Spine: An Efficient DRL-based Congestion Control with Ultra-low Overhead

Han Tian[1]*, Xudong Liao[1]*, Chaoliang Zeng[1], Junxue Zhang[1,2], Kai Chen[1]†
[1]iSING Lab, Hong Kong University of Science and Technology [2]Clustar

ABSTRACT

Previous congestion control (CC) algorithms based on deep reinforcement learning (DRL) directly adjust flow sending rate to respond to dynamic bandwidth change, resulting in high inference overhead. Such overhead may consume considerable CPU resources and hurt the datapath performance. In this paper, we present Spine, a hierarchical congestion control algorithm that fully utilizes the performance gain from deep reinforcement learning but with ultra-low overhead. At its heart, Spine decouples the congestion control task into two subtasks in different timescales and handles them with different components: i) a lightweight CC executor that performs fine-grained control responding to dynamic bandwidth changes, and ii) an RL agent that works at a coarse-grained level that generates control sub-policies for the CC executor. Such two-level control architecture can provide fine-grained DRL-based control with a low model inference overhead. Real-world experiments and emulations show that Spine achieves consistent high performance across various network conditions with an ultra-low control overhead reduced by at least 80% compared to its DRL-based counterparts, similar to classic CC schemes such as Cubic.

CCS CONCEPTS

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

KEYWORDS

Congestion control, Deep reinforcement learning, Transport layer protocols

ACM Reference Format:
Han Tian[1]*, Xudong Liao[1]*, Chaoliang Zeng[1], Junxue Zhang[1,2], Kai Chen[1]†. 2022. Spine: An Efficient DRL-based Congestion Control with Ultra-low Overhead. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3555050.3569125

1 INTRODUCTION

Driven by the tremendous successes achieved by deep reinforcement learning (DRL) models in wide areas, e.g., games [32, 33, 39], computer systems, and networking [16, 29, 30, 45], the community is exerting efforts to gain the same success on the network transport by incorporating DRL into congestion control (CC) [1, 19, 27]. As one of the major advantages, DRL-based CC schemes have the capability to adapt to variant network conditions with one single control policy. Therefore, network engineers can be free from the operational challenge of manually tuning CC hyperparameters for unseen network conditions.

Despite being promising, previous DRL-based CC schemes suffer from high computation overhead incurred by complex model inference for sending rate adjustment (§2.2). Such high inference overhead consumes non-negligible CPU resources [1] and interferes with the datapath performance [47]. Existing solutions handle the overhead issue by lowering the inference frequency to extend the response interval [1, 19, 27]. During the interval, the CC is out of DRL control, either making no rate adjustment [19, 27] or relying on a classic scheme, e.g., Cubic in Orca [1], as a remedy. Therefore, they fail to fully exploit the performance superiority provided by DRL models and is vulnerable to network congestions due to the coarse-grained control [47].

Given the above dilemma, we ponder a question: can a DRL-based CC provide fine-grained control for every ACK while preserving a low computation overhead? In this paper, we present Spine to answer this question affirmatively.

At its heart, Spine adopts a hierarchical control architecture consisting of a lightweight CC executor that reacts to every ACK and loss event as well as a DRL-based policy generator that periodically generates control sub-policies for the CC executor to adapt to the change of network condition, e.g., bandwidth capacity variation or flow arrivals and departures. Specifically, a sub-policy in Spine is a lightweight parameterized control logic based on additive-increase/multiplicative-decrease (AIMD) and can be defined by a set of parameters outputted by the policy generator (§3.3). Compared to the network event reaction (e.g., ACK and packet loss), the sub-policy adaption for a new network condition is infrequent, resulting in low-frequency DRL model inference. Therefore, Spine can perform a fine-grained DRL-based CC with a low computation overhead.

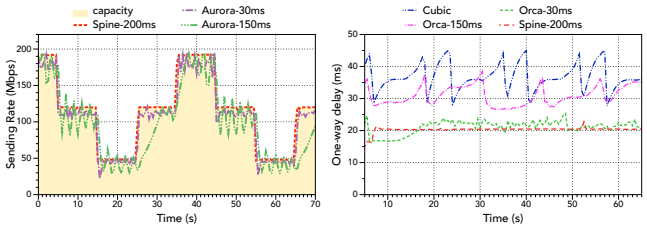
To support a timely network adaption while preserving a lower model inference overhead, Spine leverages a flexible sub-policy update strategy by further introducing a *watcher* module, a smaller model compared to the policy generator (§3.4). The watcher judges whether the current sub-policy still works well and triggers the policy generator to update it if necessary. As a result, this flexible update strategy significantly reduces the execution frequency of the policy generator and potential cross-space (kernel and userspace) communications involved by sub-policy update operations[1], especially in a stable network condition.

*Equal contribution. † Corresponding author.

CoNEXT '22, December 6–9, 2022, Roma, Italy

ACM ISBN 978-1-4503-9508-3/22/12…$15.00
https://doi.org/10.1145/3555050.3569125

[1]The CC executor can be enforced in kernel while the DRL model is usually executed in userspace



(a) Sending rates (b) Delay

Figure 1: The performance of DRL-based CC schemes with different control intervals.

We have implemented a fully functional Spine prototype in Linux. We have integrated the CC executor with Linux kernel TCP and implemented cross-space communication functionalities for the RL model to update its sub-policy. Based on this prototype, we performed efficient distributed training in various emulated network conditions and evaluated it extensively. Experimental results show that Spine achieves consistent high performance across emulated networks and real-world testbeds with ultra-low control overhead. For example, when using Spine with a large monitor interval of 300ms, Spine stills achieves higher throughput and lower latency inflation compared to previous DRL-based CC schemes using monitor interval of 30ms. Meanwhile, it takes a much lower CPU utilization (2.6% for a single flow) reduced by at least 80% compared to its DRL-based counterparts (131.5% for Aurora and 14.3% for Orca) and comparable to classic CC Cubic (1.1%).

2 MOTIVATIONS

In this section, we explain the trend and difficulty of current learning-based congestion control algorithms and how the characteristics of CC motivate our hierarchical control logic.

2.1 DRL-based Congestion Control

Unlike supervised learning algorithms that focus on prediction and classification tasks, reinforcement learning tackles sequential decision-making processes by maximizing the cumulative reward in the long run during the interactions with the environment. Armed with deep neural networks, deep reinforcement learning plays the core role in achieving super-human performance in many games and real-world decision-making tasks [33, 39]. Therefore, researchers have recently incorporated DRL into congestion control to seek to improve control policy and generalize to various unseen network conditions [1, 19, 27]. While heuristic-based CC algorithms hand-craft signal-response mapping based on some specific assumptions, DRL-based methods learn the mapping automatically from the empirical data, thus showing better generalization and outperforming specially engineered methods across various network conditions [1, 19, 27].

2.2 Overhead vs. Performance

Despite being promising, DRL-based models spend several orders of magnitude more time and consume much more computation resources than those simple ACK response functions in classic CC schemes. Empirical experiments in §7.1 have shown that previous DRL-based CC schemes consume much more CPU resources than

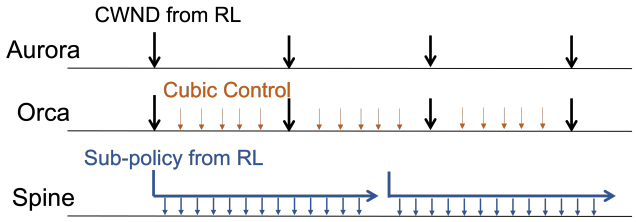


Figure 2: The control logics of Spine and previous DRL-based CC algorithms.

classic ones such as Cubic, and the overhead increases with the control frequency. As a result, there may not be sufficient CPU resources to fully support the kernel datapath processing pipelines, resulting in degraded throughput, especially with multiple concurrent flows [47]. Furthermore, such DRL-based CC schemes are hard to employ a small control interval, given the high model inference latency.

However, adopting DRL-based CC schemes with a coarse-grained control fails to fully unleash the potential of deep reinforcement learning. We perform experiments to demonstrate the performance degradation when the control frequency decreases. To demonstrate how quickly these schemes respond to network changes, we emulate a network with a minimum RTT of 30ms and buffer size of 375KB, where the bottleneck link bandwidth changes every 10 seconds. A portion of a 10-minute trace is shown in Figure 1(a). We use a clean-slate DRL-based scheme Aurora [19] with different control intervals to send traffic over the network and record the sending rate. The results show that Aurora with a large control interval (150ms) can hardly respond to bandwidth changes quickly due to a less frequent *cwnd* adjustment, compared to Aurora with a small control interval (30ms). Also, it tends to be more unstable under static bandwidth due to the response lag[2].

One way to improve the trade-off between overhead and performance is to incorporate classic schemes back for fine-grained control. Orca [1] realizes it by building a two-level control framework where the RL model and the underlying scheme (Cubic) control the current *cwnd* simultaneously at different frequencies. By restraining the sending rate according to delay increase, Orca avoids the bufferbloat issue caused by the loss-based CC Cubic. To show how Orca works with different control intervals, we emulate the same network with a static bottleneck link bandwidth (100$Mbps$) and record the delay. The result is shown in Figure 1(b). It is obvious that introducing Cubic will not eliminate the requirement of fine-grained control of the DRL model: when Orca interacts less frequently with the *cwnd* (150ms), it can hardly restrain the sending rate increased by Cubic after every ACK, leading to a higher packet delay.

