Dynamic TCP Initial Windows and Congestion Control Schemes through Reinforcement Learning

Xiaohui Nie[†¶], Youjian Zhao[†¶], Zhihan Li[†¶], Guo Chen[‡], Kaixin Sui[§], Jiyang Zhang[††], Zijie Ye[†],Dan Pei[†¶*] ,
[†]Tsinghua University [‡]Hunan University [§]Microsoft Research [††]Baidu
[¶]Beijing National Research Center for Information Science and Technology (BNRist)
Email: {nxh15, lizhihan17, yezj16}@mails.tsinghua.edu.cn, {zhaoyoujian,peidan}@tsinghua.edu.cn,
guochen@hnu.edu.cn, kasui@microsoft.com, zhangjiyang01@baidu.com

***Abstract*—Despite many years of improvements to it, TCP still suffers from an unsatisfactory performance. For services dominated by short flows (e.g., web search, e-commerce), TCP suffers from the *flow startup problem* and cannot fully utilize the available bandwidth in the modern Internet: TCP starts from a conservative and static initial window (*IW*, 2-4 or 10), while most of the web flows are too short to converge to the best sending rate before the session ends. For services dominated by long flows (e.g., video streaming, file downloading), the congestion control (*CC*) scheme manually and statically configured might not offer the best performance for the latest network conditions.**

To address these two challenges, we propose *TCP-RL*, which uses reinforcement learning (*RL*) techniques to dynamically configure *IW* and *CC* in order to improve the performance of TCP flow transmission. Basing on the latest network conditions observed at the server side of a web service, *TCP-RL* dynamically configures a suitable *IW* for short flows through group-based *RL*, and dynamically configures a suitable *CC* scheme for long flows through deep *RL*. Our extensive experiments show that for short flows, *TCP-RL* can reduce the average transmission time by about 23%; and for long flows, compared with the performance of 14 *CC* schemes, *TCP-RL*'s performance ranks top 5 for about 85% of the 288 given static network conditions, whereas for about 90% of conditions, its performance drops by less than 12% compared with that of the best-performing *CC* schemes for the same network conditions.

***Index Terms*—TCP Initial Window, Congestion Control, Web Service, Reinforcement Learning**

I. Introduction

Nowadays, most online web services (e.g., Microsoft [2], Baidu [3]) are based on TCP transmission; TCP performance directly affects user experience and company revenue [4, 5]. However, despite many years of improvements to it, TCP still suffers from an unsatisfactory performance [6, 7]. In this paper, we focus on two well-known TCP performance problems: (1) TCP cannot deal with short flows gracefully [1], and (2) the performance of congestion control (CC) algorithms remains far from ideal [8].

In the first problem, TCP begins the transmission with a slow start phase to probe the available bandwidth, and then it uses a *CC* algorithm to converge to the best sending rate. Specifically, TCP starts with a conservative and static initial congestion window (*IW*, 2, 4, or 10) [9], and then it tries to find the best sending rate of a flow by keeping probing and adjusting its congestion window (*CWND*) during flow transmission. However, most web flows are so short that they could be finished in just one RTT with the best *CWND*, but they inadequately take multiple RTTs to probe for their optimal *CWND* within the slow-start phase. As a real-world example, Table I shows that for the mobile search service in a top global search company, *Baidu*, where *IW* =10, more than 80% of TCP flows are still in the slow start phase when the sessions end; they do not fully utilize the available bandwidth.

TCP state after flows end	% of flows
Slow Start	80.27%
Congestion Avoidance	19.73%

TABLE I: Distribution of TCP states after flows end. Measured in the mobile search service of *Baidu* during one week in 2017. The average flow size is about 120 KB.

The above *TCP flow startup problem* is still considered by the research community as an open research problem [10] for the general TCP environment. Google proposed increasing the standard *IW* from 2-4 to 10 [9]. But is 10 still too small for high-speed users (e.g., with fiber access), or is 10 too large for low-speed users (e.g., GPRS access in remote areas)? As network conditions can be highly variable both spatially and temporally, choosing a static *IW* that is best for all flows is infeasible.

In the second problem, an efficient *CC* algorithm is critical for data transmission, especially for services dominated by long flows, such as video streaming and large file downloading. With the rapid development of network techniques and Internet infrastructure in the past few decades, many variants of *CC* schemes (e.g., Tahoe, Reno [11], Cubic [12], BBR [13], PCC Vivac [7], Copa [14], Indigo [8]) have been proposed. However, a recent study in Pantheon [8] shows that the performance of different *CC* schemes varies significantly across various network conditions, and there is no single *CC* scheme that can outperform all others in all network conditions. As the network conditions for the same web service can vary temporally (across time) and spatially (across different users), to achieve the best TCP performance for long flows, a web service might need different *CC* schemes across different time and/or users. Yet, the common practice is that a web service's

* This paper significantly extends [1] by adding dynamic congestion control schemes through deep reinforcement learning.

* Dan Pei is the corresponding author.

providers *manually* and *statically* configure one specific *CC* scheme for the web service and stick to it, leaving the TCP performance undesirable.

To address the two problems above, we argue that *a web-service provider can use reinforcement learning RL methods to dynamically configure IW and CC for improving the network transmission performance in the Internet.* Basically, network conditions determine the ideal value of *IW* and *CC*, so the process of configuring *IW* and *CC* involves building a mapping between *states* (network conditions) and *actions* (*IW*, *CC*). Here we cast above two problems as *RL* problems because *RL*'s basic idea is to maximize some notion of cumulative reward through building the mapping between environment *states* and *actions*. Our choice of *RL* is inspired and encouraged by recent progress in applying *RL* to Internet video QoE optimization through dynamically deciding a video session's serving frontend servers [15] or through tuning ABR algorithms [16].

In this paper, to maximize the cumulative reward (network performance, which needs to be continuously monitored), *RL* is used to continuously update the decisions of the suitable *IW* for services dominated by short flows, and the suitable *CC* scheme for services dominated by long flows. Although applying *RL* to configure *IW* and *CC* is a promising abstraction, there are some major challenges in practice:

- **Challenge 1**: *How to measure the fresh TCP data on the server side only? RL* methods need fresh data to compute reward and the states. However, a traditional web service server cannot directly measure some TCP data (e.g. transmission time) without clients' collaboration.
- **Challenge 2**: *How to apply RL methods on highly variable and non-continuous network conditions of the Internet? RL*'s decisions are determined by a static (but unknown) distribution of the context. *RL* methods require the continuity of the context that affects the reward of the decision [17, 18]. In our case, the context is a flow's **network conditions (i.e., available end-to-end bandwidth and RTT)**, but these are highly variable across different times and user granularities (i.e. IP or subnet). Using which *RL* methods in which user granularity is a problem.
- **Challenge 3**: *How to rapidly search for the optimal TCP IW or CC from a large decision space? RL* methods are essentially based on *trial and error*. They require clever exploration mechanisms. Brute-forcely selecting actions results in poor performance [19] and typically only suits a small and limited decision space, which can quickly converge to the best decision after a small number of trials. However, the search space for *IW* (ranging from 2 to more than 100) and *CC* (at least 14 variants [8]) is so large that the network conditions might have already changed before brute-force searching can find the optimal *IW* or *CC* for the previous network conditions.

In this paper, to address the above challenges, we propose a system called *TCP-RL* that can automatically and dynamically configure *IW* at the server side through group-based *RL*, and can automatically and dynamically configure *CC* through deep *RL*. The contributions of this paper are summarized as follows:

- To address challenge 1, we modify the Linux kernel and Nginx software to enable web servers to collect and store billions of TCP flow performance records (e.g., transmission time, throughput, loss rate, RTT) in real time without any client assistance (§VI).
- To address challenge 2 for the *IW* configuration problem, we apply a traditional model-free *RL* method at the group granularity, because *IW* should be configured before transmission when there is no state information (network condition). The basic idea is to apply online exploration-exploitation [15]. Since fine-grained user groups (i.e., IP) could have too few data samples to detect context continuity, we propose a bottom-up approach to group flows from users with the same network features in order to find the most fine-grained user groups that both have enough samples and satisfy the *RL*'s context continuity requirements [17, 18]. Compared with previous works [9, 20, 21], *TCP-RL* utilizes much richer historical information from the user group to help configure the suitable *IW*.
- To address challenge 2 for the *CC* configuration problem, *TCP-RL* uses a deep *RL* method at per-flow granularity. It trains an offline neural network model that can be used online to select the suitable *CC* for the given network conditions. Furthermore, in order to handle the variable network conditions, we build a model that detects changes of network conditions to help the deep *RL* method adapt to the network changes over time. Instead of building a new *CC* [7, 8, 11–14], *TCP-RL* proposes to use deep *RL* to dynamically configure the right *CC* schemes for different network conditions on a per-flow level.
- To address challenge 3, for the *IW* configuration problem, we improve *RL* with a fast decision space searching algorithm. Based on the common perception of the relationship between TCP performance and *IW*, we propose a sliding-decision-space method that can quickly converge to the best *IW* (§IV-A). For the *CC* configuration problem, our neural network model trained offline can directly select the right *CC* online without brute-force searching.
- To the best of our knowledge, *TCP-RL* is the first work to solve *IW* and *CC* configuration problems through *RL*. It is much more applicable, and only needs to modify the sender side. *TCP-RL* has been deployed in one of the top global search engines for more than a year. Our online and testbed experiments show that for short flow transmission, compared with the common practice of $IW = 10$, *TCP-RL* can reduce the average transmission time by 23% to 29%. For long flows, compared with the performance of 14 *CC* schemes, *TCP-RL*'s performance ranks top 5 for about 85% of the 288 given static network conditions, whereas for about 90% of conditions, its performance drops by less than 12% compared with that of the best-performing *CC* schemes for the same network conditions.

