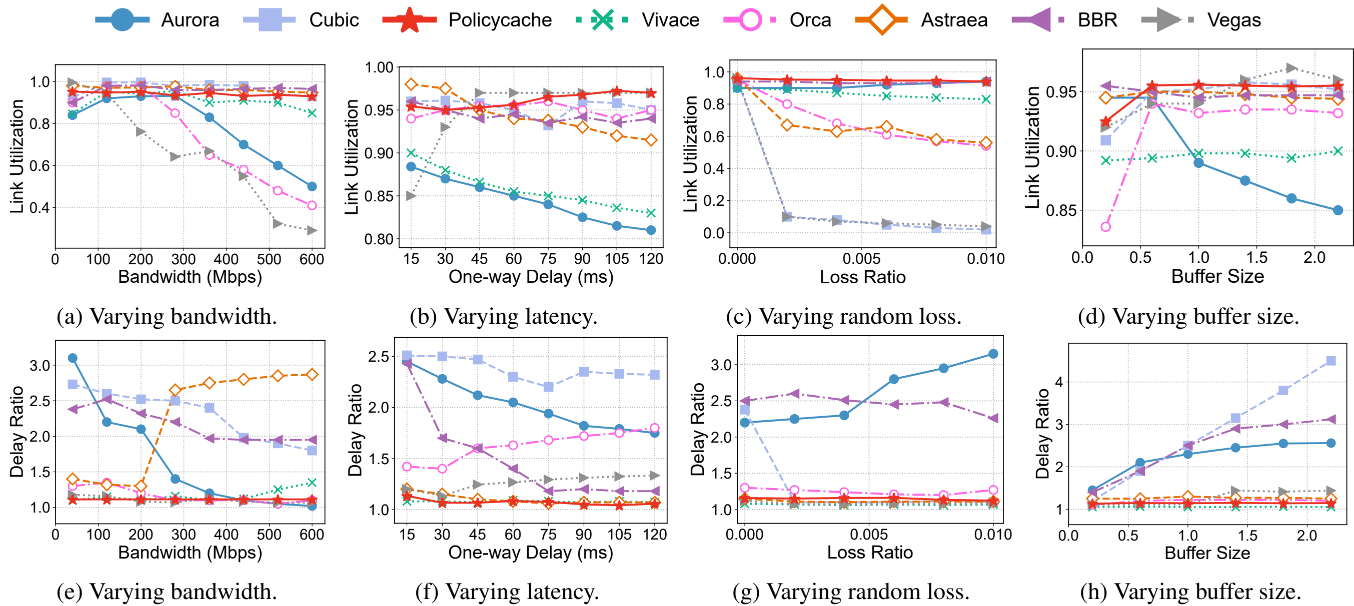
We evaluate the performance of POLICYCACHE and baseline schemes across diverse emulated network environments. Specifically, we compare their link utilization and delay ratio while varying key link characteristics—bandwidth, base delay, random packet loss rate, and buffer size. The emulation adopts a dumbbell topology with a single long-lived flow, changing one characteristic at a time while keeping the others fixed. Parameter ranges are: bandwidth 10-600 Mbps, base delay 15-120 ms, random packet loss rate 0-1%, and buffer size $0.2\times$-$2.2\times$ the bandwidth-delay product (BDP). Constant parameters are set as: bandwidth 100 Mbps, RTT 30 ms, buffer size $1\times$BDP, and no random packet loss. Fig. 8 reports the results averaged over 10 trials. Overall, POLICYCACHE consistently achieves high utilization and low delay under varying conditions, benefiting from its intra-flow learning mechanism. Without any pre-training on these settings, POLICYCACHE combines exploration fallback with the tree-based model to rapidly stabilize at high performance.

In contrast, DRL-based algorithms (Aurora, Astraea, Orca) degrade sharply once bandwidth or base delay exceeds their training regime. For example, on links above 300 Mbps, they either underutilize capacity or incur excessive queueing delays (Fig. 8a, 8e), reflecting insufficient exposure to large-BDP links during training. The online learning scheme Vivace also struggles under high delays (Fig. 8b), since its trial-and-error probing converges too slowly when RTT is large. Heuristic schemes such as BBR and CUBIC show persistent queue buildup across settings, and CUBIC further suffers from low utilization under high loss rates due to its rule-based window reduction upon packet loss (Fig. 8c).

6.1.2 POLICYCACHE on Challenging Conditions

Cellular Networks We deploy POLICYCACHE in an LTE network with rapidly fluctuating bandwidth. Following [21], we evaluate POLICYCACHE's dynamic responsiveness in the LTE setting with traces captured from real cellular networks [35]. As shown in Fig. 9, with a 15ms one-way baseline delay and unconstrained buffering, POLICYCACHE closely tracks bandwidth variations and overall outperforms existing learning-based schemes. We adopt deep buffers to absorb traffic bursts and avoid excessive losses and retransmissions. Vivace, due to slower convergence, struggles to cope with rapid bandwidth drift. POLICYCACHE's advantage stems from its adaptive exploration-based policy and the online-friendly tree-based model, which together strike an effective exploration-exploitation balance and enable rapid response and high utilization under highly dynamic conditions. We also evaluate POLICYCACHE's performance under satellite networks in Appendix D.1.

Delay Jitter Non-congestive delay jitter can arise from factors such as ACK aggregation, delayed ACKs, and transient queuing effects. To evaluate how POLICYCACHE performs under such conditions, we conducted experiments introducing controlled delay variations. The experiments were performed on two AWS c5a.4xlarge instances, with the sender in Seoul and the receiver in Tokyo. This geo-distributed setup emulates realistic wide-area network conditions while maintaining reproducibility through precise delay and jitter control. Using Linux traffic control (tc), we limited the link capacity to 100



(a) Varying bandwidth. (b) Varying latency. (c) Varying random loss. (d) Varying buffer size.

(e) Varying bandwidth. (f) Varying latency. (g) Varying random loss. (h) Varying buffer size.

Figure 8: Performance under different network conditions.

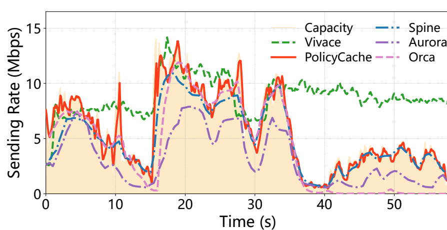


Figure 9: The responsiveness in LTE networks.