Is it possible to get rid of the trade-off between control granularity and model performance of DRL-based CC? Our observation is that all the previous DRL-based schemes focus on *directly adjusting the current sending rate*. As a result, the RL agent needs to perform two subtasks: i) to consistently update sending rate in high frequency to timely respond to dynamic bandwidth changes;

[2]We also tested Aurora and Orca re-trained with a larger interval (150ms) and action range (5 * original action). However, due to the slow response, their model performance degrades severely with dramatic oscillation compared to their counterparts trained with an interval of 30ms.

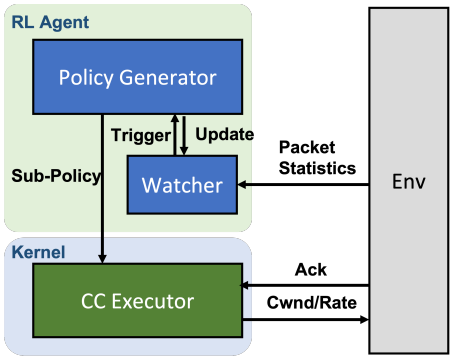


Figure 3: The high-level architecture of SPINE.

ii) to adapt its current event-action mapping to the change of network condition when the link changes or flows arrive and depart. While DRL-based schemes are good at subtask (ii) for their capabilities to generalize to various network conditions, the overhead issue keeps them from responding to dynamic bandwidth changes quickly, resulting in sub-optimal performance. On the other hand, classic schemes are good at subtask (i) due to their simple hardwired signal-action mapping but can hardly automatically adapt to the network conditions where their assumptions no longer hold. Thus, we argue that in order to achieve high model performance with low overhead, we need to decouple the CC task and adopt a hierarchical architecture to handle the two subtasks in different timescales.

2.3 Key Design Decisions

Inspired by §2.2, our key design decision is to detach the time-consuming DRL processing from quick sending rate adjustment. To achieve this, SPINE builds a hierarchical policy structure powered by DRL, which learns a DRL model that generates sub-policies instead of sending rates. A sub-policy can be regarded as a parameterized mapping from packet-level events to sending rate adjustment, which is simple and fast compared to the DRL model, thus enabling instantaneously responding to network signals at fine-grained level. As shown in Figure 2, the DRL agent periodically observes the current network condition and generates a sub-policy. As the network changes, the DRL agent keeps updating the current sub-policy at a coarse-grained level. This design brings several benefits to solve the dilemma between performance and overhead as follows:

- By generating sub-policy instead of volatile *cwnd* and sending rate, we greatly reduce the required work frequency for DRL model, resulting in a much lower overhead than previous DRL-based CC algorithms. As a result, the overhead issue is no longer a severe concern when designing DRL-based CC algorithms. (§ 7.1)
- By learning to control every single response at fine-grained level through sub-policy, SPINE achieves consistent high performance across various network conditions, even in dynamic ones where the link capacity varies drastically. (§7.2)
- The hierarchical policy architecture enables a more flexible policy update strategy. After inspecting the network condition, the DRL model can judge whether the current sub-policy still works properly. If so, the DRL model has no need to update the sub-policy, saving the cost of model inference and cross-space communication. This adaptive strategy further lowers the framework overhead significantly under stable network conditions (§ 7.5.1).

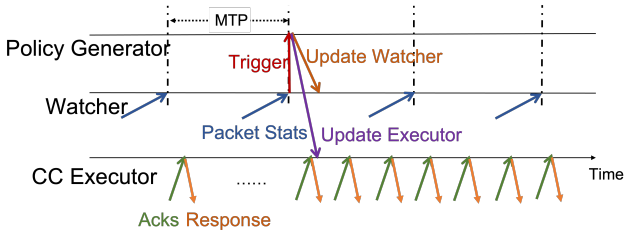


Figure 4: The time diagram of SPINE's hierarchical control logic.

3 DESIGN

3.1 Overview

Figure 3 overviews SPINE. It consists of three blocks: a *policy generator*, a *watcher* and a *CC executor*. The policy generator and the watcher together form the RL agent that traces the traffic pattern and updates the current sub-policy. The CC executor is implemented in the kernel to enforce the control sub-policy generated by the RL agent, adjusting sending rate responding to ACK and packet loss. Observing the packet statistics collected, the watcher checks whether the deployed sub-policy is still working well under the current network condition. If so, the RL agent takes no action. Otherwise, it will trigger the policy generator module and submit a *report* encoding the network condition information. Once activated by the watcher and receiving a new report, the policy generator will output a new sub-policy and update it in the CC executor. In addition, it will update the watcher so as to continue supervising the updated new sub-policy.

As a result, SPINE utilizes a hierarchical control logic. As shown in Figure 4, the policy generator, watcher and CC executor run in different timescales. The CC executor provides fine-grained control to respond to every acknowledgement. For every monitor interval (MI), the watcher observes the packet statistics as the current state input of the RL agent and triggers the policy generator once in a while. Finally, the policy generator works in a flexible signal-driven style: it only updates the watcher and the executor when triggered. As a result, the policy generator will not be triggered when the current sub-policy works well and thus have a much lower average working frequency. In addition, the watcher is generally smaller compared to the policy generator due to its simple learning target and thus has a smaller regular computation cost than previous DRL-based schemes.

We will introduce each block in detail in the following subsections. §3.2 introduces the basic components of the RL agent, including input state and reward definition. We leave the action definition to §3.3 as it relates to the sub-policy used in the CC executor. §3.4 introduces the hierarchical LSTM model architecture that combines policy generator and watcher modules in the RL agent, and its training algorithm is given in §5.

thr	The average throughput.
lat	The average packet delay.
thr_{max}	The maximum throughput of the flow so far.
lat_{min}	The minimum packet delay of the flow so far.
$cwnd$	The current congestion control window.
$loss$	The average packet loss rate.
$flight$	The number of packets in flight.
p_{rate}	The average pacing rate.

Table 1: The packet statistics used as Spine's RL agent input.

3.2 RL Agent

In Spine, the RL agent works as follows: in each monitor interval (e.g. t-th MI), it perceives the current network condition by gathering packet statistics, which are regarded as the current state s_t of the agent. That state input is then fed into the model based on deep neural networks, which will decide whether to update the underlying sub-policy by setting a flag $trigger$. If $trigger$ is $True$, the model will generate a new parameterized sub-policy by outputting its parameter setting a_t and update it to the CC executor by sending the parameter setting into the kernel. Otherwise, the model output is ignored and the sub-policy in the CC executor keeps unchanged. Once a flow is initialized, the RL agent consistently observes the dynamic network condition and updates sub-policy to maximize target performance goal.

State The input state of the RL agent contains collected packet statistics of the flow during the last MI. Here we consider the features that are closely related to the characteristic of the current network condition and flow status, which are shown in Table 1. The throughput and delay are normalized with maximum observed throughput and the minimum observed one-way delay respectively. Beyond that, we also include the currently deployed sub-policy into the state, so Spine can assess the performance of the current sub-policy for more intelligent update decision-making.

Some previous DRL CC schemes [1, 19, 27] stack a fixed-length of history features so that the agent can infer the current network condition more precisely by extracting information from history packet statistics. In Spine, however, we adopt the recurrent neural networks (RNN) as the building block of the hierarchical policy structure (§3.4). RNN is able to capture patterns and dependencies from long-term history features without the need of stacking features, which is important for the policy generator and the watcher to memorize their state histories across sparse triggering events. As a result, we can directly feed the current state feature into the model without state stacking.

Reward An RL agent needs to define a reward function to quantify the performance criterion of the task, which guides the agent to improve the generated sequence of sub-policies in the training phase. Updated with a new sub-policy from the policy generator, the CC executor interacts with the network environment by adjusting the flow sending rate and collecting rewards for each MI. Inspired by the Power-based reward in Orca [1], we define the reward function as follows:

$$R = \left(\frac{thr - \zeta \times loss}{lat'}\right) / \left(\frac{thr_{\max}}{lat_{\min}}\right) - \alpha_{psp} \times trigger, \qquad (1)$$

where

$$lat' = \begin{cases} lat_{min} & (lat_{min} \le lat \le \beta \times lat_{min}) \\ lat & otherwise. \end{cases} \qquad (2)$$

The first term in Equation 1 is based on the well-studied metric Power $\frac{thr}{lat}$ [14] and has been used in Orca. Generally, we can regard it as the ratio of normalized throughput to normalized latency, with a penalty on lost packets (weighted by ζ). [22, 23] shows that by maximizing Power, both the network and individual flows achieve the optimal point. However, as the Power cannot be fully optimized in a decentralized way [18], a small queuing delay is allowed to achieve the maximum bandwidth. As shown in Equation 2, the parameter β controls the tolerance: when the latency is smaller than $\beta * lat_{min}$, no penalty of latency is incurred.

The second term defines the penalty of triggering the policy generator to update a new sub-policy, as it will cause further inference overhead of the policy generator and the cross-space communication for sub-policy update. We call the penalty term the *pit stop penalty*, as changing sub-policies takes overhead, and we want to avoid it if not necessary, which is similar to that of changing tyres in motorsports. Without the pit stop penalty, the RL agent cannot resist the temptation to switch sub-policies for every MI, even with negligible changes. The indicator $trigger$ is equal to 1 when the watcher triggers the policy generator, otherwise 0. The coefficient α_{psp} defines the significance of the penalty. Empirical experiments in §7.5.1 show how this penalty controls the behavior of the watcher and thus the actual working frequency of the policy generator.