The rest of the paper is organized as follows: §II provides the background of this work, §III introduces the core idea

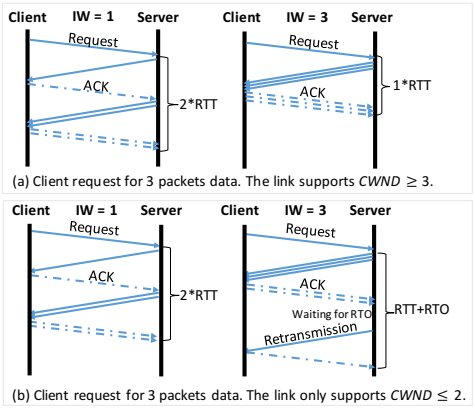


Fig. 1: An illustrative example to show the effect of *IW*.

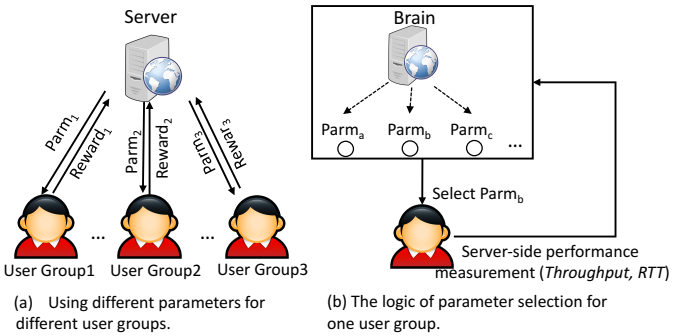


Fig. 2: The key idea of *TCP-RL*. *Parm* is *IW* or *CC*.

and overview of *TCP-RL*, and §IV and §V describe the algorithms for the configuration of the *IW* and *CC* schemes, §VI presents the implementation details of *TCP-RL*, §VII and §VIII systematically evaluate *TCP-RL*'s performance on the *IW* and *CC* configuration, respectively, §IX presents related works, and §X concludes the paper.

II. Background

A. The Preliminary of TCP Congestion Control

TCP is one of the main Internet protocols providing reliable, ordered, and end-to-end transportation service. *CC* is TCP's core algorithm to determine the rate of data transmission. A good *CC* should efficiently utilize network resources, guarantee fair rates when multiple senders compete over network resources, and be easily deployable. Despite thirty years of research on *CC*, it is still a challenging task in the Internet. The reason is that the ideal sending rate is determined by the link (router, switch, etc.) between the sender and the receiver. However, the sender does not have a global view and actually has hardly any information about the link *before data transmission*. Thus, most of the *CC* schemes take a conservative approach by starting transmission from a small sending rate, and then adjust the sending rate during the transmission with some strategies (e.g., AIMD). However, probing from a small sending rate cannot deal with short flows gracefully because these flows could have finished their transmission when TCP is still at a conservative sending rate. Furthermore, although many versions of *CC* schemes have been proposed, there is no scheme that works best in all network conditions [8].

B. Initial Window and Short Flows

For short flow transmission, the transmission time is the key performance metric. As shown in §I, for services dominated by short flows, most of the flows end the transmission without exiting the slow start phase. The initial window (*IW*) is a key parameter that determines the initial sending rate. The common perception of *IW* is that too small an *IW* suffers from more RTTs than necessary to finish the transmission; too large an *IW* results in congestion or even an expensive TCP timeout, which results in a high network transmission time. Thus, different *IWs* can significantly affect the transmission time of short flows. Fig. 1 shows *IWs*' effects in two example network conditions. In Fig. 1(a), the network condition is relatively good as the link can support *CWND* = 3. If the server's *IW* = 3, the TCP response time is 1 RTT. But when *IW* = 1, the response time is 2 RTTs. In Fig. 1(b), the network condition is worse, i.e., when the *CWND* is larger than 2, it would cause link congestion and packet loss. As such, if *IW* = 3, the response time would be RTT + retransmission timeout (RTO, which is typically several times of RTTs). But if *IW* = 1, the response time is 2 RTTs, shorter than that when *IW* = 3. We can see that a proper *IW* can significantly reduce the network transmission time.

C. Congestion Control and Long Flows

For long flow transmission, achieving higher throughput and lower RTT are the goals of network transmission. Simply tuning *IW* might reduce only a few RTTs, so it is not very helpful to long flow transmission with hundreds or thousands of RTTs. However, achieving both high throughput and low RTT is quite challenging, and that is why many *CC* schemes are being proposed. Some of them are heuristic (Cubic [12], BBR [13]), some of them are generated by machine learning (TCP Remy [22], Indigo [8]). However, none of the existing algorithms can consistently perform optimally in diverse network conditions in the real Internet [8]. §VIII compared the performance of 14 *CCs* on 288 kinds of synthetic networks, whose bandwidth, RTT, and loss rate are sampled from 1-200Mbps, 10-200ms and 0-10%. The results show that 10 different *CCs* performed best in at least one network condition. Ideally, if we can somehow choose an appropriate *CC* for each specific network condition, the transmission performance can be significantly improved.

III. Core Ideas and System Overview

As mentioned in §II, *IW* and *CC* work on different scenarios. For short flows, *IW* is the major factor that affects TCP performance, and the flow ends its transmission before *CC* takes effect. For long flows, *CC* is the key to achieving good performance, whereas *IW* has little impact. Therefore, we treat configuring *IW* and configuring *CC* separately.

In theory, the optimal *IW* or *CC* is determined by the client-server end-to-end link's network conditions (e.g., available bandwidth and RTT), but Internet network conditions are highly variable, both temporally and spatially. Ideally, we

should configure an appropriate *IW* or *CC* for each new flow, but doing so is challenging without enough knowledge of network conditions.

For the *IW* configuration problem for short flows, learning *IW* in a single flow is impossible because before configuring *IW*, the sender cannot observe the state of the network environment. To deal with the variability of network conditions, we propose a bottom-up approach to group flows from clients (users) with the same **network features (e.g., *subnet, ISP, province*)** to find the most fine-grained user groups that both have enough samples and satisfy the *RL*'s context continuity requirements; then, we apply an online *RL* method in each user group. On the one hand, using an online *RL* method can naturally deal with the temporal variability of network conditions, i.e., *RL* aims to *find the best IW scheme for each given network condition of a specific user group and dynamically adapt to the latest network conditions of the user group*. On the other hand, user grouping is used to handle the spatial variability of network conditions. Our intuition is that for a given server at a given time, users' network features (i.e., *subnet, ISP, province*) largely determine the client-server end-to-end link's network conditions. If we run *RL* for each user group, within which the network conditions are similar, the performance of each group can be improved.

For the *CC* configuration problem for long flows, during the transmission, the sender can observe the network state (i.e., *throughput, RTT, loss rate*), and there is enough time to make the decision *during* the transmission. Thus, we propose to dynamically configure *CC* at the most fine-grained granularity (i.e., flow level). On the other hand, unlike *IW*, which is a numerical value, *CC* schemes are categorical; thus, online brute-force searching for *CC* is inefficient. We thus train a neural network model offline to build the mapping from the state (network conditions) to the best actions (*CC*). When a new flow starts transmission, the model can dynamically configure the suitable *CC* according to the state (network condition) it observed. In this way, it can improve TCP performance in variable network conditions.

A. Why reinforcement learning

Choosing a proper *IW* or *CC* scheme for a TCP flow is not an easy task. Using the data-driven method is a promising direction, but even if we have logged the networking conditions in details, choosing a proper *IW* or *CC* scheme is still difficult because it is highly related to multiple complex factors such as network bandwidth, RTT, router buffer size, flow size and the end-to-end path between the user and the server. All these factors can frequently change over time, which means that the proper *IW* or *CC* scheme changes over time. *RL*, inspired by human behavior psychology [23], is a popular technique in the machine learning community and is very suitable to cope for addressing the above situation. Basically, it continuously makes decisions based on environment feedback. Once the optimization goal (called the *reward function* in *RL*) is properly defined, *RL* can gradually find the best decision based on *trial and error*, by striking a dynamic balance between exploring suboptimal decisions and exploiting currently optimal decisions. Moreover, its *exploration and exploitation* algorithm can quickly react to the environment change. As such, *RL* naturally fits the task of dynamically configuring *IW* or *CC*.