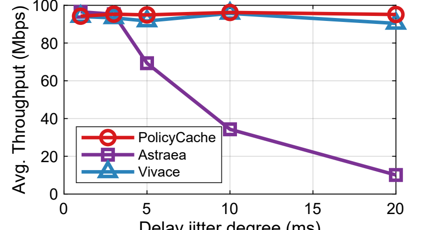


Figure 10: Performance across varying delay jitters.

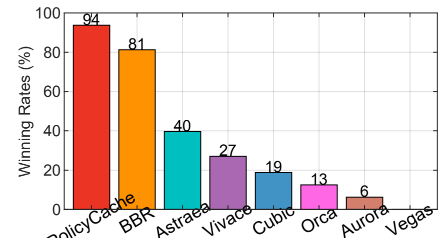


Figure 11: Fairness ranking.

Mbps and applied a base delay of 1 ms, adding jitter values of 1 ms, 3 ms, 5 ms, 10 ms, and 20 ms following a normal distribution. For each configuration, we ran 10 independent trials and measured the average throughput. The results in Fig. 10 show that POLICYCACHE, benefiting from its exploration-based design, maintains robust performance against network jitters, demonstrating strong adaptability in noisy link conditions. In contrast, Astraea, which relies on offline-learned RTT responses, is misled by jittered RTT signals and fails to utilize the available bandwidth under high delay jitter.

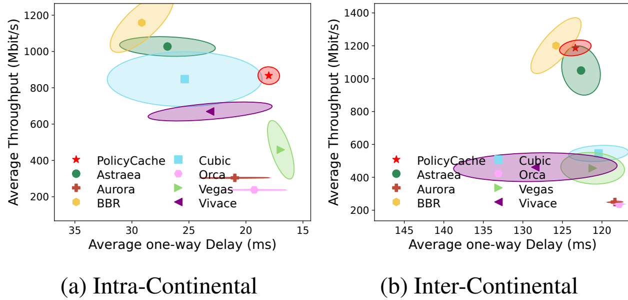
6.1.3 Real-world Networks

We conduct real-world experiments to evaluate POLICYCACHE 's performance on the wide-area Internet using AWS c5a.4xlarge instances deployed in Seoul, Tokyo, and London. The sender is fixed in Seoul, while the receiver alternates between Tokyo and London, enabling us to assess POLICYCACHE in both intra-continental and inter-continental settings. Each CC scheme is tested for 60 seconds and repeated 10 times. The results are shown in Fig. 12. We observe that POLICYCACHE consistently achieves higher link utilization and lower latency inflation compared to most CC algorithms, demonstrating both high responsiveness and strong adaptability. For example, in the inter-continental scenario, POLICYCACHE attains an average throughput of 1186.9 Mbps, outperforming Vivace (461.6 Mbps) by 2.6×, Orca (236.6 Mbps) by 5×, and CUBIC (545.3 Mbps) by 2.2×. Among other baselines, BBR delivers the highest throughput but at the cost of significant latency inflation. Clean-slate machine learning–based approaches such as Aurora and Orca fail to reach high utilization, likely due to mismatches between their training environments and the unpredictable Internet. The superior real-world performance of POLICYCACHE stems from its online tree-based structure, which enables lightweight stream learning and integrates online exploration. This design allows POLICYCACHE to adapt rapidly and robustly to Internet bandwidth variations while avoiding bufferbloat.

6.2 Fairness and Friendliness

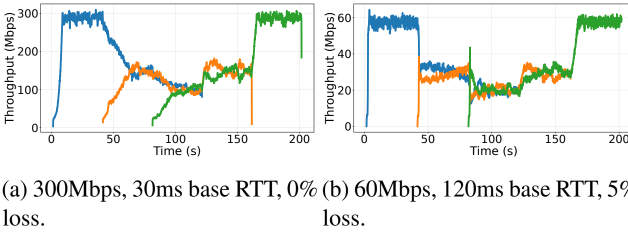
6.2.1 Fairness Among POLICYCACHE Flows

Fairness ranking. In §2.2, we presented initial evidence that POLICYCACHE converges to a fair operating point. To fur-



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

Figure 12: Real-world experiments.



(a) 300Mbps, 30ms base RTT, 0% loss. (b) 60Mbps, 120ms base RTT, 5% loss.

Figure 13: POLICYCACHE's fairness illustration.

ther evaluate fairness under multi-flow conditions, we adopt a ranking methodology inspired by the winning rate metric in [39]. Specifically, we compute Jain's Fairness Index (JFI) based on per-flow throughput: JFI $= \frac{(\sum_{i=1}^{n} x_i)^2}{n \times \sum_{i=1}^{n} x_i^2}$,, where x_i is the throughput of the i-th flow and n is the number of flows. Flow dynamics are illustrated in Fig. 13. A scheme is considered a *winner* in a scenario if it achieves at least 90% of the highest JFI while also sustaining more than 70% link utilization. The winning rate of a scheme is then defined as the proportion of scenarios in which it is a winner, and these rates are aggregated into the overall fairness ranking.

We conduct 100 experiments with varying bandwidth, base delay, and random loss rates within the training region. The results (Fig. 11) show that POLICYCACHE consistently achieves higher fairness than all baselines across diverse multi-flow conditions. This improvement stems from its probing mechanism and utility-based feedback: when congestion arises, flows adjust their sending rates based on exploration-driven utility signals, yielding more equitable bandwidth allocation, as analyzed in §4. Moreover, POLICYCACHE maintains strong fairness even under challenging environments such as high RTT(Fig. 13), owing to its efficient model updating and execution process. As a result, it achieves better fairness than exploration-based schemes such as Vivace, while simultaneously preserving high link utilization. We further evaluate POLICYCACHE under mixed traffic and multi-bottleneck topologies in Appendix D.2 and D.3.

RTT fairness We further tested the performance of POLICYCACHE under multi-flow concurrent scenarios with different RTTs. Ideally, flows sharing the same bottleneck link should achieve equal throughput. To evaluate RTT fairness, we initiated five long-duration transmission flows on a 100 Mbps emulated link, progressively adding five POLICYCACHE flows with increasing RTTs (70ms, 110ms, 150ms, 190ms, and 210ms). Each flow was started 60 seconds apart and ran for 300 seconds. Fig. 14 shows the dynamic changes in throughput and RTT for these flows. We observe that POLICYCACHE flows with different RTTs almost equally shared the bandwidth, with minimal increase in delay, as proven in the convergence analysis in Appendix B.