3.3 CC Executor

The CC executor in Spine is implemented as one of the pluggable congestion control modules, similar to Cubic and BBR, yet executing sub-policy generated by the policy generator. To meet the design goal of Spine, we expect our parameterized sub-policy structure to have the following features:

- **Simple.** The sub-policy should be simple enough so that the CC executor can execute it in the kernel with a very low computational overhead.
- **Fine-grained control.** The sub-policy should employ a fine-grained control over the sending rate or $cwnd$ to quickly respond to dynamic bandwidth changes.
- **Flexible.** The sub-policy is able to approximate various control mappings from signals to sending rates, enabling the learning of arbitrary optimal policy.

Based on these feature requirements, we design a simple yet efficient sub-policy based on the idea of AIMD. It adopts a combination of the three most commonly used indicators in congestion control: received ACK, packet delay, and loss. While executing the sub-policy, the CC executor also performs an extra slow start function at the beginning in which the sender multiplies its sending rate by 1.1 for every RTT until packet loss occurs, similar with Orca. For the sub-policy execution part, when receiving an ACK packet, the CC executor updates $cwnd$ with the following equation:

$$\Delta cwnd = \begin{cases} -\alpha_{lat} & \frac{RTT}{RTT_{min}} \ge \alpha_{tol} + 1 \\ \alpha_{thr} & otherwise, \end{cases} \qquad (3)$$

where $0 \leq \alpha_{thr}, \alpha_{lat} \leq 0.5$ and $0 \leq \alpha_{tol} \leq 2$ are hyperparameters. The behavior of Equation 3 can be described as this: given a new ACK, the CC executor will inspect the current RTT by calculating the ratio $\frac{RTT}{RTT_{min}}$. If the ratio is lower than $\alpha_{tol}+1$, it judges the link is not congested and increases the $cwnd$ by α_{thr}. Otherwise, it decreases the $cwnd$ by α_{lat}. As a result, the sub-policy defines a target packet delay point and the sending rate adjustment aggressiveness towards it in two directions: α_{thr} controls the aggressiveness in increasing $cwnd$, α_{lat} controls the sensitivity to queuing delay and α_{tol} determines the target delay point that indicates the degree of tolerance for queueing. We note that in order to reflect the transit delay changes in fine-grained control for fast response, we estimate RTT with smoothed round-trip time $srtt$, which is different from the average packet delay lat of one MI in the reward function in §3.2.

When packet loss happens, the CC executor performs a multiplicative decrease of $cwnd$ by a factor of α_{loss}, similar to Cubic:

$$cwnd_{new} = \alpha_{loss} \times cwnd \quad 0 \leq \alpha_{loss} \leq 1, \tag{4}$$

where α_{loss} indicates the sensitivity to packet loss event. After the $cwnd$ is updated, the CC executor calculates the new pacing rate as follows:

$$p_{rate} = \frac{cwnd}{RTT}. \tag{5}$$

With the above sub-policy structure, the parameter setting $(\alpha_{thr}, \alpha_{lat}, \alpha_{tol}, \alpha_{loss})$ determines the behavior of the sub-policy. We define the above parameter set as the action output of the policy generator a_t. When triggered, SPINE will update the sub-policy with a_t for the underlying CC executor that performs lower-level control. Through controlling the parameter setting, SPINE customizes the response of sub-policy to different signals to best suit the current network condition, as illustrated in §7.3.

3.4 Hierarchical Recurrent Architecture

In this section, we introduce the model architecture used in the RL agent. Inspired by the hierarchical multiscale recurrent neural network architecture (HM-RNN) proposed in [8], we design a hierarchical recurrent model, where both the watcher and the policy generator adopt recurrent neural networks as the basic building block and are connected with the report and update communications. We depict the two-layer model in Figure 5. The first layer represents the watcher, which is fed with input state s_t and adaptively triggers the upper layer. The second layer represents the policy generator that outputs the sub-policy parameter setting $a_t = (\alpha_{thr}, \alpha_{lat}, \alpha_{tol}, \alpha_{loss})$ mentioned in §3.3. Once triggered, it will receive the submitted report from the watcher layer and then i) generate a new parameter set a_t to update the sub-policy, and ii) update the watcher. By integrating the policy generator and the watcher together, we can perform gradient descent to collaboratively learn the model weights in both modules. The rest of this section briefly introduces how our hierarchical policy model works.

We use $h_t, o_t = f_{RNN}(h_{t-1}, x_t)$ to denote how an RNN block receives input and hidden state to update its internal state and output new hidden state, which, in fact, can be instantiated with any popular recurrent neural network architectures. At time step t, the watcher receives the current state s_t and the hidden states from both itself and the upper layer (h^w_{t-1}, h^p_{t-1}) generated in the last time step. It then outputs i) z'_t that decides whether to trigger the upper layer, and ii) the new hidden state h^w_t.

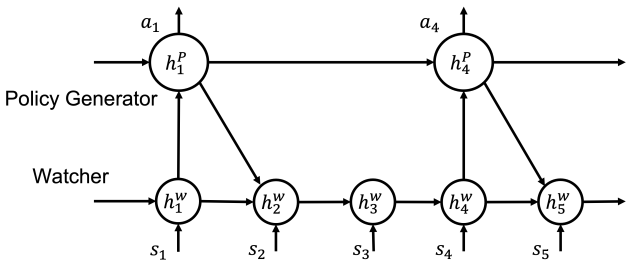


Figure 5: The hierarchical recurrent neural network architecture with different timescales.

$$h^w_t, z'_t = f^w_{RNN}\left(concat\left(h^w_{t-1}, h^p_{t-1}\right), s_t\right). \tag{6}$$

The binary trigger value z_t is then obtained by:

$$z_t = \begin{cases} 1 & z'_t \geq 0 \\ 0 & otherwise. \end{cases} \tag{7}$$

The policy generator works based on value of the trigger z_t:

$$h^p_t, a_t = \begin{cases} h^p_{t-1}, a_{t-1} & z_t = 0 \\ f^p_{RNN}(h^p_{t-1}, h^w_t) & z_t = 1, \end{cases} \tag{8}$$

where a_t is the output sub-policy parameter setting. When triggered ($z_t = 1$), the policy generator takes the hidden state of the watcher as the input report to generate new sub-policy output. Otherwise, it just reuses the previous hidden state and output results, and the watcher reuses the old h^p_{t-1} for Equation 6.

The hierarchical policy model in SPINE is different from HM-RNN in several aspects: i) HM-RNN adopts the hierarchical structure to extract high-level representations for natural language modeling, where the higher layers model the long-term dependencies (e.g. sentences or paragraphs). In SPINE, we adopt the hierarchical architecture as the policy model for reinforcement learning, where the policy generator layer focuses on outputting long-term policies; ii) HM-RNN sets complex constraints on the communications between layers to automatically detect the boundaries between sentences and paragraphs. In SPINE, we simplify it to let the policy generator obtain all the available history information for policy decision-making.

4 ANALYSIS

In this section, we perform a theoretical staleness analysis on SPINE with assumptions to show that controlling sub-policies makes the model less sensitive to control interval. We then present how the watcher module brings overhead reduction to the control system. Due to space limitations, the full proofs of the theorems in this section can be found in Appendix A.

Modeling the trade-off between performance and control interval. For our CC algorithm working with fixed monitor interval T (i.e., the working frequency of the watcher/agent is $1/T$), it detects the network condition for each interval and updates a new sub-policy that fits best. We assume that the sub-policy will become stale as the network environment changes, which we define as a *policy drift event*. After the event, the performance will degrade due to outdated sub-policy until SPINE updates it in the next MI.

T(sec)	$\lambda = 1$		$\lambda = 1000$	
	k	ETR	k	ETR
0.005	0.005	99.7%	5	19.9%
0.05	0.05	97.5%	50	2%
0.2	0.2	90.6%	200	0.5%
0.5	0.5	78.7%	500	0.2%

Table 2: The effective time ratios (ETR) with different MIs and policy drift event frequencies.

To simplify the analysis, we assume the sub-policy works in binary mode: it either functions "well" or "badly". Thus, all time periods during which the sub-policy is stale (or fresh) are treated equally in terms of performance. We refer to the ratio of the effective time when the sub-policy functions well to the total time as the *effective time ratio* (ETR) and use it to measure the effectiveness of CC algorithms in this section. The following theorem shows how the effective time ratio relates to the dynamic of the network condition and the control interval.

Theorem 1. *Suppose the time between two policy drift events complies with an exponential distribution with parameter λ. Then, the effective time ratio of Spine with fixed MI length T is $\frac{1}{k}(1 - e^{-k})$, where $k = \lambda T$ is the expected number of policy drift events happening during one MI.*

The distribution parameter λ defines the policy drift event frequency and how fast a sub-policy will become stale. It is decided both by the dynamics of the network environment and the sub-policy structure. A sub-policy that works well for a longer period should have a lower λ value. Based on the theorem, we present how the effective time ratio changes according to the MI length T and λ in Table 2. The result intuitively shows that ETR is much less sensitive to the control interval when policy drift event frequency is low, e.g., $\lambda = 1$, which means the sub-policy works well for one second on average. On the other hand, if we regard adjusting *cwnd* also as a type of sub-policy, it will incur a much larger policy drift event frequency, as the sending rate needs to be updated quickly to respond to available dynamic bandwidth. For example, if we need to update the sending rate for every millisecond to adapt to network changes, we have $\lambda = 1000$. As a result, the ETR will decrease drastically under larger control intervals (e.g. from 19.9% to 0.2%). How to deliberately design a parametrized sub-policy with a minimum λ (i.e., being effective for the longest period) is an interesting topic that we hope to study in the future.