B. Overview of TCP-RL

The key idea of *TCP-RL* is shown in Fig. 2. The frontend server can dynamically configure *IW* or *CC* for different user groups. For each user group, the server collects its TCP performance data (throughput, RTT, etc.) periodically, which make up *RL*'s state and reward. Then, it reports the data to the server, which runs per-group *RL* with fresh data, and the server acts as the brain for learning each user group's new *IW* and *CC* in the next time step. Note that, all procedures are done at the server side without any client or middleware (e.g., router, switch, link) modification or assistance.

Specially, for the *IW* configuration, the user groups are the flows with the same network features (e.g. subnet, ISP, province). When the frontend server receives a request from a user, it establishes a TCP session with the user and identifies the user group it belongs, and then it obtains the most up-to-date decision of *IW* from the per-group *RL*'s result and configures the *IW* for the session quickly before sending the response to the user. After the transmission of the data response is completed, the frontend server outputs the TCP performance data to the brain server in order to run *RL*. Besides, the brain runs the user grouping algorithm with the historical data. The brain continuously sends the decision of each user group to the frontend servers at a timescale of minutes. This way, it controls all sessions' behavior.

For the *CC* configuration, the user group is at per-flow granularity, and the brain server is the frontend server itself, which means that *TCP-RL* dynamically configures the *CC* for each TCP flow. When a TCP flow's connection is established, the model first randomly configures one *CC*, and then it can obtain the network state and reward from the network environment. The *RL*'s neural network model takes the value of the state and reward as the input, and then it outputs and configures the new *CC* for this flow. This procedure is conducted repeatedly at a timescale of seconds. The neural model is trained offline, and this will be introduced in §V.

Because *IW* configuration and *CC* configuration work separately, the servers serving services (e.g. mobile search) dominated by short flows can deploy *TCP-RL*'s *IW* configuration, and the servers serving services (e.g. video streaming) dominated by long flows can deploy *CC* configurations.

IV. *RL* for *IW* Configuration

In this section, we present two *TCP-RL* core algorithms for *IW* configuration: the online *RL* algorithm for learning per-group *IW* given a user group (§IV-A) to address challenge 3 in §I, and the user grouping algorithm (§IV-B) to address challenge 2 in §I.

A. Online Reinforcement Learning

In this paper, we formulate the *IW* learning problem as a non-stationary multi-armed bandit problem. It focuses on

Algorithm 1 The discounted UCB

1: for t from 1 to K, play arm $I_t = t$
2: for t from K + 1 to T, play arm

$$I_t = \arg\max_{1 \le i \le K} \overline{X}_t(\gamma + i) + c_t(\gamma + i)$$

online performance by striking a balance between exploration (uncharted actions) and exploitation (current optimal actions). Many algorithms for this problem have been proposed [23].

For the stationary multi-armed bandit problem, the basic UCB algorithm [24] has been shown to be quite efficient [25]. Its assumption is that the unknown distribution of the context (network conditions) does not change over time. However, in our scenario, the network conditions could change over time, making our *RL* problem a non-stationary bandit problem. In this paper, we use the discounted UCB algorithm [25], which was proposed to solve the non-stationary bandit problem. The intuition is that, to estimate the instantaneous expected reward of each *IW*, it averages past rewards with a discount factor that gives more weight to recent observations. The basic procedure is shown in Algorithm 1. At each time t, the player chooses an arm $I_t \in 1, ..., K$ (a decision) with the highest expected upper-confidence $\overline{X}_t(\gamma, i) + c_t(\gamma, i)$. $\overline{X}_t(\gamma, i)$ is the discounted empirical average reward shown in Equation 1. $X_s(i)$ denotes the arm i's instantaneous reward at time s. When $I_s = i$, $\mathbb{1}_{\{I_s=i\}}$ = 1; otherwise, $\mathbb{1}_{\{I_s=i\}}$ = 0. $\gamma \in (0, 1)$ is a discount factor to calculate the average reward.

$$\overline{X}_t(\gamma, i) = \frac{1}{N_t(\gamma, i)} \sum_{s=1}^{t} \gamma^{t-s} X_s(i) \mathbb{1}_{\{I_s=i\}} \tag{1}$$

$$N_t(\gamma, i) = \sum_{s=1}^{t} \gamma^{t-s} \mathbb{1}_{\{I_s=i\}} \tag{2}$$

$c_t(\gamma, i)$ is the discounted padding function defined in Equation 3, where B is an upper bound on the rewards and $\xi > 0$ is an appropriate constant variance to control the probability of exploration. Note that if one arm is frequently used in the history, its $c_t(\gamma, i)$ is smaller than that of the other arms, so the suboptimal arm can be used for exploration. In this way, the algorithm can strike a balance between exploration and exploitation.

$$c_t(\gamma, i) = 2B\sqrt{\frac{\xi \log n_t(\gamma)}{N_t(\gamma, i)}}, n_t(\gamma) = \sum_{s=1}^{K} N_t(\gamma, i) \tag{3}$$

To successfully apply the above general discounted UCB framework to a given scenario (*IW* configuration), the key is to appropriately define the reward function and the arms.

Reward function definition: Services dominated by short flows are sensitive to the TCP response time [1, 6], which is the flow completion time. Our goal is to configure the ideal *IW*, which can fully utilize the end-to-end link's bandwidth without causing congestion; thus, our reward function needs to somehow capture both bandwidth utilization and congestion level. For bandwidth utilization, the throughput (the number of bytes transmitted per unit of time) is a better metric than the TCP response time because the latter is affected by the size of flows, which can still vary largely even for different short flows in web services. On the other hand, considering congestion is also necessary because simply increasing *IW* to obtain the best throughput of flows that deploy our approach (called *TCP-RL flows*) can hurt the performance of *non-TCP-RL flows* that share some network resources with *TCP-RL* flows[1]. Given that RTT has been successfully used to reflect the congestion levels in many well-known *RTT*-based *CC* algorithms [26, 27], we select RTT as the signal of network congestion.

Based on the above analysis, our goal for the reward function is to maximize the throughput and minimize the RTT, as shown in Equation 4. We normalize the reward because UCB requires the reward to be within [0, 1]. $Throughput_s(i)$ is arm i's instantaneous throughput at time s, and $RTT_s(i)$ is arm i's RTT at time s. $throughput_{max}$ is the maximum throughput in the history measurements, and RTT_{min} is the minimum RTT in the history measurements. In case of the normalized reward being out of [0,1], before computing the reward, *TCP-RL* will check whether to update $throughput_{max}$ or RTT_{min} with the new observations. If the new $throughput > throughput_{max}$, $throughput_{max} = throughput$; if the new $rtt < RTT_{min}$, $RTT_{min} = rtt$. *TCP-RL* will recalculate the reward with the latest $throughput_{max}$ and RTT_{min}. Because all the arms (*IWs*) of the discounted UCB use the same $throughput_{max}$ and RTT_{min}, the order of their reward values remains the same after the above renormalization. In this way, the decision of the discounted UCB will not be affected. α is the parameter that strikes a balance between the throughput and RTT. Small α favors a low RTT, which may make the algorithm conservative with a small *IW*. Large α favors a high throughput, which may make the algorithm aggressive with a large *IW*.

$$X_s(i) = \alpha * \frac{Throughput_s(i)}{Throughput_{max}} + (1 - \alpha) * \frac{RTT_{min}}{RTT_s(i)} \tag{4}$$

Arms definition: The list of arms in a discounted UCB is the decision space with some discrete values. However, *IW* has a continuous and large value space. Our goal is to find the best *IW* in a large decision space quickly. Brute-forcely searching the entire decision space is inefficient because too many arms will waste time in the exploration procedure. To address this problem, we propose a sliding-decision-space method based on the *common perception* (mentioned in § II-B) about the relationship between TCP performance and *IW*. At first, we start with a short list of *IWs* as the arms, and the value in the arm list is dynamically adjusted based on the arms' performance.

The sliding-decision-space approach is illustrated in Fig. 3. We use n *IWs* as the initial arms list (e.g., $n = 4$, *IWs* = [15, 20, 25, 30]). When updating the decision, we will first check whether the arm list should be updated. The basic idea is to check whether the largest arm IW_{large} or the smallest arm IW_{small} is currently the best arm. If yes, we update the arm list; otherwise, the arm list remains the same. The best arm is

[1]If *TCP-RL* only considers throughput as the reward, it will aggressively maximize its own throughput and occupy other *non-TCP-RL flows'* bandwidth. The reason is that, when *TCP-RL* flows increase their *IWs* and cause network congestion, the other *non-TCP-RL flows* will reduce their *CWND* and occupy less bandwidth, but *TCP-RL flows* will continue to increase their *IWs* in the next time because they can obtain larger throughput (reward) as a result of the *non-TCP-RL flows* occupying less bandwidth.