6.2.2 Friendliness

This section examines the TCP friendliness of POLICYCACHE. We configure a 100 Mbps bottleneck link (RTT 30 ms, buffer size one BDP), run one POLICYCACHE flow alongside one CUBIC flow for 120 s, and record their throughput ratio. The experiment is repeated under different RTTs, with results shown in Fig. 15. The ideal friendliness ratio is 1. We observe that POLICYCACHE achieves a more balanced throughput share than other learning-based schemes such as Aurora, Astraea, Orca, and Vivace. This advantage stems from POLICYCACHE 's exploration-driven adaptation: by comparing the utilities of rate increase and decrease actions, POLICYCACHE reduces its rate only when congestion signals meaningfully reflect its own sending behavior, while disregarding latency inflation and loss events caused by CUBIC's buffer-filling. This design prevents POLICYCACHE from being "killed" the way delay-based schemes (e.g., Vegas) are dominated by CUBIC, yet still enables it to optimize latency when it operates as the sole sender. We further examine friendliness against BBR (Fig. 16). While CUBIC suffers severe starvation due to BBR's aggressive buffer filling, POLICYCACHE achieves a stable balance and maintains a consistent throughput share across varying RTTs.

6.3 Low Overhead

We evaluate the computational overhead of POLICYCACHE and baseline CC schemes by measuring their CPU utilization during flow transmission. For each scheme, we run a 60-second transmission and record the CPU utilization of the process (aggregated across cores), averaging the results across runs (Fig. 17). The results show that POLICYCACHE incurs lower computational overhead than most baselines, second only to lightweight rule-based schemes such as CUBIC, BBR, and Vegas. This efficiency stems from its online tree-based structure, which supports lightweight stream learning and control, reducing the runtime cost of decision-making while retaining adaptability. Moreover, POLICYCACHE invokes its model less frequently than pure learning-based schemes, as it falls back on exploration when model confidence is low. In contrast, Aurora and Astraea rely on deep neural networks with heavy parametric models, making updates computationally expensive and slow to adapt, which results in substantially higher overhead. We also provide memory footprint and multi-flow scalability details in Appendix D.4.

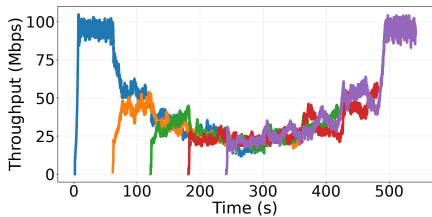


Figure 14: Multiple POLICYCACHE flows with different RTTs.

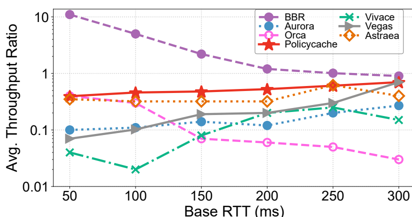


Figure 15: The throughput ratios of CC schemes to CUBIC under varying RTTs.

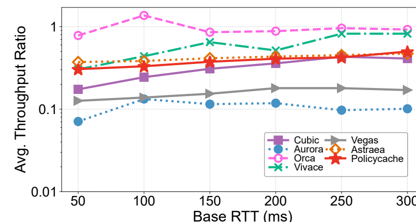


Figure 16: The throughput ratios of CC schemes to BBR under varying RTTs.

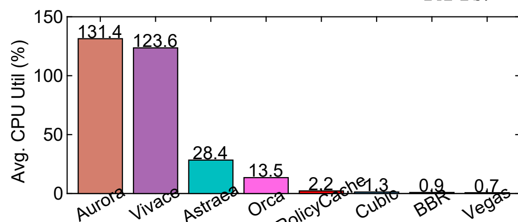


Figure 17: Average CPU utilization.

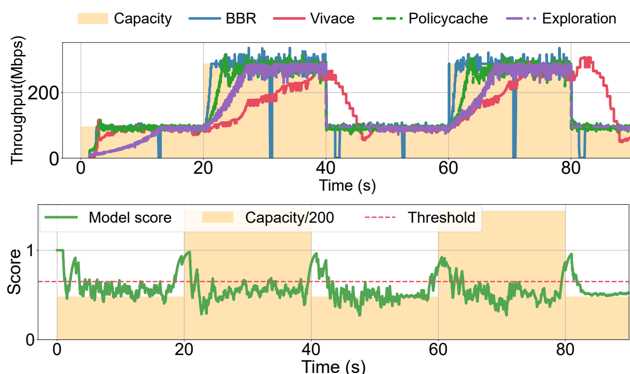


Figure 18: Dynamics of model score and throughput.

6.4 Deep Dive

To illustrate how POLICYCACHE updates the model performance score and switches between model execution and backup exploration, we emulate a network with a minimum RTT of 60 ms and a buffer size of 2.25 MB. The bottleneck bandwidth changes every 20 seconds to mimic link change events. Fig. 18 shows the dynamics of throughput, RTT, and the model performance score. We also add POLICYCACHE's backup exploration mode for comparison. We observe that: (i) When the link bandwidth remains stable and the flow converges to the optimal sending rate, the model performance score stabilizes around 0.5. This is because the optimal control direction becomes stochastic at equilibrium due to noise-dominated signals, and thus any deterministic predictor cannot exceed 50% accuracy. In this regime, the model outputs approximately equal probabilities (50% increase vs. 50% decrease), and thus remains inactive while backup exploration suffices. (ii) When the link bandwidth changes, the model rapidly adapts: within 1–2 seconds (after collecting 10–30 samples), the performance score rises above the threshold, triggering a switch to model execution mode. As a result, POLICYCACHE quickly adapts to the new bandwidth, significantly faster than exploration-based schemes such as Vivace and POLICYCACHE's backup exploration mode.

7 Related Work

Congestion Control The congestion control (CC) task has been a central focus in networking research for over three decades. Traditional schemes [5, 7, 14, 16–18] are often referred to as heuristic-based, as they are handcrafted according to specific assumptions about network conditions. For example, loss-based protocols [16, 18] use packet loss as a congestion signal and react by reducing the congestion window. More recently, numerous learning-based schemes have been proposed, which derive control policies from data rather than predetermined rules [2, 11, 19, 21, 22, 29, 31, 32, 34, 38]. Additionally, exploration-based approaches have been introduced to learn control policies through trial and error [10–12, 23]. A detailed discussion and comparison between POLICYCACHE and these prior schemes is presented in the main text.

8 Conclusion

In this paper, we introduced POLICYCACHE, a congestion control algorithm built on the new paradigm of intra-flow learning. Our lightweight, non-parametric tree model integrates seamlessly with online exploration, enabling rapid adaptation within only a few RTTs. Results show that POLICYCACHE consistently delivers stable high performance, fairness among flows, robustness to dynamic conditions, and low computational overhead. Looking ahead, we believe that intra-flow learning opens a promising avenue for designing practical, adaptive transport protocols.