Watcher analysis. The following theorem shows when to add a watcher to lower the computation overhead of Spine.

Theorem 2. *With the assumption in Theorem 1, the costs of Spine with or without a watcher are equal if $\frac{cost_w}{cost_p} = e^{-k}$, where $cost_w$ and $cost_p$ are the costs of the watcher and the policy generator, and $k = \lambda T$.*

Thus, when k is small with our sub-policy strategy, a watcher model slightly smaller than the policy generator ($e^{-k} cost_p$) is enough to gain overhead benefit. For example, when $T = 200ms$ and $\lambda = 1$, the watcher design will lower the computation overhead of Spine as long as its model overhead is less than $e^{-k} \approx 81.9\%$ of the policy generator.

5 TRAINING ALGORITHM

To design a DRL-based CC algorithm, we first formulate the CC problem as a reinforcement learning problem. At each t-th MI, the flow/agent sequentially interacts with the network environment in the following way: it observes packet statistics as the state $s_t \in \mathcal{S}$, and generates new sub-policy $a_t \in \mathcal{A}$ based on the agent policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$. The sub-policy responds to packet-level signals by adjusting sending rate during the next MI, and the flow will receive a reward r_t based on the reward function and newly collected statistics as the next state s_{t+1}. Though the formulation assumes that the agent outputs sub-policy for every MI, it will not conflict with our adaptively updated strategy because, as shown in §3.4, we can directly reuse the old sub-policy in the intervals when the watcher is not triggered. The goal of the agent is to maximize the expected cumulative reward during the sequence of interactions $\mathcal{J} = \mathbb{E}(\sum_{t=0}^{T} \gamma^t r_t)$, where γ is a discount factor to help the agent focus more on collected reward in the near future.

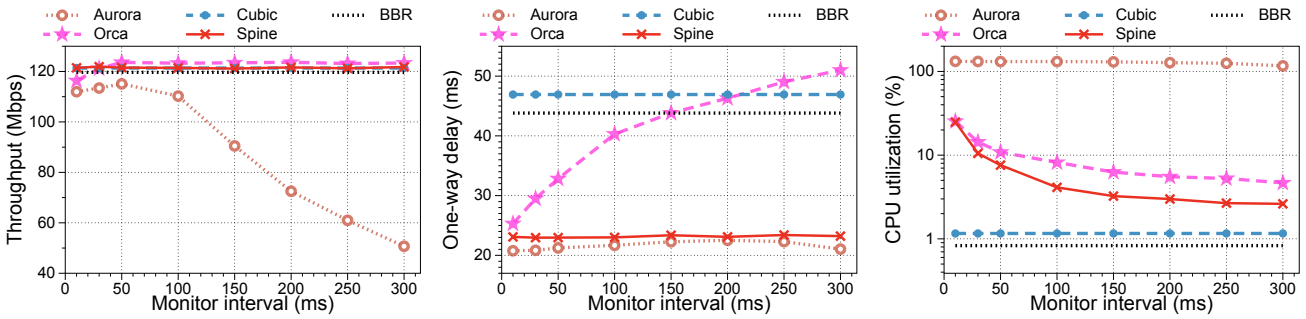
We adopt deep deterministic policy gradient (DDPG) [25], a famous model-free off-policy RL algorithm to learn sub-policies. During the training, the RL agent updates the model parameters of our hierarchical recurrent neural networks to adjust the mapping from packet statistics to sub-policy so as to maximize collected rewards. The key features of the training algorithm of Spine are as follows. See Appendix B for the complete training algorithm.

Stored hidden state and burn-in steps. Spine adopts recurrent model as the policy model, which receives and generates hidden states to encode history information. However, the traditional RL training method only stores interaction information and ignores generated hidden states in the process, which may lead to loss of history information and unstable training. We adopt the tricks proposed in [21] to solve the problem. First, we store the recurrent hidden state in the collected trajectories and use it to initialize the policy model during the training. Second, when sampling sequences of interactions for training, we also take an extra portion of the sequence at the beginning (burn-in steps), which is only used in the forwarding phase to produce a well hidden state at the beginning of the sample sequence.

Probabilistic trigger Two problems exist in training the trigger unit of the watcher: i) It is non-differentiable. The derivatives of Equation 7 are zero almost everywhere, so gradient back-propagation cannot proceed; ii) As exploration plays an important part in reinforcement learning to collect rich experience, when untriggered, the deterministic trigger unit prevents the agent from exploring more diversified sub-policy decisions (e.g., "Can I challenge the status quo with a better sub-policy?"). Therefore, we inject noise into the unit by replacing the trigger unit in Equation 7 with a probabilistic one during training:

$$z_t \sim Binomial(z'_t). \tag{9}$$

It is plain to see that $\mathbb{E}(z_t) = z'_t$ and $\frac{d\mathbb{E}(z_t)}{dz'_t} = 1$. Thus for back-propagation, we let the gradient simply go through the unit without change as if it is an identity function so that the watcher can learn to update its triggering strategy. As a result, Spine is able to explore the possibility of improving the current sub-policy generated by the policy generator, even if it performs poorly for the time being and the watcher does not recommend it through a low z'_t value.



(a) Throughput (b) Latency (c) Overhead

Figure 6: The performances and overheads of SPINE and previous DRL-based algorithms.

Bandwidth	One-way delay	Buffer size	Loss rate
40-200 Mbps	10-100ms	0.5-2 BDP	0-1%

Table 3: Training environment parameters.

6 IMPLEMENTATION

Model Architecture We use Pytorch [36] to build the hierarchical policy model in §3.4, where LSTM is used as the building block for the recurrent model. The LSTM layers in the watcher and policy generator consist of 64 and 128-dimensional hidden state vectors, respectively. In the policy generator, we feed the LSTM layer hidden state h_t^p into an MLP layer and a tanh layer to get the output action a_t. The critic model used during the training (see the training details in Appendix B) also adopts a single LSTM layer with a 128-dimensional hidden state vector. For the CC executor, we implement a congestion control module inside the Linux kernel TCP stack which receives control parameters from the RL agent and performs ACK-level congestion control. Inspired by CCP [34], the CC executor and userspace RL model are communicated via netlink [37].

Training Our training implementation is based on a generalized RL training framework DI-engine [11], which supports various DRL algorithms and customized environments and policies. We build our emulated congestion control training environments based on Pantheon [44], where Mahimahi [35] is used to emulate various network conditions. The range of settings of the training environments is shown in Table 3. We also add a random number of Cubic flows as the background traffic. We use 8 actors to collect the training experience in parallel. The entire training hyperparameter set is given in Table 6 in Appendix C. We train and evaluate SPINE on a Linux server with 80 CPU cores, 256GB RAM, and equipped with NVIDIA GeForce RTX 3090 GPU.

7 EVALUATION

In this section, we evaluate the performance of SPINE with emulated and real testbed experiments. In §7.1, we show how SPINE preserves high performance with low DRL model inference frequency due to its insensitivity to monitor interval. In §7.2, we demonstrate that SPINE achieves consistent high performance across a wide range of network conditions, including dynamically changing ones. For a better understanding of the control logic of SPINE, we inspect how SPINE updates its sub-policy in §7.3. We evaluate the convergence properties of SPINE in §7.4. Finally, we inspect the improvement brought by the watcher module and explore more possibilities of SPINE in §7.5.

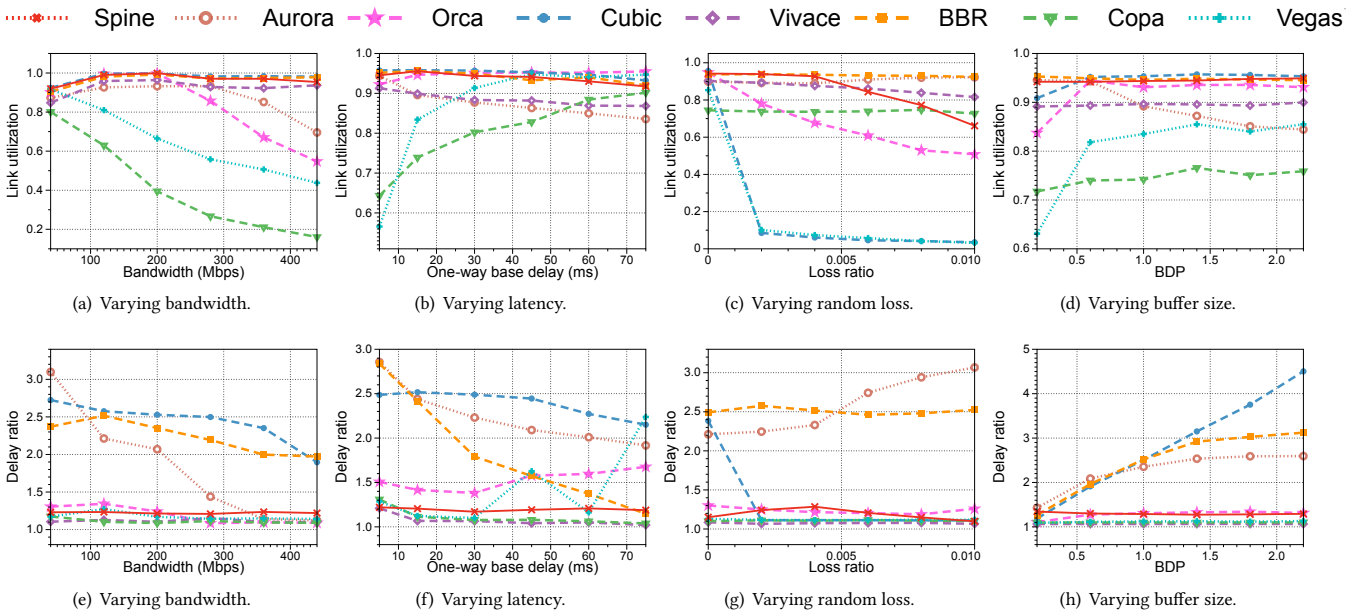
Evaluation setup. In emulated experiments, unless specified otherwise, we establish the network as the dumbbell topology with one single flow by default. The emulated bottleneck is implemented with Mahimahi [35]. In the real-world experiments, we turn to Pantheon [44] framework and deploy the sender and receiver at two AWS nodes.