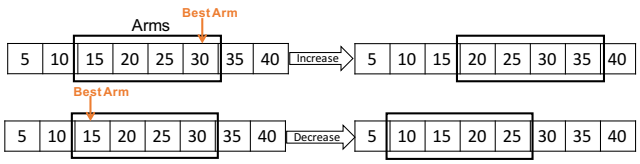


Fig. 3: The procedure of the sliding-decision-space method.

the one which has the largest reward and the smallest value of padding function in Equation 3. The smallest value of padding function means the arm has been exploited more frequently than the other arms. Based on the *common perception of IW* (§ II-B), a too large or a too small *IW* is sub-optimal. If the current best arm is IW_{large}, we will add a new *IW* ($IW_{large}+\triangle$) to the arm list and delete IW_{small}. If the current best arm is IW_{small}, we will add a new IW ($IW_{small}-\triangle$) to the arm list and delete IW_{large}. $\triangle$ is the constant step size for searching the *IW* space. If the current best *IW* is not the largest or smallest, the arm list remains the same.

With the above sliding-decision-space approach, although the best arm selected in each step might not be the eventual ideal *IW*, it is gradually approaching the eventual ideal *IW*. Such a design has two benefits: it reduces the search space from a larger number of potential *IWs* to only n *IWs* in the arms window, and it avoids being too aggressive in increasing the *IWs*, helping avoid congestion. This successfully addresses challenge 3 in §I for the *IW* configuration.

B. User Grouping

In reality, users' network conditions highly vary because users have different network features (i.e., *subnets, ISP, province*). For users coming from different provinces (e.g., *Beijing, Shanghai*) and ISPs (e.g., *CHINANET, CMNET, UNICOM*), their network conditions (e.g., bandwidth and RTT) could be different. To apply *RL* in highly spatially variable network conditions, users with different network conditions should be treated differently.

The flow's *IW* is determined by its end-to-end link's network conditions (i.e., bandwidth and RTT). The ideal solution would be learning *IW* for each link. However, each link hardly has enough samples for *RL* to learn a suitable *IW* or *CC* scheme. Thus, we argue that grouping users with similar network features to share their samples is a promising solution. However, this is challenging because of the following dilemmas: 1) A too fine-grained user group (e.g., *IP*) typically lacks enough samples to monitor its network performance continuously, so it cannot satisfy the requirement of *RL*; 2) A too coarse-grained user group (e.g., all flows) leads to suboptimal performance.

To address the above problem, we propose a new user grouping method. *The goal of user grouping is to find the most fine-grained user groups that can satisfy the RL's requirement (i.e., maintains continuity in network conditions).* The basic idea is to use a bottom-up (finest-to-coarsest) searching technique in order to find the finest user groups, each of which has enough samples and keeps the continuity of the network conditions. We quantify the network conditions with the reward function in §IV-A, which considers both throughput and RTT.

Subnet ($IP_{start} \sim IP_{end}$)	***ISP***	***Province***
S1 (223.72.97.0∼223.72.98.255)	CMNET	Beijing
S2 (223.71.208.0∼223.71.208.255)	CMNET	Beijing
S3 (123.118.89.0∼123.118.93.255)	UNICOM	Beijing
S4 (114.243.33.0∼114.243.33.255)	UNICOM	Beijing
S5 (123.120.72.1∼123.120.72.189)	UNICOM	Beijing
S6 (219.143.38.0∼219.143.38.255)	CHINANET	Beijing
S7 (58.130.48.0∼58.130.54.255)	CHINANET	Beijing
S8 (58.131.131.0∼58.131.132.255)	CHINANET	Beijing

TABLE II: An example of *Baidu* company's geolocation database.

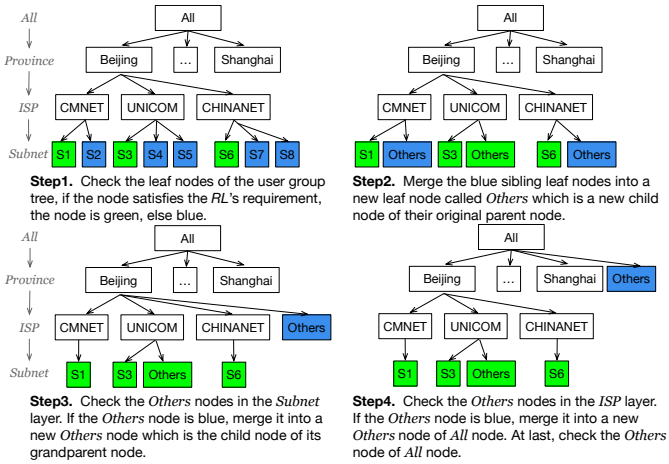


Fig. 4: Procedure of the user grouping algorithm.

More specifically, before the data transmission, IP is the most fine-grained user group because the server at that time can only obtain the IP as the user's network feature. By looking up *Baidu* company's geolocation database by IP, which is similar to the geolocation database [28], we can infer the other network features, such as *subnet, ISP and province*. Table II shows an example of the geolocation database. Note that one given IP can only appear in one record of the features in the table, and all the records are mutually exclusive in IP space. Thus, the structure of the user grouping result forms a four-layer (subnet, ISP, Province, All) tree, as Fig. 4 shows.

We say that a user group has enough samples when it has at least S_{min} samples in a time bin with length t. For each time bin, we calculate the distribution of the reward and use the average reward to quantify the network condition of this time bin. In this way, we obtain a time series of the average reward to characterize the changes of network conditions.

We then define a metric called network jitter J, shown in Equation 5, to capture the continuity of network conditions. n denotes the number of time bins, and X_s is the reward at time bin s. As *IW* or *CC* scheme affects the reward, when computing J, *IW* or *CC* scheme should remain the same in each time bin. Note that a small J means the change in network condition is small. To apply the *RL* method to a given user group, the smaller the J, the better. Here, we choose a threshold T; if the user group has $J \leq T$, it satisfies the requirement of *RL*.

$$J = \frac{\sum_{s=2}^{n} |X_s - X_{s-1}|/X_s}{n-1} \tag{5}$$

In the example, we assume that the finest users' network feature is the *subnet*, and the coarsest feature is *All*. *Beijing*

has three ISPs, namely *CMNET*, *UNICOM*, and *CHINANET*, and they have 8 subnets which are *S1*∼*S8*. The user grouping algorithm has 4 steps:

- Step 1: We check whether all the leaf nodes can satisfy the *RL*'s requirement (Equation 5). The example's result is that *S1, S3*, and *S6* (in green color) satisfy the *RL*'s requirement, and *S2, S4, S5, S7*, and *S8* (in blue color) do not, so *S1, S3*, and *S6* are the three finest user groups that can use *RL* to learn *IW*.
- Step 2: The sibling leaf nodes that cannot satisfy the *RL*'s requirement are merged into a new leaf node called *Others*, which is a new child of their original parent node. In the example, *S2* is turned into *Others*, a new child node of *CMNET*. *S4* and *S5* are merged into *Others*, a new child node of *UNICOM*. *S7* and *S8* are merged into *Others*, a new child node of *CHINANET*.
- Step 3: We check whether *Others* nodes satisfy the *RL*'s requirement. If an *Others* node does not meet the requirement, it needs to be merged with the *Others* nodes of its parent (ISP)'s sibling, and form a new child *Others* node of its original grandparent (Province). In the example, the node *Others* of *CMNET* and the node *Others* of *CHINANET* are merged into *Others*, a new child node of *Beijing*. The node *Others* of *UNICOM* satisfies the requirement because it has sufficient samples to measure its network conditions after merging *S4* and *S5*.
- Step 4: The algorithm continues to check the leaf *Others* nodes until all leaf nodes (except the root's child *Others* node) satisfy the *RL*'s requirement. Finally, if the *Others* node of *All* does not meet the requirement, we use the standard *IW* [9] for its flows. In the example, the leaf nodes, except the *Others* of *All* node, are the user groups that can use *RL* to learn *IW*.

When one user group (e.g., S2, S4, S5) cannot satisfy the *RL*'s requirement, its data will be merged into the *Others* node with the same parent node. Finally, if the leaf *Others* node cannot satisfy the *RL*'s requirement, its data will be merged into the *Others* node one level up (a sibling node of the current *Other* node's parent node). Notice that the *Others* nodes are used to group fine-grained users who cannot satisfy the *RL*'s requirement. The *Others* nodes at an upper level would have more data samples to describe the distribution of context, so there is a larger chance to satisfy *RL*'s requirement.

In summary, to address challenge 2 (in §I) for the *IW* configuration problem, we apply model-free *RL* methods at the group granularity because *IW* should be configured before transmission when there is no state information (network condition). The basic idea is to apply online exploration-exploitation [15]. As a fine-grained user group (i.e. IP) could have too few data samples to detect context continuity, we propose a bottom-up approach to group flows from users with the same network features in order to find the most fine-grained user groups that both have enough samples and satisfy the *RL* context continuity requirement [17, 18].

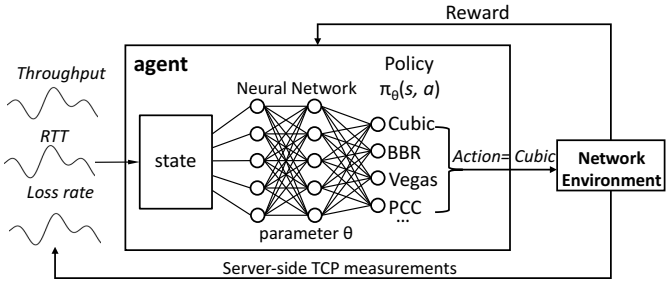


Fig. 5: Applying deep reinforcement learning to *CC* configuration.