Acknowledgments

We thank the anonymous reviewers and our shepherd Dushyanth Narayanan for their constructive comments. This work is supported in part by the Hong Kong RGC TRS T41-603/20R, GRF 16213621, NSFC 62402407 and TACC [37]. Kai Chen is the corresponding author.

References

[1] Pantheon tunnel. https://github.com/StanfordSNR/pantheon-tunnel. Accessed: 2021-05-30.

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

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in end-to-end congestion control. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 177–192, 2022.

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

[6] Albert Bifet and Ricard Gavalda. Adaptive learning from evolving data streams. In *International symposium on intelligent data analysis*, pages 249–260. Springer, 2009.

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

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

[9] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80, 2000.

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

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

[12] Zhuoxuan Du, Jiaqi Zheng, Hebin Yu, Lingtao Kong, and Guihai Chen. A unified congestion control framework for diverse application preferences and network conditions. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 282–296, 2021.

[13] Eyal Even-Dar, Yishay Mansour, and Uri Nadav. On the convergence of regret minimization dynamics in concave games. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 523–532, 2009.

[14] Sally Floyd, Tom Henderson, Andrei Gurtov, et al. The newreno modification to tcp's fast recovery algorithm. 1999.

[15] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: low-latency network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

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

[17] Jinbin Hu, Shuying Rao, Min Zhu, Jiawei Huang, Jianxin Wang, and Jin Wang. Srcc: Sub-rtt congestion control for lossless datacenter networks. *IEEE Transactions on Industrial Informatics*, 21(4):2799–2808, 2025.

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

[19] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning ICML*, pages 3050–3059, 2019.

[20] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Xudong Liao, Han Tian, Chaoliang Zeng, Xinchen Wan, and Kai Chen. Astraea: Towards fair and efficient learning-based congestion control. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 99–114, 2024.

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

[23] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. Pcc proteus: Scavenger transport and beyond. 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*, pages 615–631, 2020.

[24] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. Scikit-multiflow: a multi-output streaming framework. *J. Mach. Learn. Res.*, 19(1):2915–2914, January 2018.

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

[27] J. Padhye, V. Firoiu, D.F. Towsley, and J.F. Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, 2000.

[28] Shinik Park, Jinsung Lee, Junseon Kim, Jihoon Lee, Sangtae Ha, and Kyunghan Lee. Exll: An extremely low-latency congestion control for mobile cellular networks. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 307–319, 2018.

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

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

[31] Han Tian, Xudong Liao, Decang Sun, Chaoliang Zeng, Yilun Jin, Junxue Zhang, Xinchen Wan, Zilong Wang, Yong Wang, and Kai Chen. Achieving fairness generalizability for learning-based congestion control with jury. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 413–427, New York, NY, USA, 2025. Association for Computing Machinery.

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

[33] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards domain-specific network transport for distributed dnn training. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[34] Keith Winstein and Hari Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 123–134, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, 2013.

[36] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM transactions on networking*, 21(2):345–358, 2012.

[37] Kaiqiang Xu, Xinchen Wan, Hao Wang, Zhenghang Ren, Xudong Liao, Decang Sun, Chaoliang Zeng, and Kai Chen. Tacc: A full-stack cloud computing infrastructure for machine learning tasks. *arXiv preprint arXiv:2110.01556*, 2021.

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

[39] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. Computers can learn from the heuristic designs and master internet congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 255–274, 2023.

[40] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 509–522, 2015.

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

[42] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. Liteflow: Toward high-performance adaptive neural networks for kernel datapath. *IEEE/ACM Transactions on Networking*, 32(1):627–642, 2024.

Appendix

A Hoeffding Tree

Decision tree basics. A decision tree works by asking a sequence of questions about the input data and branching accordingly, until it reaches a decision at a leaf. The training process is recursive: starting from the root, the algorithm evaluates all candidate attributes and chooses the one that best splits the data into groups that are as "pure" as possible (i.e., each group dominated by a single class). This selection is guided by *information gain*, which measures how much the uncertainty (entropy) about class labels is reduced after the split. Once the best attribute is chosen, the node branches on its possible values, and the process is repeated within each branch. The recursion continues until the data in a node are sufficiently homogeneous or no further useful splits can be made. At the end, each leaf node stores a class distribution, and prediction is made by following the path defined by an input's attributes down the tree and returning the majority class at the reached leaf.

Hoeffding Tree. Hoeffding Trees (HTs) [9] are widely used in data stream mining due to their ability to incrementally build decision trees with theoretical guarantees under the Hoeffding bound.

The earliest and most classic form of the Hoeffding Tree is the Very Fast Decision Tree (VFDT) [9]. As introduced in 3.2 VFDT focuses only on the information and splitting behavior at leaf nodes. It estimates the "split merit" of candidate attributes (e.g., information gain, Gini index) based on a finite number of samples, and uses the Hoeffding bound to determine whether the empirical advantage of the current best attribute is already "large enough" to almost certainly surpass the second-best attribute, thereby making a splitting decision.

If the information gain difference ΔG between the best and second-best candidate attributes exceeds a Hoeffding bound ε, then with probability $1-\delta$ the top attribute is guaranteed to be the true best split, even without observing all future samples:

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

where R is the range of the random variable, n is the number of observed samples, and δ is the significance level.

The Hoeffding Tree algorithm trains incremental decision trees based on this principle, aiming to provide efficient online learning for streaming data environments. The Hoeffding Tree algorithm is given in Algorithm 1. According to the algorithm, this mechanism leverages the statistics maintained at the leaf nodes to compute the information gain G, which in turn drives the splitting process. After describing the Hoeffding Tree algorithm, we apply it to the congestion control problem by instantiating the variables S and X in the algorithm with the

relevant information from the congestion control scenario.

However, a key limitation of standard HTs is their poor adaptability to *concept drift*, i.e., changes in the underlying data distribution over time. Since the tree structure and sufficient statistics accumulate over a long stream, outdated splits may dominate, making it difficult for the model to adjust to new patterns. A naïve solution is to periodically *refresh* the tree by discarding the existing model and rebuilding from scratch. While simple, this approach is inefficient as it wastes previously learned structure and fails to distinguish between genuine distributional shifts and stochastic fluctuations.

Hoeffding Adaptive Tree. To address this challenge, we employ the Hoeffding Adaptive Tree (HAT) algorithm [6], which extends the Hoeffding Tree framework to effectively handle concept drift. HAT integrates the Adaptive Windowing (ADWIN) change detector into the Hoeffding Tree structure, enabling it to adaptively modify its architecture in response to detected changes in the data distribution.