7.1 Monitor Interval Insensitivity

To understand how SPINE reduces the overhead without undermining performance by lowering the control interval, we continue the motivation experiment in §2.2 and evaluate SPINE and previous DRL-based algorithms with different MIs. We use a larger buffer size (750KB) to allow enough latency inflation to indicate the congestion. We repeat each trial 10 times and report their average performance (throughput and latency) and the corresponding overheads in Figure 6[3]. The performances of Cubic and BBR are also shown as baselines. With the results of the experiment, we identify the following key observations:

- With the increase in monitor interval, the performance of previous DRL-based CC algorithms degrades: Aurora fails to achieve full bandwidth utilization (Figure 6(a)). The reason is that its sending rate is directly controlled by the RL model, which is unable to respond to bandwidth changes when the control frequency is low, and the control action may easily become stale. Orca, on the other hand, though achieving low latency with high frequency, has even higher latency than Cubic with larger MIs (Figure 6(b)). The reason is that, without a fine-grained control granularity, Orca can hardly restrain the increasing *cwnd* promptly, and thus keep experiencing high queuing delay.
- The performances of previous DRL-based CC algorithms also degrade when the MI decreases to less than 30ms. We inspect the implementations of these algorithms and find that they generally take more than 10ms for the model inference and cross-space communication between the user space and the kernel. Thus, we think the reason for the degradation is that when the MI is

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



(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 7: The performance of Spine and other CC algorithms in terms of link utilization and latency under varying bandwidth, base delay, loss rate and bottleneck buffer size. The link utilization is defined as the ratio of throughput to the link capacity, and the delay ratio as the ratio of packet delay to the base one-way delay.

small enough to be similar to the running time for enforcing new action, the actual time between when the action is enforced and when the next state is collected is very small. Therefore, there is not enough time left to collect effective feedback, and the agent cannot make timely intelligent decisions.

- Spine's high performance is insensitive to the changing of MI. As shown in Figure 6(a) and Figure 6(b), Spine yields consistent high link utilization with small queue in bottleneck across various MIs from 20ms to 300ms. The reason is that Spine imbues control response to every ACK with reinforcement learning intelligence, whose actual control frequency is independent from and much higher than the MI used. The results illustrate the important role of the hierarchical policy structure in Spine's design. Even with a much lower control frequency (e.g., MI of 300ms), Spine's sub-policy in the CC executor can still properly adjust the sending rate at a packet-level control level and thus can adapt to bandwidth changes agilely.
- With the increase of monitor interval, the overheads of Spine and DRL-based CC algorithms decrease due to lower inference frequency, as shown in Figure 6(c). The extent of overhead decrease also depends on their implementations. For example, the major overhead in Aurora may result from its inefficient userspace implementation.

The insensitivity property to control interval enables the deployment of Spine with an ultra-low working frequency without undermining model performance, which therefore achieves a much lower model overhead compared to other DRL-based algorithms. For example, when using Spine with a monitor interval of 300ms, it achieves better performance than previous DRL-based solutions (e.g., Orca) using monitor intervals of 30ms but with a much lower CPU utilization (from 10.5% to 2.6%), which is comparable to heuristic-based algorithms such as Cubic (1.1%).

7.2 Consistent High Performance

Here, we evaluate Spine with extensive emulations and real-world experiments to show its consistent high performance across various network environments. We repeat each test 10 times and report the average values. We use Orca and Aurora with MI of 30ms as it preserves relatively high performance, as shown in §7.1. For Spine, we use MI of 200ms, which has a much lower overhead due to its low inference frequency. We also compare Spine with other heuristic-based CC schemes including Cubic [15], BBR [6], Copa [3], Vegas [5] and online learning scheme Vivace [10].

7.2.1 Diverse Emulated Networks. We first compare Spine with other CC algorithms across a wide and diverse range of emulated networks by demonstrating the link utilization and latency ratio with varying bandwidth, base delay, random loss rate, and buffer size. Specifically, we alter one link characteristic of them at a time while holding the other three constant, and compare Spine with other baselines. For constant values, we use the bandwidth of 100Mbps, base RTT of 30ms, buffer size of 1 BDP, and no random loss rate. The average results of 10 trials are shown in Figure 7[4].

We observe that Spine achieves consistent good performance across different bandwidths, latencies, random losses, and buffer sizes compared to other CC baselines. For example, when changing bandwidth, Spine achieves high throughput similar to Cubic and BBR, which, however, both incur much larger queuing latency with a delay ratio from 2 to 3. The performance of previous DRL-based schemes Aurora and Orca degrades when the bandwidth or the

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

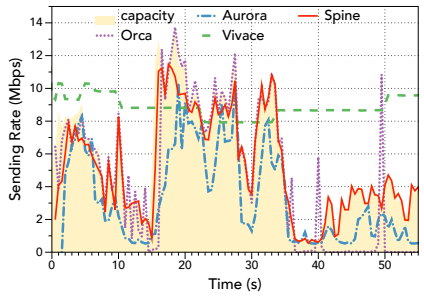
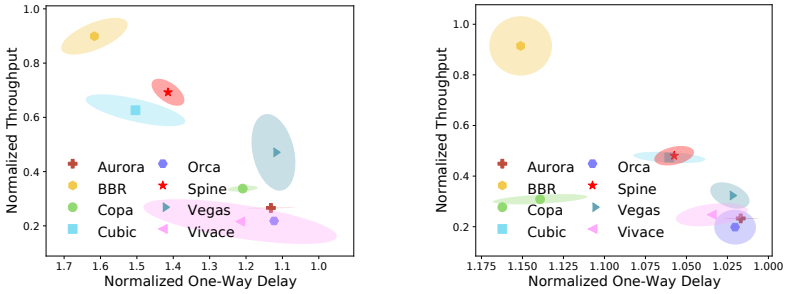


Figure 8: Spine demonstrate good reactivity.



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

Figure 9: Overall normalized throughput vs one-way delay in real-world.

base delay is large. This may be due to their limited generalization capability as they have not been trained in links with large BDPs. On the other hand, with our designed sub-policy providing domain knowledge of congestion control, Spine exhibits promising generalization ability with a limited range of training environments, which, we think, can further be improved with extensive training data from the wild Internet.

When we vary the random loss rate, schemes that use packet loss (Orca, Vivace, Vegas, Cubic, and Spine) as congestion signals reduce their bandwidths apparently. Because Spine learns to use both latency and loss as signals to detect network congestion, non-congestion loss alone has a limited effect on the decision-making of Spine. For example, when the random loss rate is 1%, the link utilization of Cubic drops dramatically (from 0.95 to 0.03), and that of Spine decreases much slower (from 0.95 to 0.64). For varying buffer size, Spine achieves small latency inflation and near full bandwidth utilization with buffer size from 0.2 to 2.2 BDP. We attribute the low requirement of the buffer size of Spine to its fine-grained control and thus a stable queue. On the other hand, the link utilization of Orca reduces to 80% in shallow buffer (0.2 BDP), as the MI of 30ms for Orca may not be enough to restrain the underlying Cubic without causing latency inflation and loss.

7.2.2 Reactivity. To evaluate how Spine reacts to dynamically changing network conditions, we create an emulated link with a trace from LTE network [43] with dramatically changing capacity. We use base RTT of 30ms, and an adequate buffer to absorb the traffic. As illustrated in Figure 8, Spine surprisingly achieves good reactivity with a large MI of 200ms: it achieves the highest bandwidth (5.08Mbps). Meanwhile, Spine can achieve the lowest latency (RTT=78.1 ms) among other learning-based CC schemes, which, however, all use a small MI of 30ms (Vivace defines RTT_{min} as its MI). We attribute the good reactivity of Spine to its hierarchical policy structure, where the sub-policy performs a fine-grained control with a low DRL model inference frequency.

7.2.3 Real-world Experiments. For real-world evaluation of Spine, we follow the experiment settings in Orca[1] and evaluate Spine in the inter-continent scenario and intra-continent scenarios. We deploy the sender at AWS Seoul and locate the receiver at AWS Singapore and London to vary the experiment environment. We evaluate each CC scheme by running its one flow for 60 seconds, repeating each trial 10 times, and summarize the overall average normalized throughput and one-way delay in Figure 9.

We observe that Spine defines one of the frontiers in terms of high throughput and low latency: it achieves better link utilization than most CCs, including Cubic. Meanwhile, it delivers smaller latency than Cubic in both intra-continental (9(a)) and inter-continental (9(b)) scenarios, respectively. The reason is that Spine adopts the DRL model to adjust the sending rate at a fine-grained level and thus can rapidly adapt to Internet bandwidth fluctuation without incurring bufferbloat. On the other hand, though performing well in the emulated experiments, other learning-based algorithms such as Orca, Aurora, and Vivace all fail to achieve high utilization, which has also been observed in emulated experiments in §7.2.1. The superior performance of Spine among learning-based schemes validates the advantage of our hierarchical policy structure to imbue ACK-level control with RL intelligence.