V. *RL* for *CC* Configuration

A. Overview

The optimal *CC* scheme on a specific network condition can be different across different network conditions [8]. However, the current practice for the *CC* configuration is that the operators of a web service manually and statically configure a *CC* from a few variants and stick to it at least for a while. This approach cannot handle the dynamic network conditions across spatial and temporal dimensions; different users of the same service can have different access networks with different network conditions, and even the same user and the same long flow might have different network conditions at different times.

How to select the appropriate *CC* during a long flow's lifetime has been seldom studied in the literature. To address this problem, *TCP-RL* uses a deep *RL* method at per-flow granularity. It trains an offline neural network model that can be used online to select the suitable *CC* for given network conditions. Furthermore, in order to handle the variable network conditions, we build a model that detects the changes in network conditions, to help the deep *RL* method adapt to the network changes over time. This flow-level automatic and dynamic *CC* configuration is a significant improvement over service-level static and manual *CC* configuration given the temporal and spatial (across different users) dynamics of network conditions.

TCP-RL applies *RL* to learn the *CC* configuration policy purely through *experience*. However, directly using the online exploration-exploitation mechanism (as used in *TCP-RL IW* configuration) is inefficient, as there are about 14 *CC* schemes, and there is no clear performance order among these *CC* schemes under each condition. It would waste much time in online exploration. Therefore, *TCP-RL* uses modern deep *RL* techniques [29] to dynamically configure *CC*. Fig. 5 illustrates how deep *RL* can be applied to the *CC* configuration. The actions of the agent are the *CCs*, and the state of the agent consists of the TCP measurement data (throughput, RTT, loss rate) at the server side. The policy of choosing *CC* is determined by the neural network model, which builds a mapping from the observed network states to the actions (selected *CCs*).

B. A3C Background

TCP-RL uses A3C [29], a state-of-the-art deep *RL* algorithm. It has been successfully applied to many problems,

such as the dynamic selection of the bitrate for video streaming [16], and to Atari games [29]. A brief description of A3C is as follows:

Input: The learning agent of *TCP-RL* takes state inputs $s_t = (Throughput_t, RTT_t, Loss_t)$ to its neural network.

Policy: After receiving s_t, the agent selects actions (*i.e.*, the *CC* for each flow) based on the *policy* π, defined as a probability distribution over actions: $\pi(s_t, a_t) \rightarrow [0, 1]$. $\pi(s_t, a_t)$ is the probability that an action a_t is taken after a state s_t is observed. In practice, we train a neural network with adjustable parameters θ to maintain the policy $\pi_\theta(s_t, a_t)$. This way, the actor network takes the raw TCP measurement data as states and determines the corresponding action to be applied in the current state.

Reward: At each step t, the agent observes a state s_t and chooses an action a_t according to the actor-critic network. Then, the state of environment transitions to s_{t+1} and the agent receives a *reward* r_t from the action. The goal of the agent is to maximize the expected cumulative reward that it receives. The reward is defined as the TCP performance metric in Pantheon [8]: $r_t = \log(\frac{Throughput_t}{RTT_t})$; it is a version of Kleinrock's power metric [30], which aims to *keep the network pipe just full, but not too full*.

C. Training A3C Offline

Deep *RL* learns purely from experience. To generate a good neural network, *TCP-RL* first trains the neural network on many network conditions offline. The A3C algorithm uses a *policy gradient method* [31] to train its policy. The data needed for offline training are generated as follows. We build an emulation environment similar to that used in Pantheon [8], and we run flows with different *CCs* under different network conditions. More details can be found in §VIII-A.

Below, we briefly introduce the algorithm and intuitively explain how the algorithm can be applied in our task. The main idea of the method is to estimate the gradient of the expected cumulative reward by observing a series of executions obtained by following the policy. The gradient of the cumulative reward can be computed as follows [29]:

$$\nabla_\theta \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r_t] = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a) A^{\pi_\theta}(s, a)] \quad (6)$$

where θ are the parameters of the policy network, γ is a discounting factor, and $A^{\pi_\theta}(s, a)$ is the advantage function, which represents the difference in the expected discounted returns from a set of experiences after we pick an action a_t, compared with the expected reward for actions drawn from policy π_θ. Intuitively, the advantage function tells the agent how much better its actions turned out to be than expected, and then the network is updated to encourage or discourage actions appropriately.

In practice, the agent samples a trajectory of actions and empirically calculates the advantage $A(s_t, a_t)$ as follows:

$$A(s_t, a_t) = r_t + \gamma V(s_{t+1}; \theta_v) - V(s_t; \theta_v) \quad (7)$$

where $V(\cdot; \theta_v)$ is the estimate of $v^{\pi_\theta}(\cdot)$, which is the value function that can be obtained from the critic network. Note that following the suggestions in [29], we share some of the parameters between the actor network and the critic network. Typically, we use a network that has one softmax output for the policy $\pi_\theta(s_t, a_t)$ and one linear output for the value function $V(s_t; \theta_v)$, with all non-output layers shared. The network structure is shown in Fig. 6.

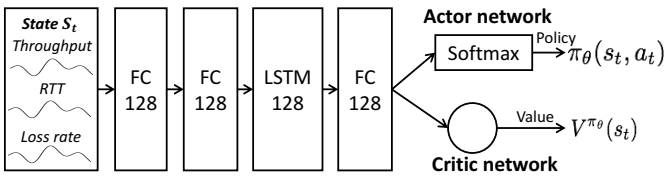


Fig. 6: The neural network structure used in *TCP-RL*. "FC 128" means a fully connected layer with 128 units.

Finally, as discussed in [16, 29], adding the entropy $H(\pi_\theta(\cdot|s_t))$ of the policy π to the objective function can improve the exploration, helping us discover better policies. Then, the gradient of the full objective function with respect to the policy parameters can be written as $\nabla_\theta \log \pi_\theta(s_t, a_t) A^{\pi_\theta}(s_t, a_t) + \beta \nabla_\theta H(\pi_\theta(\cdot|s_t))$, where β is a hyperparameter that controls the strength of the entropy term.

D. Online Running of A3C

For the online *CC* configuration, the agent randomly uses a *CC* at the beginning of the flow. After a few seconds (e.g., 5 seconds), the agent will receive the states from the network environment, which are used as the input of the neural network (illustrated in Fig. 5). Then, the neural network outputs the *CC* that the flow should use in the next step. This way, the agent continuously updates its decision to find the appropriate *CC* for the flow transmission. From receiving the states from the network environment to updating the *CC*, the time consumption is about 700 ms in our experiments on a server with 21 Intel(R) Xeon(R) 2.40 GHz CPUs, 20 GB RAM. It is quite small compared with the time of the *RL* decision-making cycle (e.g., 5s).

Furthermore, as the network environment changes frequently and complicatedly, it might be difficult for the *RL* model to notice the environment changes through only observing the current states. Therefore, we train another neural network offline to detect changes in the network conditions. We manually change the network conditions over time and take the states and action of the last step and the current states as the input of a neural network. The neural network has two fully connected layers and output probabilities of whether the network condition is changed. After offline training, the network can judge whether the network condition is changed with the inputs (last states, last action, current states) during online running. In this way, the agent randomly selects a new *CC* when the neural network detects a network change (just fit the way we do in training, as discussed in VIII-A), which can flush out the states under the old network condition and make our model adapt to the network changes over time.

VI. Design and Implementation

In this section, we present the system design and implementation of *TCP-RL*, as shown in Fig. 7. It has the following

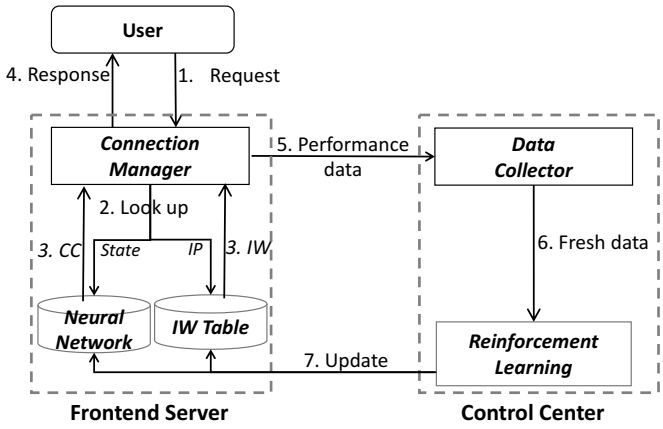


Fig. 7: System design

three key components: 1) *Connection Manager* is a module implemented in the web proxy (e.g., *Nginx* [32]), which is deployed at frontend servers. Its basic functions are to configure an appropriate *IW* or *CC* for each TCP flow and to log the performance data. It has two configurations, called *IW Table* and *Neural Network*, which store each user group's *IWs* and the policy of *CC* configuration. 2) *Data Collector* collects and stores all the performance data of the frontend servers. It provides the fresh data for *Reinforcement Learning* component. 3) *Reinforcement Learning* runs *user grouping* and online *RL* algorithms in §IV based on the fresh data, and it trains the neural network for §V. It updates the *IW Table* and *Neural Network* for *Connection Manager* periodically. It is the controller of the *TCP-RL* system.