Algorithm 1 Hoeffding Tree (VFDT) Algorithm

1: **Input:** Stream of examples S, attribute set X, split evaluation function $G(\cdot)$, confidence parameter δ
2: **Output:** Hoeffding Tree HT
3: Initialize HT with a single leaf l_1 (root)
4: For each class y_k and attribute value x_{ij}, set counters $n_{ijk}(l_1) \leftarrow 0$
5: **while** examples continue in stream S **do**
6: Receive new example (x, y_k)
7: Sort (x, y_k) to a leaf l using HT
8: Update counts $n_{ijk}(l)$ for each attribute value in x
9: Label l with majority class among seen examples
10: **if** examples at l are not pure **then**
11: Compute $\overline{G}_l(X_i)$ for each attribute X_i
12: Let X_a be the best attribute, X_b the second best
13: Compute Hoeffding bound $\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$
14: **if** $\overline{G}_l(X_a) - \overline{G}_l(X_b) > \varepsilon$ **then**
15: Split l on attribute X_a
16: Create one new leaf for each branch of X_a
17: Initialize counters n_{ijk} for each new leaf to 0
18: **end if**
19: **end if**
20: **end while**
21: **return** HT

Unlike traditional Hoeffding trees, to cope with concept drift HAT replaces VFDT's static counts n_{ijk} with ADWIN-based online estimators $A_{i,j,k}$ at each node (Figure 19). ADWIN both outputs up-to-date statistics for computing $\overline{G}$ and performs drift detection via a *variable-length window*: when the means of two subwindows differ by more than the threshold

$$\varepsilon_{\text{cut}} = \sqrt{\frac{1}{2m} \ln \frac{4|W|}{\delta}}, \qquad m = \frac{2}{1/|W_0| + 1/|W_1|},$$

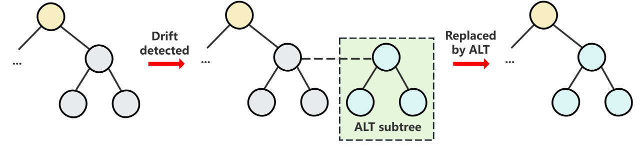


Figure 19: HAT node-level adaptation: (1) ADWIN detects drift; (2) spawn a local ALT subtree and train in parallel while comparing online performance; (3) promote ALT to replace the current subtree when significantly better. This complements the Hoeffding-bound split rule for fast, localized adaptation.

the older segment is automatically discarded, achieving *adaptive forgetting* without a preset window size(Here, W denotes ADWIN's current window and $|W|$ its size; W_0/W_1 are the old/new segments at a candidate cut; m is the harmonic mean of the segment lengths; and δ is the significance level.). In this way, HAT retains the statistical guarantee for splitting while gaining immediate sensitivity to distributional change. Meanwhile, when ADWIN at a node determines that the local statistics have drifted, HAT spawns an *alternate subtree* at that node and trains it in parallel(Figure 19). As new data arrive, if the alternate subtree's online performance becomes significantly better than the current one (e.g., as assessed by an ADWIN over the node's error stream), the alternate subtree is promoted immediately to replace the old subtree. This mechanism makes structural updates both localized (only where drift occurs) and fast.

HAT leaves can predict by the majority class derived from $A_{i,j,k}$; on some streams, a leaf-level Naïve Bayes can be used to improve accuracy. The choice of leaf predictor is orthogonal to the ADWIN-driven structural adaptation.

In our setting, we use the HAT algorithm (see Algorithm 2) to build the model: the stream S maps to online network-flow observations, and the attribute set X describes network-state features (e.g., bandwidth, RTT, loss). HAT relies on ADWIN to detect regime shifts (e.g., sudden loss onset, queue build-up) and, through the pipeline of leaf update → split → alternate-subtree promotion, responds immediately to link-state evolution; thus, under continually changing network conditions, it can consistently output reliable congestion-control decisions.

B PolicyCache Convergence Analysis

In this section, we formally prove the convergence property of PolicyCache. Specifically, we show that one or multiple PolicyCache flows converge to an *optimal equilibrium point*, i.e., fully utilizing the link, fairly sharing bandwidth, and incurring no queuing latency.

We consider a network with n senders competing over a bottleneck link with FIFO queuing. Each flow i has sending rate x_i and follows PolicyCache's exploration mechanism:

Algorithm 2 Hoeffding Adaptive Tree with ADWIN (HAT-ADWIN)

1: **Input:** Stream S of examples (x,y), attribute set X, split evaluation $G(\cdot)$, confidence δ
2: **Output:** Adaptive Hoeffding Tree HAT
3: Initialize HAT with a single leaf ℓ_1 (root);
4: For each class y_k and attribute value x_{ij}, create ADWIN estimators $A_{i,j,k}(\ell_1)$; set ℓ_1.alt $\leftarrow \emptyset$
5: **while** $(x,y) \in S$ **do**
6: Route (x,y) from root to a leaf ℓ
7: **for** each node N on the path (incl. ℓ) **do**
8: Update $A_{i,x_i,y}(N)$
9: **if** ADWIN at N signals drift and N.alt $= \emptyset$ **then**
10: N.alt $\leftarrow$ new single-leaf subtree with fresh estimators
11: **end if**
12: **if** N.alt $\neq \emptyset$ **then**
13: Train N.alt on (x,y) as a standard online tree
14: **if** ADWIN indicates N.alt is more accurate than N **then**
15: Replace N's subtree with N.alt
16: set N.alt $\leftarrow \emptyset$
17: **end if**
18: **end if**
19: **end for**
20: Label ℓ by the (estimated) majority class from $\{A_{i,j,k}(\ell)\}$
21: **if** examples at ℓ are not pure **then**
22: Compute $\overline{G}_\ell(X_i)$ from $\{A_{i,j,k}(\ell)\}$;
23: Let X_a be the best attribute, X_b the second best
24: Compute Hoeffding bound $\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$
25: **if** $\overline{G}_\ell(X_a) - \overline{G}_\ell(X_b) > \varepsilon$ **then**
26: Split ℓ on X_a
27: Create child leaves with fresh $A_{i,j,k}$ and alt $= \emptyset$
28: **end if**
29: **end if**
30: **end while**
31: **return** HAT

in two consecutive steps, it alternately increases and decreases its sending rate by a factor $(1+\alpha)$ or $(1-\alpha)$, starting in a uniformly random direction. The utility function of flow i is

$$u(x_i) = x_i\Big(1 - b\frac{d\,\text{RTT}}{dt} - c \cdot Loss\Big), \tag{5}$$

where b and c are weighting coefficients for RTT variation and loss, respectively. Since PolicyCache targets low-latency operation point (large b), we can neglect the loss term for convergence analysis:

$$u(x_i) = x_i - bx_i\frac{d\,\text{RTT}}{dt}. \tag{6}$$

Let $S = \sum_i x_i$ be the aggregate sending rate. According to the network model:

$$\frac{d\,\text{RTT}}{dt} = \begin{cases} 0 & \text{if queue is empty and } S < C, \\ \frac{S-C}{C} & \text{otherwise,} \end{cases} \tag{7}$$

where C is the link capacity.