7.3 Under the Hood

In this section, we take a deeper look at the behavior of Spine to understand how it updates the sub-policy to adapt to various network conditions. One line of criticism of learning-based algorithms is their poor interpretability, which hinders researchers and engineers from inspecting cases with poor performance and improving the algorithm. However, Spine provides semantically task-related sub-policy parameters for lower-level congestion control, which, as we illustrate in this section, will provide insights into the design of a better CC algorithm.

To shed light on how Spine "thinks" during the control process, we run a Spine flow on an emulated link of 100Mbps with 30ms base RTT and one BDP buffer. We start a Cubic flow during this process and inspect how Spine responds by updating its sub-policy. We show its behavior in Figure 10 and mark the updated parameter for the sub-policy on the figure. We observe that Spine starts with low α_{thr} (0.06) and α_{lat} (0.08) to enforce a moderate sending rate adjustment strategy (see §3.3 for meaning of parameters). It also adopts a low α_{tol} (0.15) to maintain low latency inflation (1.2× RTT_{min}). When the Cubic flow goes in, Spine detects sudden inflation at packet delay. Thus, it updates the sub-policy with more aggressive parameters ($\alpha_{thr} = 0.18, \alpha_{lat} = 0.18$) to quickly adjust its *cwnd* responding to ACK and delay inflation signals. Meanwhile, it adopts a higher α_{tol} (0.75) and a lower α_{loss} to enable tolerance on higher latency inflation and possible packet loss. When the Cubic flow exits, Spine then restores its conservative sub-policy. One interesting observation is that Spine maps its packet statistics, primarily the link rate and the maximum throughput, to a target delay

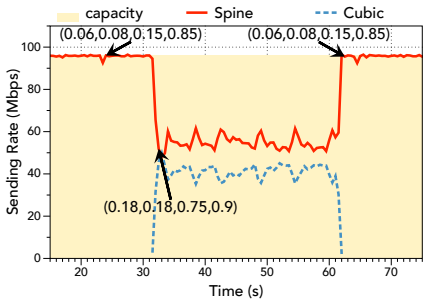


Figure 10: Details of Spine's sub-policy when competing with Cubic.

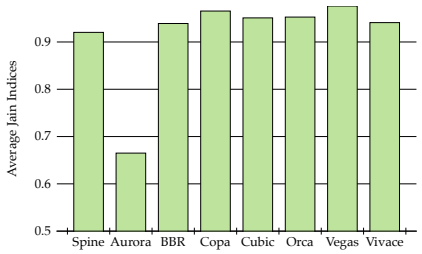


Figure 11: Jain indices of competing flows.

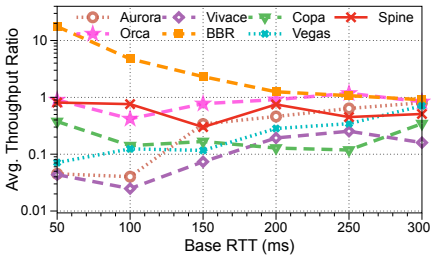
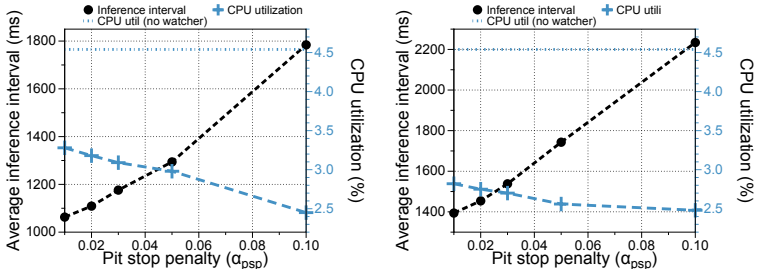


Figure 12: Throughput ratios to Cubic across RTTs.



(a) Trace 1 (square wave)

(b) Trace 2 (static)

Figure 13: The actual average inference interval of the policy generator changes with different pit stop penalty values.

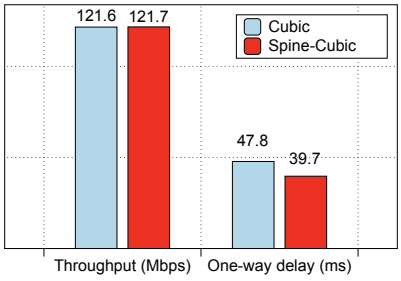


Figure 14: Spine-Cubic improves Cubic by adaptively tuning its hyperparameters.

point. When the Spine flow detects that its throughput decreases and latency increases under a conservative sub-policy, it judges that other aggressive flows come and thus resets its delay equilibrium point to a larger value until a new consensus on the queuing delay is achieved. Therefore, though working as a delay-based scheme, Spine is able to grab bandwidth from Cubic with its flexible delay target.

7.4 Fairness and Friendliness

In this section, we seek to understand how Spine performs when competing with other Spine flows and with Cubic flows. First, we setup a 100Mbps link with 30ms RTT and 1 BDP buffer, and we start three flows with a running time of 120s, for whom we set the inter-arrival time to be 40 seconds. We repeat each experiment 10 times and calculate the average Jain Index of each CC algorithm in Figure 11.

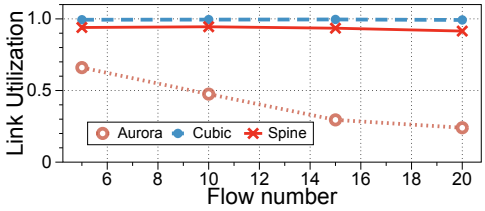
We observe that by adopting a Power-based reward that is maximized at the fair operating point, Spine achieves better fairness than the clean-slate DRL-based CC Aurora but is still not good enough compared to other classic schemes and Orca, which incorporates Cubic to provide fairness property. The reason is that the current DRL-based algorithms have not learned toward a fair scheme and have no provably guarantee of fair convergence. While we focus on reducing overhead and improving the performance of DRL-based scheme in this paper, we believe the fairness issue of DRL-based scheme will soon be solved in the near future, as several works have been devoted to learning fairness for deep reinforcement learning recently [20, 38, 48].

We also study the TCP friendliness of Spine by competing it with one Cubic flow under different base RTTs. We use the same link setting as we use in the fairness part and tune the buffer size correspondingly with respect to varying RTT. Figure 12 plots the ratio of throughput of evaluated CC to the throughput of the Cubic flow. We observe that Spine achieves good friendliness to Cubic. The reason is that we have added Cubic as the background traffic during the training, so that Spine learns to become more aggressive when competing with Cubic flows, as illustrated in §7.3.

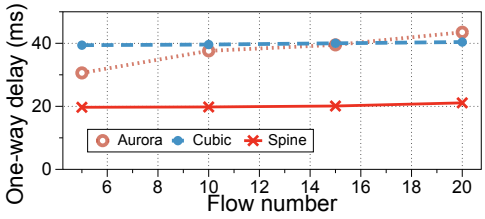
7.5 Deep Dive

7.5.1 The watcher and pit stop penalty. We next demonstrate how the watcher affects the working frequency of the policy generator as well as the overall overhead. As mentioned in §3.2, the pit stop penalty α_{psp} controls the trade-off between overhead and the triggering frequency of the policy generator. Therefore, we retrain Spine with different α_{psp}, and evaluate both their overheads and the actual average inference intervals of the policy generator. These models are evaluated on the two traces used in the motivation experiment (§2.2), where the first trace has a more dynamic link capacity (square wave) than the second one (static). To set a baseline, we also train and evaluate Spine without a watcher module, which means the policy generator directly receives the input state and outputs sub-policy for every MI. We fix the MI of all models as 200ms.

Figure 13 shows the results. We observe that adding a watcher will consistently and significantly reduce the working frequency of the policy generator. For example, when α_{psp} is set to be 0.03 (that is used in previous evaluation sections), we can decrease the



(a) Link utilization



(b) One-way delay

Figure 15: The average link utilization and latency with concurrent flows.

Flows	SPINE		Cubic	
	Thr. (Mbps)	Delay ratio	Thr. (Mbps)	Delay ratio
Overall	184.2	1.79	194.7	2.39
Per Long flow	10.5	1.79	6.2	2.39
Per Short flow	10.8	1.87	11.8	2.41

Table 4: The throughput and delay of flows in long-short experiment. The delay ratio is defined as the ratio of average one-way packet delay to the base one-way delay.

Flows	SPINE		Cubic	
	Thr. (Mbps)	Delay ratio	Thr. (Mbps)	Delay ratio
Overall	193.2	/	197.3	/
Per small-RTT flow	6.6	2.17	13.8	2.61
Per large-RTT flow	15.1	1.40	8.3	1.49

Table 5: The throughput and delay of flows in heterogeneous RTT experiment.

actual inference interval from 200ms to 1.2 seconds and 1.5 seconds in both traces. As a result, the CPU utilization is further reduced by almost 40% compared to the baseline with no watcher. Furthermore, with a larger penalty value, the trained model tends to trigger the policy generator less frequently, which would further lower the CPU overhead. It is because the watcher will learn not to trigger the policy generator unless it can obtain adequate performance gain over the pit stop penalty. Therefore, the penalty allows for small performance variation of the current sub-policy. However, we do not observe obvious performance degradations until using a very large penalty value (α_{psp} = 0.3). Finally, we find that the interval reduction is larger on the second trace. For example, when α_{psp} = 0.03, the inference interval for the second trace is 1.54 seconds, 30% larger than that in the first one (1.17 seconds). The result validates our assumption that the watcher tends to update sub-policy less frequently under stable network conditions.