A. Connection Manager

For the *IW* configuration for short flow dominated services, when the frontend server establishes a TCP connection with a user, the *Connection Manager* queries the *IW Table* with the user's IP, and the result is the *IW* for this flow. Then it modifies the *IW* for this flow immediately. All the procedures are quickly finished before the frontend server sends the TCP data to the user. When the TCP session is closed, the *Connection Manager* logs the TCP performance data of this session.

For the *CC* configuration for long flow dominated services, the difference is that the *Connection Manager* queries the *Neural Network* with the network state (defined in §V) obtained recently, and the result is the *CC* for this flow; then, it modifies the *CC* for this flow. The *Connection Manager* repeats these procedures at a timescale of seconds until the flow transmission is finished.

To realize the functions of the *Connection Manager*, we implemented a new module in a web proxy (e.g., *Nginx*) and modified the Linux kernel. To be a robust and easily controllable system, most of the jobs are done in the application level in the web proxy, such as obtaining a user's IP after a TCP three-way handshake and looking up *IW* and *CC* from the *IW Table*, *Neural Network*. The modified Linux kernel's job is just exporting two new APIs: configuring the *IW* and *CC* for the TCP flow, and obtaining the TCP performance data. The web proxy cooperates with the modified kernel by calling these two APIs. The first API is *SetParm(fd, iw, cc)*, which is implemented in the *setsockopt* function in Linux, *fd* is the file descriptor of the TCP socket, *iw* is the value of *IW*, and *cc* is the *CC* scheme. When the first API is called, the socket's *IW* and *CC* will be changed. The second API is *GetData(fd)*, which is implemented in the *getsockopt* function in Linux. When it is called, it returns the performance data of the TCP socket. For the web proxy, the *IW Table* and *Neural Network* are the configuration files, providing per-group *IW* and per-flow *CC* policy. As the *IW* or *CC* for each user group or flow is based on *RL*'s decision and can change over time, the web proxy also reloads the *IW Table* and *Neural Network* on demand or periodically.

TABLE III: The performance data in our system

TCP Metrics	**Description**
Size	Data size of the response
TCP Response Time	Data transmission time
MSS	Maximum segment size
Throughput	$\frac{Size}{TcpResponseTime}$
RTT	Smoothed round-trip time ($srtt$) at the end of the transmission
Client Initial Rwnd	Initial receive window of the user
IW	Initial congestion window of the TCP session
CC	*CC* scheme of the TCP session
Loss Rate	Loss rate of the flow
Network Features	**Description**
IP	The user's IP
Province	The user's province
ISP	The user's ISP
Subnet	The subnet that user belongs to

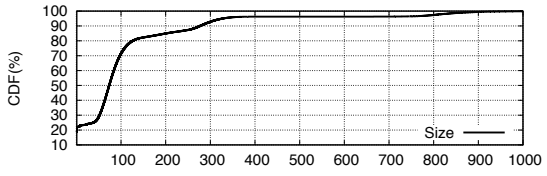
B. Data Collector

All the return data of *GetData(fd)* are shown in Table III. Each frontend server outputs the data in a log file and also uses *HTTP POST* to send the data to a centralized data storage platform in *Data Collector*. *Reinforcement Learning* takes these performance data as the basic input. All the data are collected in real time, and *Data Collector* aggregates and monitors the network performance of each user group, including *TCP Response Time*, *RTT Goodput*, and *Retrans*. Note that these TCP performance data can be used beyond just *TCP-RL*, e.g., for TCP performance troubleshooting.

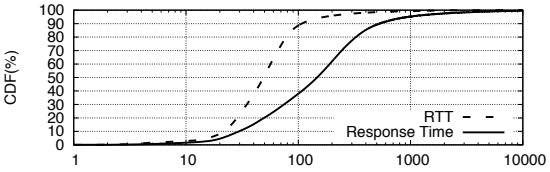
The *TCP Response Time* is the key metric for evaluating the performance for short flow. However, it cannot be obtained directly. Our system aims to be easily deployable with only server-side modification. Here, we use a carefully designed method to record the latency with only server-side change. The key is to collect the timestamp of T_{start} and T_{end} [1], where T_{start} is the time when data sending begins, and T_{end} is the time that the server receives the last ACK from the user. When the web proxy begins to send data to the user or terminate the connection, it calls *GetData(fd)*, which labels the T_{start} of this response, and it also records the T_{end} of the previous response. When it is called, the *TCP Response Time* of the previous response can be computed.

C. Reinforcement Learning

For the *IW* configuration for short flows, after all the performance data are collected, *TCP-RL* runs user grouping and *RL* algorithm in §IV. The user grouping algorithm runs at



(a) Distribution of response size (KB).



(b) Distribution of RTT and response time (ms). The x-axis is the logscale.

Fig. 8: Characteristics of the mobile search service in *Baidu*. The response time is the *TCP response time* [1].

a long timescale, such as days or weeks. The *RL* algorithm runs at a timescale of minutes to continuously learn the suitable *IW* or *CC* scheme for each user group. The result is the *IW Table*, which contains the user groups' TCP parameters at the next learning iteration. This module controls all the frontend servers' behavior by updating their *IW Tables*. It is implemented with Golang [33] and Python in the *Control Center*. For the *CC* configuration for long flows, it runs the A3C [29] offline training in §V and updates the online *Neural Network* periodically. Our A3C is implemented with Tensorflow [34].

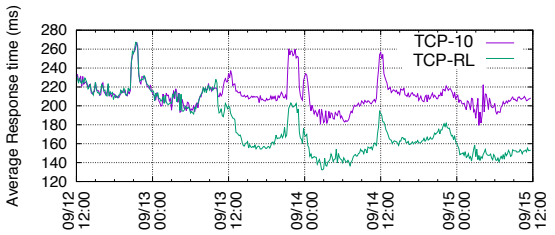
VII. Evaluation for *IW* Configuration

In this section, we use large-scale online experiments and testbed experiments to systematically evaluate the performance of *TCP-RL*'s dynamic *IW* configuration for short flow transmission. The online experiment results show that *TCP-RL* can continuously bring about a 23% improvement in the average response time. For some specific user groups, the improvement can be up to 30%. Then, we run a trace-driven evaluation with ground truth in a testbed. The testbed experiments validate our key ideas:

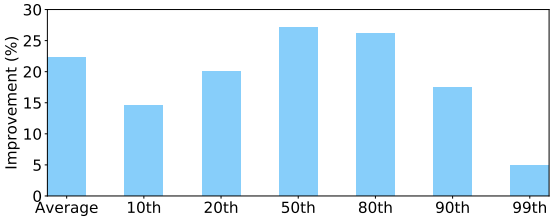
1) The two keys techniques, which are *user grouping* and *RL*, can help improve performance.
2) Directly using an aggressive *IW* causes network congestion, which even hurts the performance of own flows.

A. Online Experiment

In this section, we mainly present the performance in one production data center of *mobile search* service in *Baidu* company; this service was chosen in our experiment because it is among *Baidu*'s most important services. It is a typical service dominated by short flows. *TCP-RL* has been deployed in this service for more than a year. Our real-world A/B testing results show that *TCP-RL* can continuously bring about a 23% improvement in the average TCP response time. For some specific user groups, the improvement can be up to 30%.



(a) The average response time of *TCP-RL* and *TCP-10*.



(b) Each percentile's improvement of *TCP-RL* compared with *TCP-10*.

Fig. 9: TCP response time of *TCP-RL*.

1) Experiment Setup: Fig. 8 shows the statistics of mobile search service, which confirms that the flow sizes are almost all small in this service, but the TCP response time (with *IW* =10) is far from the ideal (one RTT transmission according to [6] for short flows). Furthermore, as shown previously in Table I, more than 80% of the TCP flows are still in the slow start phase when the sessions end without utilizing the available bandwidth.