We first prove the convergence to the equilibrium point when multiple PolicyCache flows coexist, where the sending rates are the same for all the flows.

Theorem 2 (Convergence to Equilibrium)**.** *When n PolicyCache flows share a bottleneck link, the sending rates converge to a unique configuration with all flows having the same rate, $x_1^* = x_2^* = \cdots = x_n^*$.*

Proof. When multiple PolicyCache flows compete over a bottleneck link, two scenarios may arise.

Scenario 1 (Underloaded link). When the queue is empty and the aggregate sending rate $S < C$, the utility function reduces to $u(x_i) = x_i$. In this case, if any flow i increases its sending rate, its utility strictly increases, and vice versa. Consequently, each flow increases its sending rate by a factor of $(1+\alpha)$ with probability 1. The aggregate sending rate therefore keeps growing until the system reaches scenario 2, where $S \geq C$ or the queue is non-empty, activating the $\frac{d\,\text{RTT}}{dt}$ term in the utility function.

Scenario 2 (Loaded link). In this case, each flow performs an exploration procedure: it first perturbs its sending rate upward (increase) or downward (decrease), and in the next step performs the opposite action. After recording the utilities of both actions, the flow updates its sending rate towards the direction that yields higher utility.

Formally, let x denote the global state of all sending rates. For flow i, define

$$\Delta_i(x) = u(x_i^+) - u(x_i^-),$$

where x_i^+ and x_i^- are the rates obtained by increasing or decreasing x_i by a factor of $(1 \pm \alpha)$, respectively. Flow i chooses to increase if and only if $\Delta_i(x) > 0$.

Our first goal is to prove the *monotonicity property*: if $x_i > x_j$, then

$$P(\Delta_i(x) > 0) < P(\Delta_j(x) > 0).$$

During one exploration iteration, for every realization of rate adjustments by all flows, partition the set of other flows into two groups: those that change their rates in the same direction as flow i, denoted by K, and those that move in the opposite direction, denoted by $\bar{K}$. The utilities of flow i after increasing or decreasing are

$$\begin{aligned} u(x_i^+) &= (1+\alpha)x_i - b(1+\alpha)x_i \cdot \left(\frac{(1+\alpha)\sum_{k\in K} x_k - C}{C}\right) \\ &\quad - b(1+\alpha)x_i \cdot \left(\frac{(1-\alpha)\sum_{j\in \bar{K}} x_j}{C}\right), \\ u(x_i^-) &= (1-\alpha)x_i - b(1-\alpha)x_i \cdot \left(\frac{(1-\alpha)\sum_{k\in K} x_k - C}{C}\right) \\ &\quad - b(1-\alpha)x_i \cdot \left(\frac{(1+\alpha)\sum_{j\in \bar{K}} x_j}{C}\right). \end{aligned}$$

Thus, their difference is

$$\Delta_i = \frac{x_i}{C}\Big[2C\alpha(b+1) - 4\alpha b \sum_{k\in K} x_k\Big]. \tag{8}$$

It follows that $\Delta_i > 0$ if and only if

$$\sum_{k\in K} x_k < \frac{b+1}{2b} C \triangleq T. \tag{9}$$

Here T is a constant threshold. Since the exploration phases of flows are independent, we have

$$P\big(\sum_{k\in K} x_k < T \,\big|\, i \in K\big) \le P\big(\sum_{k\in K} x_k < T \,\big|\, j \in K\big),$$

whenever $x_i > x_j$. Equivalently,

$$\mathbf{1}\{\Delta_i > 0\} \le \mathbf{1}\{\Delta_j > 0\}, \tag{10}$$

which establishes the monotonicity property.

Strict inequality. Based on Equation 9, we know that flows happen to probe in the same direction order will always increase or decrease together, leading to equality in (10). To establish the *strict* inequality, we therefore focus on the case where the two flows explore in opposite directions. Specifically, we compare $P(\sum_{k\in K} x_k < T | i \in K, j \in \bar{K})$ and $P(\sum_{k\in K} x_k < T | j \in K, i \in \bar{K})$.

Let O denote the subset of other flows that explore in the same direction as the target flow. Then, under any realization of O, the sum of sending rates in K for flow i is

$$x_i + \sum_{o\in O} x_o,$$

while for flow j it is

$$x_j + \sum_{o\in O} x_o.$$

Clearly, for every realization of O, the former is strictly larger than the latter since $x_i > x_j$. Thus, the key step is to identify realizations of O where

$$x_j + \sum_{o\in O} x_o < T \quad \text{but} \quad x_i + \sum_{o\in O} x_o > T.$$

In such realizations, flow j increases while flow i decreases, yielding strictly different probabilities.

To ensure that such realizations exist, assume that $\alpha > 0$ is chosen sufficiently small so that a single step never bridges the entire gap between any two flows, i.e.,

$$(1+\alpha)x_j < (1-\alpha)x_i,$$

for all $x_i > x_j$ (otherwise, we may approximately regard x_i and x_j as already equal). At the same time, α is large enough that finite steps allow rates to cover the interval.

If, for every realization of O, both $x_i + \sum_{o\in O} x_o$ and $x_j + \sum_{o\in O} x_o$ lie entirely above or below T, then both flows move in the same direction (or remain unchanged in asynchronous exploration). Without loss of generality, suppose both keep increasing, so there exists at least one realization with

$$x_i + \sum_{o\in O} x_o < T \quad \text{and} \quad x_j + \sum_{o\in O} x_o < T.$$

As the process continues, there will eventually be a transition realization in which both sums, previously below T, simultaneously cross above T. Within this region, we necessarily encounter realizations where

$$x_j + \sum_{o\in O} x_o < T \quad \text{and} \quad x_i + \sum_{o\in O} x_o > T,$$

implying that flow j increases while flow i decreases. Averaging over all realizations across sufficient exploration steps preserves strictness. Hence we obtain

$$x_i > x_j \implies P(\text{flow } i \text{ increases}) < P(\text{flow } j \text{ increases}),$$

and equivalently,

$$P(\text{flow } i \text{ decreases}) > P(\text{flow } j \text{ decreases}).$$

Convergence under Asynchronous Update. We now extend the convergence result to the case of asynchronous update frequencies between flows, which is the situation in POLICYCACHE. In practice, flows with different model performance scores update their sending rates at different frequencies. In each update step, a flow can choose to adopt a new rate (based on exploration outcomes) or continue exploring. We further assume that the probability of adopting a new rate is non-zero

for all flows. This setting also covers the RTT-fairness scenario, where different flows require different amounts of time to complete their exploration and update cycles.