7.5.2 SPINE-X. Though in the paper, we mainly evaluate SPINE using a sub-policy based on AIMD, SPINE can also adopt other customized sub-policies and even classic CC schemes by tuning the knobs of the fine-grained control process. Thus, we can generalize our framework to SPINE-X, where X can be any parameterized sub-policy. To further explore the potential of our hierarchical policy structure with existing CC schemes, we adopt a parameterized Cubic algorithm as the sub-policy in our framework, where the RL model updates the multiplicative factor β and the cubic function coefficient C of Cubic. We call it SPINE-Cubic, and evaluate it under the motivation trace in §7.1. The average throughput and one-way delay are shown in Figure 14. As expected, with the help of our hierarchical policy structure, the parameterized Cubic scheme is able to achieve lower delay while preserving full bandwidth utilization. The reason is that SPINE-Cubic adaptively controls the aggressiveness of the underlying Cubic through β and C over changing network environment. Thus, SPINE-Cubic is more flexible than the original algorithm with fixed hyperparameters. This improvement provides a great opportunity to adopt SPINE-X as an auxiliary tool to automatically tune current CC schemes regarding various network conditions.

In addition, we observe that SPINE-Cubic also inherits the limitation of Cubic. Because SPINE-Cubic only responds to packet loss to cut the congestion window, it will still be hard to mitigate bufferbloat and distinguish non-congestion loss from congestion loss. Therefore, though classical heuristic-based CC algorithms can be improved by SPINE framework, SPINE-X may still hold the drawbacks of the underlying sub-policy. It remains an open problem how to design a flexible and simple sub-policy with the minimum assumptions of the network environment.

7.5.3 Scalability. In this section, we further explore the benefits brought by SPINE's hierarchical design through the scalability experiments. We start N concurrent flows simultaneously (N = 5,10,15,20) on an emulated link of 200Mbps bandwidth, 30ms base RTT and 1 BDP buffer. 4 CPU cores are dedicated exclusively to this set of experiments to evaluate the scalability. Aurora and Cubic are also tested as baselines. We record the overall link utilization and latency, as shown in Figure 15. We observe that with the increased number of concurrent flows, the aggregated throughput of Aurora, the naïve DRL-based CC scheme, decreases. The reason is that due to the high inference overhead of multiple Aurora flows, the remaining CPU resource is insufficient to support the pipelined transmissions in the datapath. On the other hand, SPINE achieves consistently high performance with up to 20 concurrent flows, similar to Cubic.

Furthermore, we also extend the scalability experiment on heterogeneous flows, with respect to flow running times and per-flow RTTs. First, we inspect the performance of short flows of SPINE coexisting with several long-running flows, which is the common case in the Internet. Starting with the previous scalability experiment setup, we vary the running time of flows so that the concurrent flows consist of 80% short flows and 20% long flows. Specifically, we initialize 4 long flows running throughout the trial and a lot of

short flows that arrive and depart in a very frequent manner[5], so that there are almost 16 short flows co-existing with 4 long flows for a long period (100 seconds). We repeat the experiment 20 times and report the average throughput and latency for both types of flows, and the overall throughput of all flows in Table 4. We can see with the slow-start phase, short flows of Spine can grab the bandwidth quickly and achieves similar throughput to that of long flows. Second, we conduct experiments to investigate the performance of multiple Spine flows with heterogeneous RTTs. With the same experiment setup, we start 10 large RTT flows (90ms) and 10 small RTT flows (30ms) running simultaneously for 100 seconds. We repeat the experiment 20 times and record their throughputs and delay ratios in Table 5. We observe that Spine flows with small RTT share lower bandwidth than those of large RTT. In addition to the aforementioned fairness issue, another reason for this result is that Spine flows with large base RTT, according to Equation 3, tend to have a target delay point tolerating larger queuing delay under the same sub-policy. Differentiating the behaviors of flows with heterogeneous RTTs and improving RTT fairness may be a future direction of DRL-based CC schemes.

8 RELATED WORK

The congestion control task has been an enduring hotspot in the networking research field for more than three decades with a plethora of CC algorithms. The classic schemes [3, 5, 12, 15, 17, 26, 41] are generally designed based on heuristics about how should we respond to specific congestion signals in specific cases and thus are often categorized as heuristic-based schemes. For example, loss-based schemes such as Cubic [15], TCP Tahoe, and TCP Reno [17] respond to packet loss events by cutting the congestion window. On the other hand, delay-based schemes such as TCP Vegas [5] and Copa [3] respond to delay changes to keep the queuing delay low. Heuristic-based algorithms require careful hand-crafting of the signal-action mapping and can backfire under network conditions where the heuristics are violated.

Recent years have seen a plethora of learning-based CC algorithms due to the rising of machine learning [1, 2, 9, 10, 19, 31, 42, 44, 45]. For example, PCC Allegro [9] and Vivace [10] utilize an online learning paradigm. Different from machine learning solutions, they adaptively optimize pre-defined utility functions by exploring various sending rates and observing feedback from the network. However, the online exploration phase of them takes several RTTs to collect empirical performance evidence, preventing them from reacting quickly to signals timely, especially when the RTT is large and the network changes rapidly. DeepCC [2] also adopts two-level logic with DRL agent and Cubic, where the agent learns to enforce the maximum congestion window allowed by the underlying Cubic ($cwnd_{max}$), which can be regarded as an instance of Spine-X mentioned in §7.5.2 without a hierarchical policy structure.

9 DISCUSSION

In this section, we give a detailed discussion about the comparison of deep reinforcement learning and other related methods including multi-armed bandit and layering as optimization decomposition.

[5]Short flows come following the exponential distribution of $\lambda = 4$ and their running times are drawn from the Gaussian distribution $\mathcal{N}(4, 1^2)$.

Multi-armed bandit vs RL Another possible RL-based approach to solve CC is multi-armed bandit [4], where the agent learns to choose the action that maximizes the instantaneous reward in one step. Multi-armed bandit is one of the simplest reinforcement learning algorithms and has been applied in fuzzing [46], wireless network spectrum scheduling [24] and small cell activation in 5G networks [28]. However, as congestion control is a sequential decision making process where the actions (sending rate adjustments) enforced by the end-host have long-term effect on both the involved network elements and other competing flows' behaviors, we adopt RL rather than bandit to optimize the cumulative collected reward in the future of the flow's lifetime in this work.

Layering as optimization decomposition Many layered network architectures have been modeled as a generalized network utility maximization (NUM) problem in an integrated framework named "layering as optimization decomposition" [7], where the original optimization problem is decomposed into subproblems handled by both distributed computation elements and functional modules in different layers. We can formulate Spine as a hybrid decomposition case of optimization decomposition: the NUM problem is decomposed not only horizontally across distributed end-hosts, but also vertically across the policy generator, the watcher and the CC executor. As the policy generator learns to control the hyperparameters of the CC executor (sub-policy) in a data-driven manner, Spine can be regarded as an algorithm that adaptively optimizes various parameterized NUM subproblems implicated by the behavior of the CC executor to approximately optimize the RL objective function. An interesting future direction is to demonstrate how the parameterized subproblem of the underlying sub-policy affects/limits the approximation of the objective function theoretically.

10 CONCLUSION

In conclusion, we present Spine, a DRL-based CC algorithm. With the help of the hierarchical policy structure, Spine achieves ultra-high control frequency with ultra-low model inference frequency. Therefore, it achieves consistent high performance across various network conditions with low overhead comparable to classic CC scheme Cubic.

Spine is far from being the end of the story, and there are still many open questions about learning-based congestion control. However, we believe Spine has made a significant step forward towards a practical fully learning-based congestion control algorithm by providing a new DRL architecture and training paradigm. Also, the hierarchical policy architecture proposed in Spine will shed light on the adoption of reinforcement learning in various networking and system applications requiring fine-grained control in the future.

ACKNOWLEDGEMENT

We thank the anonymous CoNEXT reviewers and our shepherd Anna Brunström for their constructive feedback and suggestions. This work is supported in part by the Key-Area Research and Development Program of Guangdong Province (2021B0101400001), the Hong Kong RGC TRS T41-603/20-R, GRF-16215119, GRF-16213621, and the NSFC Grant 62062005. Kai Chen is the corresponding author.

REFERENCES

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

[2] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. Wanna make your TCP scheme great for cellular networks? Let machines do it for you! *IEEE Journal on Selected Areas in Communications* 39, 1 (2020), 265–279.

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

[4] Djallel Bouneffouf, Irina Rish, and Charu Aggarwal. 2020. Survey on applications of multi-armed and contextual bandits. In *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1–8.

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

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

[7] Mung Chiang, Steven H Low, A Robert Calderbank, and John C Doyle. 2007. Layering as optimization decomposition: A mathematical theory of network architectures. *Proc. IEEE* 95, 1 (2007), 255–312.

[8] Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. 2016. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704* (2016).

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

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

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

[12] Sally Floyd, Tom Henderson, Andrei Gurtov, et al. 1999. The NewReno modification to TCP's fast recovery algorithm. (1999).

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

[14] Alfred Giessler, J Haenle, Andreas König, and E Pade. 1978. Free buffer allocation—An investigation by simulation. *Computer Networks (1976)* 2, 3 (1978), 191–208.

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

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

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

[18] Jeffrey Jaffe. 1981. Flow control power is nondecentralizable. *IEEE Transactions on Communications* 29, 9 (1981), 1301–1306.