In the studied data center located in Beijing, the HTTP sessions are uniformly load balanced to frontend servers, which have the same functions and configurations. They all use 21 Intel(R) Xeon(R) 2.40 GHz CPUs, 62 GB RAM, and 10 Gbps NIC. The Linux kernel version is 2.6.32, and the congestion control algorithm is Cubic [12]. To perform an A/B test experiment, we select 4 frontend servers in the data center and divide them into two groups as follows:

- *TCP-10*: Current standard method with a static *IW* = 10.
- *TCP-RL*: Our method with group-based *RL*. The learning iteration interval is 10 min, which means *TCP-RL* will recalculate and update the *IWs* for all user groups in every 10 min. The user grouping method takes *Subnet, ISP,* and *Province* as users' network features. The parameters of user grouping are $S_{min} = 100$, $\theta = 0.1$ and the size of time bin = 10 min. The *RL*'s $\alpha = 0.8, \triangle = 5$, and $\xi = 0.1$. The initial arm list is *IW* = [5, 10, 15, 20]

2) Overall Performance: Fig. 9a shows an example of the average TCP response time in two groups of frontend servers. Before 2017.09.13 10:00:00, both groups of servers use *IW* = 10 [9], and their performance proved to be the same. After that, *TCP-RL* is started in one group. The TCP response time of *TCP-RL* dramatically decreases and continuously outperforms the other group, *TCP-10*, with about 23% improvement. Furthermore, Fig. 9b shows that *TCP-RL* can make improvements in each percentile of the response time. The 50th and 80th percentiles have been improved by about 25%. We observe that the 99th percentile has the smallest improvement, and the main reason is that *TCP-RL* does not explicitly help with the loss, which is one of the factors causing the 99th percentile's

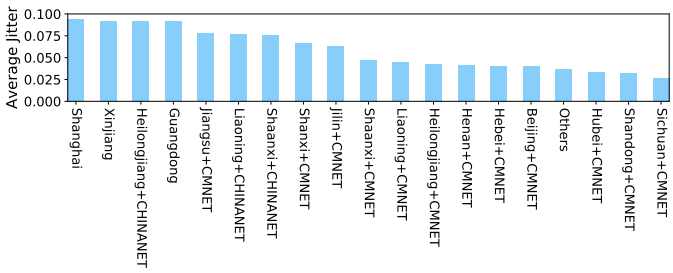


Fig. 10: The average network jitter of each user group.

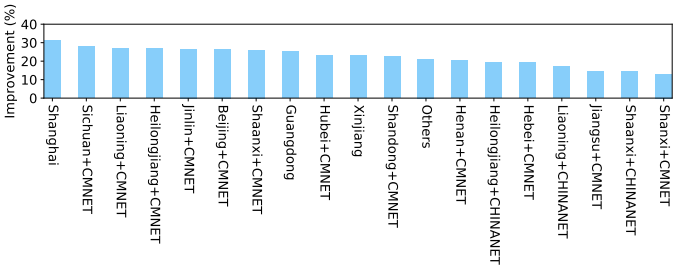


Fig. 11: The average TCP response time's improvement of each user group.

long tail response time [6]. Compared with *TCP-10*, *TCP-RL* may appear to be more aggressive in *IW*, but the results show that even in the 99th percentile, *TCP-RL* also has about 5% improvement over TCP-10. From this, we can see that *TCP-RL* is also quite cautious when increasing *IW*.

3) User Group's Performance: *TCP-RL* uses the *user grouping* technique to treat different user groups individually. In this section, we mainly introduce the performance of each user group. Af first, there are about 1501665 IPs, after using the user grouping method. The output is 19 user groups (3 *Provinces*, 15 *Province+ISP*, 1 *Others*) that can use *RL* to improve their performance. Fig. 10 shows the jitter J of these 19 groups. For the user groups *Shanghai, Guangdong,* and Xinjiang, their jitters are high (closer to the threshold 0.1); flows cannot be grouped into more fine-grained user groups because the more fine-grained user groups cannot satisfy the *RL*'s requirement. i.e., there are not enough samples (either the jitter is larger than 0.1 or the number of samples is smaller than S_{min}). This is because the requests of these users should be routed to other data centers, but they are instead routed, by accident, to the studied data center in Beijing. Fig. 11 shows that all the use groups' response times have been improved (by about 15%-31%).

B. Testbed Evaluation

In this section, we use a trace-driven method to systematically evaluate *TCP-RL*. We built a testbed that supports replaying the online data traces and running different optimization techniques. The data traces consist of user groups' network conditions (e.g. *Bandwidth, RTT*) and application information (e.g. size) in each time bin, which is collected from the online production data center in *Baidu*.

1) Testbed Setup: The testbed consists of 10 physical machines, which are connected by one switch. Its network environment is totally private. Every machine has two 1 Gps NICs, 64 GB RAM, and 64 CPUs (2.4 GHz). One machine acts as the server with *TCP-RL* deployed, and each of the other 9 machines acts as one user group. All the machines use TCP *cubic* with default configuration.

As the HTTP traffic typically has a daily pattern, we selected one day of data traces for 9 user groups from the online experiments in §VII-A. The HTTP requests are replayed in the original timing order by the users and served by the server with the original response sizes. To simplify the experiments, we assume that the network conditions of each user group change at the timescale of one hour. Then, for each hour of each user group, we estimate its bandwidth (RTT) by using the *90th-percentile throughput* (*average RTT*) from the data traces of this user group and hour. We then simulate the network conditions by using the Linux *TC tool* [35] (with *HTB* and *netem* queue) to shape the traffics.

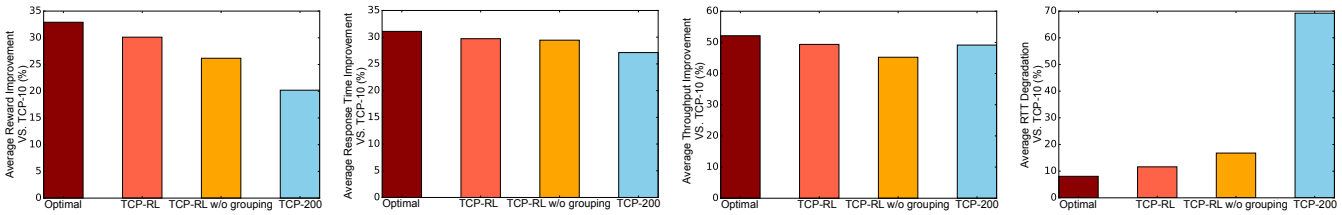
To systematically evaluate the performance of *TCP-RL*, we compare the performance of the following techniques:

1) *TCP-10*: It is the baseline [9], which uses one static *IW* = 10 for all the flows. The data size of *IW* = 10 is about 14 KB ($MSS = 1448$).
2) *TCP-200*: It uses an aggressive *IW* = 200 for all the flows[2]. The data size of *IW* = 200 is about 280 KB.
3) *TCP-RL*: The learning iteration interval is 10 min. The *RL*'s $\alpha = 0.8, \triangle = 1$, and $\xi = 0.1$. The initial arm list is *IW* = [5, 10, 15, 20].
4) *TCP-RL without grouping*: Compared with *TCP-RL*, it only uses RL to learn the *IW* for all flows.
5) *Optimal*: It is the best possible performance in the testbed experiment setting, obtained by exhaustively searching the *IW* space for the best *IW* for each network condition of each user group.

2) Overall Performance: Fig. 12 shows each technique's improvement or degradation over *TCP-10*, and Fig. 13 shows each technique's distribution of the network metrics (response time, reward, throughput, and RTT). We define technique t's improvement in reward, throughput, and response time as $(reward_t - reward_{tcp-10})/reward_{tcp-10}$, $(throughput_t - throughput_{tcp-10})/throughput_{tcp-10}$, $(responsetime_{tcp-10} - responsetime_t)/responsetime_{tcp-10}$. The degradation of RTT is defined as $(RTT_t - RTT_{tcp-10})/RTT_{tcp-10}$. We can see that *TCP-RL* is closest to the optimal, and it significantly outperforms *TCP-10*. Its improvement over *TCP-10* is 30% for the average reward, 29% for the average response time, and 49% for the average throughput.

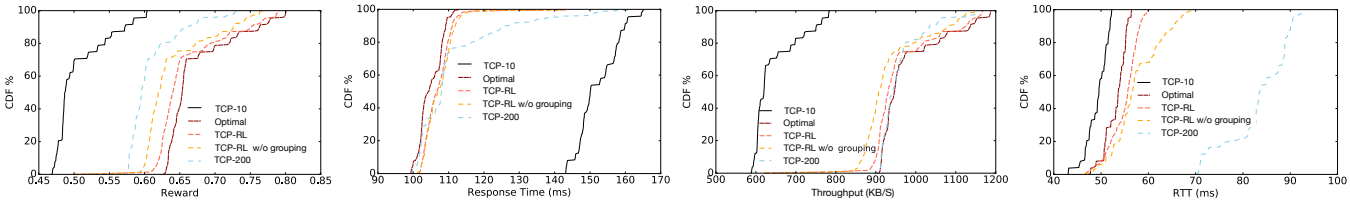
3) Contributions of User Grouping *and RL:* In this experiment, we use reward as the metric because it captures the effects of both throughput and RTT. First, to evaluate the performance of *RL*, we compare the reward of *TCP-RL without grouping* with *TCP-10*. Fig. 12a shows that *TCP-RL without grouping* has a 25% improvement. Fig. 13a also shows that *TCP-RL without grouping* significantly outperforms *TCP-10*. Second, to evaluate the performance of *user grouping*, we

[2]The actual sending window size is *min(Rwnd, Cwnd)*. *Rwnd* is the client receive window. In order to remove the influence of the client's *Rwnd*, the initial *Rwnd* is set to be larger than 200.



(a) Reward (b) TCP response time (c) Throughput (d) RTT

Fig. 12: Compared with the baseline *TCP-10*, the figures show each technique's improvement in average reward, TCP response time, throughput, and degradation in the average RTT.