From (9), we observe that any flow whose rate exceeds T will continue to decrease. Hence, after sufficiently many steps, the aggregate sending rate S of all n flows satisfies the upper bound

$$\lim_{t\to\infty} S(t) < nT.$$

On the other hand, when $S < C$, every flow always increases its rate (Scenario 1), which yields the lower bound

$$\lim_{t\to\infty} S(t) \geq \max(C, T) = C.$$

Therefore, in steady state the aggregate sending rate satisfies

$$C \leq \lim_{t\to\infty} S(t) < nT.$$

Within this interval, the long-run average probability that a flow increases its rate must not exceed 0.5, since otherwise S would violate the upper bound. Moreover, because the largest flow has a strictly smaller probability of increasing than any smaller flow (by the strict monotonicity result), its long-run increase probability is strictly less than 0.5. Consequently, the dynamics drive all flows towards the same average rate in the long run. We note that this balance condition is independent of the update frequencies: heterogeneous exploration periods (e.g., 200 ms vs. 20 ms) influence only the speed and variability of convergence, but not the limiting allocation itself.

□

We now show that, for sufficiently large b in the utility function, the equilibrium point corresponds to the *optimal equilibrium* with no queuing delay. Equation (8) reveals how the choice of b influences the increasing probability $P(\Delta_i > 0)$. Suppose that, after convergence, the expected contribution of flows satisfies

$$\mathbb{E}_{j\in K}\left[\sum_{k\in K} x_k\right] = \frac{b+1}{2b}C.$$

Let x^* denote the common sending rate of all flows at equilibrium. Since the target flow i is always included in K, the expected number of flows in K is

$$1 + \frac{n-1}{2} = \frac{n+1}{2}.$$

Substituting into (8), we obtain the aggregate equilibrium rate:

$$nx^* = \max\left(\frac{b+1}{b}\cdot\frac{n}{n+1}C,\ C\right).$$

Here, nx^* is bounded below by C, which corresponds to the case where the queue is fully drained (Scenario 1). To avoid persistent queueing, we require that the first term be less than C, ensuring that the system oscillates between Scenario 1 and Scenario 2 without sustaining excess queuing packets.

Observe that

$$\frac{b+1}{b}\cdot\frac{n}{n+1} = \frac{f(b)}{f(n)}, \quad \text{where} \quad f(x) = \frac{x+1}{x},$$

and $f(x)$ is strictly decreasing. Hence, the condition

$$b > n$$

guarantees that $\frac{f(b)}{f(n)} < 1$, yielding the no-queueing-delay equilibrium.

Lemma 2 (Convergence with No Latency Inflation). *If $b > n$, then the equilibrium point in Theorem 2 coincides with the optimal equilibrium point with no queuing delay.*

This result matches the requirement on b for achieving a no-queue equilibrium established in prior work [11].

Basic parameters	
Monitor Interval	30ms
Exploration parameters	
Step size (α)	0.025
Utility weight (b)	900
Utility weight (c)	11
Probing pairs (k)	2
Model parameters	
Performance score decay (β)	0.9
Performance threshold	0.65
History window size	3
Tree model (HAT) parameters	
Grace period	50
Split confidence	1×10^{-7}
Tie threshold	0.1
Leaf prediction	Naive Bayes Adaptive
NB threshold	50
Split Strategy	Binary

Table 2: Hyperparameters used in POLICYCACHE.

C Training Hyperparameters

The training hyperparameters of POLICYCACHE are given in Table 2.

D Additional Experiments

D.1 Satellite Link Evaluation

Following the configuration in [11], we also evaluate POLICYCACHE on an emulated satellite link (bandwidth 42 Mbps, RTT 800 ms, random loss rate 0.74%). The average over ten

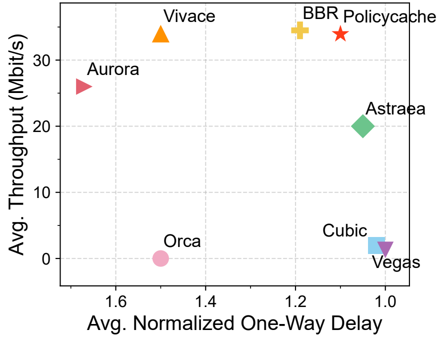


Figure 20: Satellite network

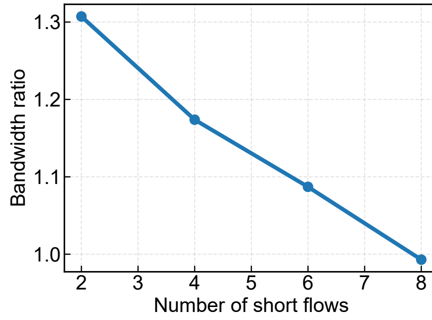


Figure 21: Impact of short-lived flow dynamics.

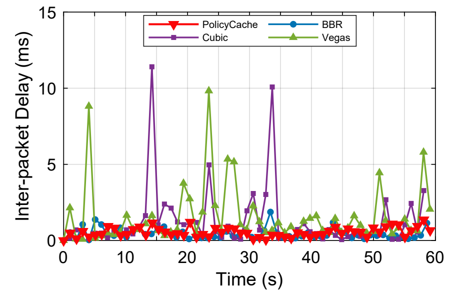


Figure 22: The average inter-packet delay of different CC schemes.

runs is shown in Figure 20: POLICYCACHE achieves link utilization above 75% while keeping latency inflation below 12% of the one-way base propagation delay.