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

[20] Jiechuan Jiang and Zongqing Lu. 2019. Learning fairness in multi-agent systems. *Advances in Neural Information Processing Systems* 32 (2019).

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

[22] Leonard Kleinrock. 1978. On flow control in computer networks. In *Proceedings of the International Conference on Communications*, Vol. 2. 27–2.

[23] Leonard Kleinrock. 1979. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *ICC'79; International Conference on Communications, Volume 3*, Vol. 3. 43–1.

[24] Feng Li, Dongxiao Yu, Huan Yang, Jiguo Yu, Holger Karl, and Xiuzhen Cheng. 2020. Multi-Armed-Bandit-Based Spectrum Scheduling Algorithms in Wireless Networks: A Survey. *IEEE Wireless Communications* 27, 1 (2020), 24–30. https://doi.org/10.1109/MWC.001.1900280

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

[26] Shao Liu, Tamer Başar, and Ravi Srikant. 2008. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* 65, 6-7 (2008), 417–440.

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

[28] Setareh Maghsudi and Ekram Hossain. 2016. Multi-armed bandits with application to 5G small cells. *IEEE Wireless Communications* 23, 3 (2016), 64–73.

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

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

[31] Tong Meng, Neta Rozen Schiff, P Brighten Godfrey, and Michael Schapira. 2020. 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*. 615–631.

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

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

[34] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. 2018. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 30–43.

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

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

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

[38] Umer Siddique, Paul Weng, and Matthieu Zimmer. 2020. Learning fair policies in multi-objective (deep) reinforcement learning with average and discounted rewards. In *International Conference on Machine Learning*. PMLR, 8905–8915.

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

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

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

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

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

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

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

[46] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/

usenixsecurity20/presentation/yue
[47] Junxue Zhang, Chaoliang Zeng, Hong Zhang, Shuihai Hu, and Kai Chen. 2022. LiteFlow: High-performance Adaptive Neural Networks for Kernel Datapath. In *Proceedings of the 2022 ACM SIGCOMM 2022 Conference*.
[48] Matthieu Zimmer, Claire Glanois, Umer Siddique, and Paul Weng. 2021. Learning fair policies in decentralized cooperative multi-agent reinforcement learning. In *International Conference on Machine Learning*. PMLR, 12967–12978.

APPENDIX

A ANALYSIS PROOF

Here we give the proofs of Theorem 1 and 2.

A.1 Proof of Theorem 1

Proof. Due to the memoryless property of exponential distribution, we consider the policy effective time in one MTP with length T without loss of generality. We observe that the sub-policy updated at the beginning ($t = 0$) of the MTP becomes stale just after the first policy drift event happens, and stays outdated until the end of the interval ($t = T$). Thus, if the first event happens at time $t <= T$, the policy effective time will be t. If the the event happens after the end of the interval ($t > T$), the policy stays fresh in the whole MTP and the effective time is T. Thus we can get the expected effective time in one MTP as follows:

$$\int_0^T t \cdot p(t)dt + P\,(t > T) \cdot T \tag{10}$$

where p(t) is the exponential distribution $p(t) = \lambda e^{-\lambda t}$. Then we write $\int_0^T t \cdot p(t)dt$ as

$$\begin{aligned}\int_0^T t \cdot p(t)dt &= \int_0^T t \cdot \lambda e^{-\lambda t}dt \\ &= \lambda \int_0^T t \cdot -\frac{1}{\lambda}de^{-\lambda t} \\ &= -\int_0^T t de^{-\lambda t} \\ &= -\left[te^{-\lambda t}\Big|_0^T - \int_0^T e^{-\lambda t}dt\right] \\ &= \left[-\frac{1}{\lambda}e^{-\lambda t}\right]\Big|_0^T - Te^{-\lambda T} \\ &= \frac{1}{\lambda} - \frac{1}{\lambda}e^{-\lambda T} - Te^{-\lambda T}.\end{aligned} \tag{11}$$

Put it back into Equation 10, we have

$$\begin{aligned}(9) &= \frac{1}{\lambda} - \frac{1}{\lambda}e^{-\lambda T} - Te^{-\lambda T} + T \cdot e^{-\lambda T} \\ &= \frac{1}{\lambda}\left(1 - e^{-\lambda T}\right).\end{aligned} \tag{12}$$

We divide it by the MTP length T to get the effective time ratio

$$\frac{1}{\lambda T}\left(1 - e^{-\lambda T}\right). \tag{13}$$

□

A.2 Proof of Theorem 2

Proof. Here we try to verify the benefits of adding a watcher in our hierarchical architecture, and find out on what condition will it benefits and how much. With the assumptions in Theorem 1, it is straight to get the cost of the computation overhead of Spine without a watcher per second, denoted by $cost_{spine\_w}$:

$$cost_{spine\_w} = \frac{1}{T}cost_p, \tag{14}$$

where $cost_p$ is the computation cost of the policy generator model. With a watcher model, the policy generator will only be triggered when a policy drift event happens before the next MTP ($t \le T$). Thus, we get the cost of Spine with a watcher per second as

$$\begin{aligned}cost_{spine\_wo} &= \frac{1}{T}cost_w + \frac{1}{T}cost_p \cdot P(t \le T) \\ &= \frac{1}{T}cost_w + \frac{1}{T}cost_p \cdot (1 - e^{-\lambda T}),\end{aligned} \tag{15}$$

where $cost_w$ is the computation cost of the watcher. Then it is plain that $cost_{spine\_wo} \le cost_{spine\_w}$ if and only if $cost_w \le cost_p \cdot e^{-\lambda T}$

□

B THE TRAINING ALGORITHM

Spine adopts the actor-critic training paradigm to learn the agent policy, because directly training on empirical rewards often suffers from large variance due to bandwidth changes. The key idea is to introduce a *critic* model to estimate the expected cumulative reward, which guides the actor to learn its sequence of sub-policies, as shown in Figure 16. Specifically, the critic model approximates the action-value function $Q^{\pi_\theta}(s, a) = \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t | a, s]$, the expected cumulative reward that an agent will collect when it executes action a at the state s, and follows the policy π_θ afterwards. With a well-learned critic, the actor is trained to select the best sub-policy so as to maximize the action-value function.

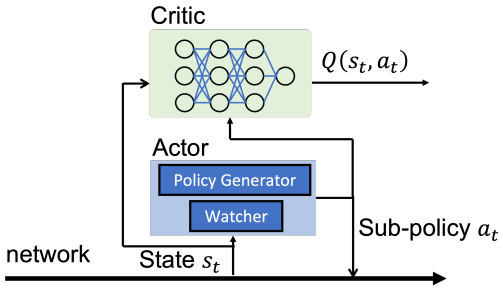


Figure 16: The actor-critic training algorithm in Spine.

The complete training procedure is as follows: firstly, the actor interacts with the network environment and collects a lot of trajectories consisting of tuples (s, a, r, h), where h is the hidden state generated by recurrent blocks in the model. While the training is performed in batches, here we simplify the case by considering one single sample. For every training step, we sample a long trajectory with burn-in prefix from collected training data $((s_0, a_0, r_0, h_0)...(s_{N-1}, a_{N-1}, r_{N-1}, h_{N-1}), (s_N, a_N, r_N, h_N))$, and initialize our recurrent model with the first hidden state h_0 at the head of the trajectory. Then, we unroll the recurrent network model on

the trajectory until the end to get the model outputs of s_{N-1} and s_N, which is the target state we will train on for this sample. For convenience, we let (s, a, r, s', a') denotes $(s_{N-1}, a_{N-1}, r_{N-1}, s_N, a_N)$ in the following objective functions.

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

$$\mathcal{J}(\theta) = \mathbb{E}\left[Q_\omega(s, \pi_\theta(s))\right], \tag{16}$$

where $Q_\omega(s, a)$ is the output of the critic that estimates the action-value function $Q^{\pi_\theta}(s, a)$ under the current policy. The policy gradient theorem [pgt] is adopted to optimize the distribution π_θ based on Equation 16. In order to estimate the action-value function accurately, the critic is also trained to minimize the objective function based on temporal difference learning [40]:

$$\mathcal{L}(\omega) = \mathbb{E}_{s,a,r,s'}\left[\left(Q_\omega(s, a) - r + \gamma Q_\omega(s', a')\big|_{a'=\boldsymbol{\pi_\theta}(s')}\right)^2\right]. \tag{17}$$

We adopt gradient-based learning algorithm for optimization. With the probabilistic trigger unit, the gradient can flow back to the watcher and previous time steps. The back-propagation stops before the burn-in steps, as they are only used as a warm start in the forwarding phase. After calculating the gradients, SPINE update the actor and critic models with learning rates α and η:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{J}(\theta), \qquad \omega \leftarrow \omega - \eta \nabla_\omega \mathcal{L}(\omega). \tag{18}$$

We also adopt several RL-related training tricks used in TD3. The clipped double Q-learning, delayed policy updates and target policy smoothing regularization techniques introduced in TD3 are all used in SPINE's training to reduce the variance of the critic model. We refer the readers to [13] for the details of these techniques.

C TRAINING HYPERPARAMETERS

Here we give the training hyperparameters of SPINE in Table 6.

Name	Value
learning rate (α)	0.005
gamma (γ)	0.98
batch size	64
model update interval (second)	5
model update step	20
action control coefficient (α)	0.05
monitoring time inverval (ms)	30
pit stop penalty (α_{psp})	0.03

Table 6: Training hyperparameters in SPINE.