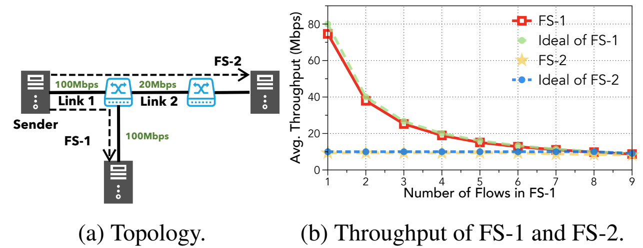
D.2 Mixed Traffic and Application Scenarios

Impact of Short-lived Flow Dynamics To evaluate the robustness of POLICYCACHE in mixed traffic scenarios, we conducted experiments with a fixed background of two long-lived flows. We treated the number of concurrent short-lived flows, k, as a variable ranging from 2 to 8. Each experiment lasted 100 seconds, during which we recorded the Link Utilization Ratio (defined as the average utilization of long-lived flows divided by that of short-lived flows), as shown in Fig. 21. The results demonstrate that POLICYCACHE maintains good inter-flow fairness with increasing short flows.

Real-Time Communication (RTC): To further evaluate the robustness of POLICYCACHE in delay-sensitive scenarios, we deploy Salsify [15], a state-of-the-art WebRTC system, modified to operate over TCP. RTC applications demand an extreme balance between throughput and latency to prevent UI freezing and lag. In our setup, a client establishes a video call with a Salsify server over a shared LAN WiFi environment. Figure 22 presents the average inter-packet delay across 10 independent trials. Notably, POLICYCACHE achieves a low inter-packet delay comparable to BBR, while significantly outperforming both Cubic and Vegas. This consistency in a fluctuating WiFi environment highlights the superior robustness of POLICYCACHE. This is because POLICYCACHE's online learning structure allows it to maintain optimal, low-latency performance by rapidly adapting to real-time channel variations.

D.3 Multi-bottleneck Networks

Multi-bottleneck Networks We next evaluate POLICYCACHE in a setting where flows encounter more than one potential bottleneck along the path. Following the multi-bottleneck setup in [21], we adopt the two-bottleneck topology



(a) Topology. (b) Throughput of FS-1 and FS-2.

Figure 23: Fairness in multi-bottleneck topology.

in Figure 23a, where Flow set 1 (FS-1) and Flow set 2 (FS-2) both traverse an upstream link (Link 1), and only FS-2 continues over a downstream link (Link 2). Link 1 and Link 2 are configured to 100 Mbps and 20 Mbps, respectively, each with a base RTT of 30 ms and sufficiently large buffers so that link capacity, rather than buffer sizing, is the dominant constraint. FS-2 always contains two long-lived flows, whereas the number of FS-1 flows is varied. When FS-1 has fewer than eight flows, FS-1 is capacity-limited by Link 1 while FS-2 is limited by Link 2; once the FS-1 flow count reaches eight or more, Link 1 becomes the common bottleneck for both flow sets. All flows are started simultaneously, and Figure 23b reports the average throughput across 10 independent runs, with standard deviation below 5%. We observe that, across all configurations, the measured throughputs of FS-1 and FS-2 closely follow the theoretically fair allocation, indicating that POLICYCACHE maintains near max-min fairness in multi-bottleneck and bottleneck transition scenarios.

D.4 Resource Efficiency and Scalability

Memory Footprint. In addition to CPU utilization, runtime memory consumption serves as a critical constraint for deployment in kernel space. Figure 24 compares the memory footprint of POLICYCACHE's Hoeffding Adaptive Tree (HAT) against the DNN-based baseline (online MLP) as the number of learned samples increases. We observe that the DNN's memory usage rises rapidly and stabilizes at approximately

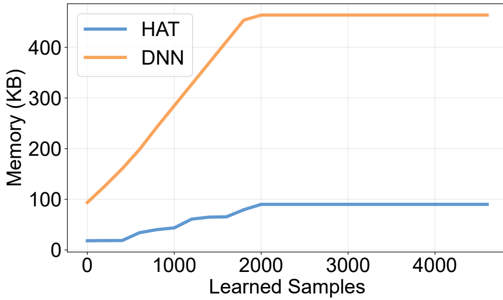


Figure 24: The memory footprint of POLICYCACHE's Hoeffding Adaptive Tree (HAT) against the DNN-based baseline (OnlineMLP) as the number of learned samples increases.

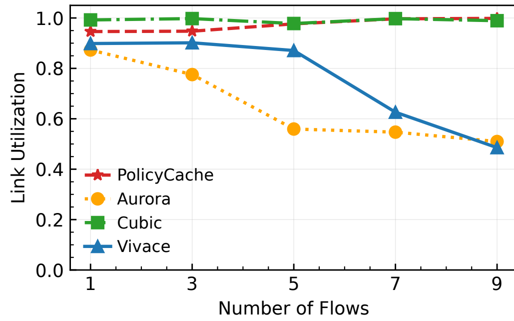


Figure 25: The link utilization of POLICYCACHE and baselines under multi-flow conditions.

460 KB. This substantial overhead is inherent to online deep learning methods. To mitigate catastrophic forgetting and stabilize gradients, the DNN model keeps an experience replay buffer with size 2048 that stores recent samples for online training. Consequently, memory consumption grows linearly as the buffer accumulates data. Once the buffer reaches capacity, the footprint plateaus, as the DNN's memory requirement is fundamentally determined by its fixed parameter count and the buffer's maximum size. In contrast, POLICYCACHE's HAT processes streaming samples sequentially, updating sufficient statistics at leaf nodes and immediately discarding raw data. The curve exhibits a gradual, step-wise increase corresponding to the tree's structural growth (node splitting), eventually stabilizing at around 100 KB, the defined maximum memory footprint of the tree model, which we find sufficient for all tested network conditions.

Multi-flow Scalability and Performance: To evaluate the robustness and computational efficiency of POLICYCACHE in high-concurrency environments, we simulate a multi-flow scenario on a 4-core virtual machine. We increase the number of concurrent flows from 1 to 9 (with a step size of 2) on a 200 Mbps bottleneck link. As illustrated in Fig. 25, POLICYCACHE consistently maintains high link utilization (near 1.0), performing on par with the lightweight rule-based Cubic. In contrast, learning-based baselines experience severe performance degradation as the flow count increases. Aurora, which relies on a heavyweight DNN, suffers from significant CPU contention; the high overhead of its inference engine interferes with the data path, leaving the CPU unable to sustain the required packet transmission rate. Similarly, Vivace relies on an extensive exploration phase with a long control loop, leading to high CPU consumption and throughput collapse ([41, 42] observes similar performance degradation). The superior robustness of POLICYCACHE stems from its lightweight HAT structure. Unlike fixed-strategy baselines or computationally expensive DNNs, POLICYCACHE integrates efficient online learning with minimal overhead. This allows it to dynamically adapt its decision-making to the shifting network state of each individual flow without saturating the system's CPU resources. These results confirm that POLICYCACHE is uniquely capable of providing high-performance, adaptive congestion control even in resource-constrained or high-density flow environments.