(a) Reward (b) TCP response time (c) Throughput (d) RTT

Fig. 13: Distribution of each technique's reward, TCP response time, throughput and RTT.

compare the reward of *TCP-RL* with *TCP-RL without grouping*. Fig. 13a shows *TCP-RL* can bring more improvement (29%) than *TCP-RL without grouping* (25%). Besides, *TCP-RL* also outperforms *TCP-RL without grouping* in throughput (Fig. 12c) and RTT (Fig. 12d). The reason is that using *RL* for all the flows (without grouping) with variable network conditions is suboptimal. From this, we can see both *user grouping* and *RL* can help improve the TCP response time.

4) Aggressive IW's Effect: According to the *common perception of IW* (see §II-B), neither a too small *IW* nor a too large *IW* is suitable. The results of *TCP-200* confirm the second half of this common perception. The flows using an aggressive *IW* = 200 can cause network congestion and even hurt their own performance. Fig. 13d shows that *TCP-200* has the highest average RTT, which is a good indicator for network congestion. On the other hand, *TCP-RL*'s RTT is closest to that of *Optimal*, and is better than that of *TCP-RL without grouping*. From this, we can see *TCP-RL* is quite cautious in avoiding congestions. For the average response time and throughput, *TCP-200* outperforms *TCP-10*, but it cannot beat *TCP-RL*. The reason is that directly using a large *IW* may cause packet loss and increase the response time. Fig. 13b shows *TCP-200* has a much longer tail than *TCP-RL* because it suffers from packet loss, which causes a costly TCP timeout.

VIII. Evaluation for *CC* Configuration

This section evaluates the performance of *TCP-RL*'s dynamic *CC* configuration for long flow transmission. The highlights of the results are as follows:

1) The performance of different *CC* schemes varies significantly across various network conditions, and no single *CC* scheme can outperform all others in all network conditions. This confirms the observations in Pantheon [8].
2) *TCP-RL*'s trained neural network model can quickly select a well-performing *CC* scheme for a specific network condition within only one or two trials. For long flows, compared with the performance of 14 *CC* schemes, *TCP-RL*'s performance ranks top 5 for about 85% of the 288 given static network conditions, whereas for about 90% of conditions, its performance drops by less than 12% compared with the that of best-performing *CC* schemes for the same network condition.
3) During the lifetime of a long flow, *TCP-RL* can detect network changes and converge to the corresponding *CC* scheme after only 2 tries.

Label	Scheme
copa	Copa [14]
taova	TCP Remy [22]
pcc	PCC [36]
pcc_expr	PCC experimental [36]
quic	QUIC Cubic [37]
sprout	Sprout [38]
bbr	TCP BBR [13]
cubic	TCP Cubic [12]
vegas	TCP Vegas [26]
vivace	PCC Vivace [7]
fillp	Fillp [8]
fillp-sheep	Fillp-Sheep [8] (the second variant)
indigo	Indigo [8] (LSTM neural network)
verus	Verus [39]

TABLE IV: Congestion control schemes in *TCP-RL*.

Overall, *TCP-RL*'s flow-level automatic and dynamic *CC* configuration is a significant improvement over service-level static and manual *CC* configuration, given the temporal and spatial (across different users) dynamics of the network conditions.

A. Experiment Setup

We use *Pantheon* [8] to test the performance of different *CC* schemes on different network conditions. *Pantheon* has a collection of *CC* schemes, shown in Table IV, and a *mahimahi* tool [40], which can emulate diverse network conditions on one server. *TCP-RL* directly utilizes *Pantheon*'s functions to dynamically set the *CC* scheme for one flow, and emulates the network conditions.

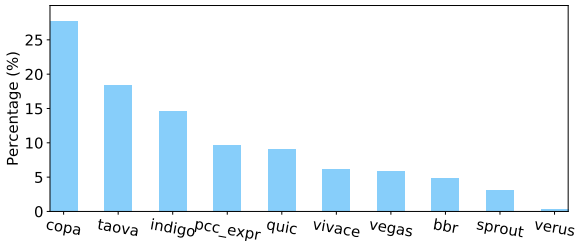


Fig. 14: Percentage of the 288 network conditions in which each *CC* scheme has the best performance.

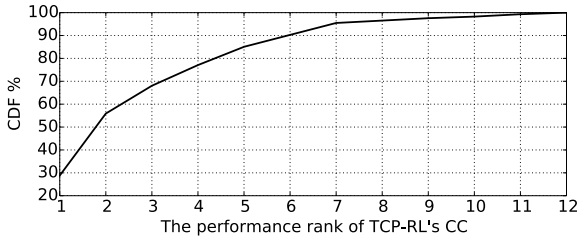


Fig. 15: Performance rank of the *CC* learnt by *TCP-RL*.

Model Training: We use 288 synthetic network conditions to train *TCP-RL*'s neural network. The synthetic network conditions consist of all the combinations of (1, 5, 10, 20, 50, 70, 100, 150 and 200 Mbps) bandwidths, (10, 20, 40, 60, 80, 100, 150 and 200ms) RTTs and (0, 1%, 2% and 10%) loss rate. A3C's neural network is implemented using TensorFlow [34]. We spent about 30 hours to train the model on a Linux server with 21 Intel(R) Xeon(R) 2.40 GHz CPUs, 20 GB RAM.

The neural network in our algorithm consists of two fully connected layers, followed by one LSTM layer and one fully connected layer, with 128 units in each layer, as shown in Fig. 6. We apply softmax function on the output of the last hidden layer to obtain the policy π_θ (a set of action probabilities), and a linear layer to obtain the value function $V(s_t;\theta_v)$. The learning rate of the actor-critic network is set to 10^{-4}, and the reward discounting factor γ is set to 0.99. In addition, the entropy term factor β is controlled to decay from 0.5 to 0.01 over 10^5 iterations, which can encourage exploration at the start of the training, and then to decrease over time in order to emphasize the improving reward. Moreover, to speed up the model training and well explore the action space under different network conditions, we use 16 parallel worker agents to experience different network conditions asynchronously and collect the (state, action, reward) data (i.e., TCP measurement data, selected *CC*, reward) continuously. In different network conditions, we apply a random *CC* in each environment for starting, and the network condition in each environment lasts for a while. A central agent receives these data and updates the actor-critic network to train the model, and then it pushes the parameters to the network in each worker agent.

B. Best CC Schemes on Different Network Conditions

We run each *CC* scheme on each of the 288 synthetic network conditions to find the best *CC* (with the largest reward = log(throughput/RTT)) for each condition. Fig. 14 shows the percentage of network conditions in which each *CC* has the best performance. The results confirm the observation in

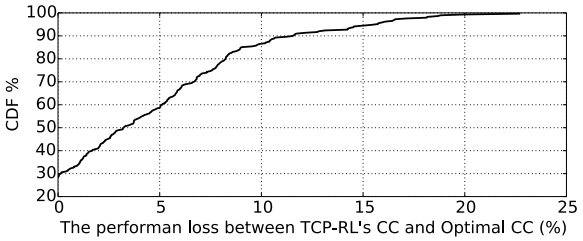


Fig. 16: CDF of the performance loss between *TCP-RL*'s *CC* and Optimal *CC*. The performance loss = $\frac{Reward_{optimal}-Reward_{TCP\text{-}RL}}{Reward_{optimal}}$

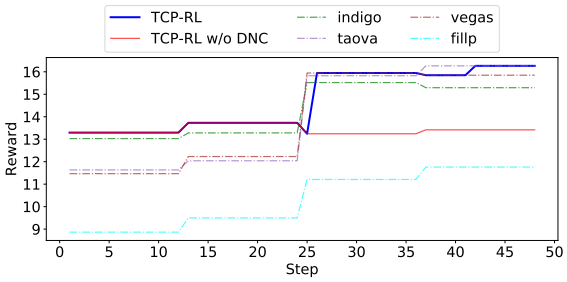


Fig. 17: Reward of each *CC* and *TCP-RL* in a dynamic network condition environment. One step is 5 s. Here we only show the curves of Top3 *CC*, namely, *TCP-RL*, *TCP-RL w/o DNC*, and the last *CC* in Table V for simplicity.

Pantheon [8] that no single *CC* scheme can outperform all the others in every network condition. Copa performs the best overall, but it can only do so in about 28% of the network conditions. For the *CC*, such as *Cubic* and *Fillp*, the percentage is 0, which means they perform the best in none of the 288 conditions. These observations validate the need for a dynamic *CC* configuration instead of manual and static configuration.

C. Performance of TCP-RL's Neural Network

We test the *TCP-RL*'s deep *RL* model in the 288 synthetic network conditions. Fig. 15 shows the CDF of the reward rank of the *CC* scheme that the *TCP-RL* model learned. The results show that *TCP-RL* can learn the top-5 *CC* for about 85% of the network conditions. Fig. 16 shows the performance loss distribution between *TCP-RL*'s *CC* and the optimal *CC*. From this, we can see that for about 90% of conditions, its performance is less than 12% worse than that of the optimal *CC* schemes for the same network conditions. *TCP-RL*'s *CC* performance is quite close to that of the optimal *CC* for the same network conditions. We also found that for 96% of the network conditions, the model can converge to one *CC* within 3 trials.

D. TCP-RL's Performance for Changing Network Conditions

In reality, the network could change over time. Fig. 17 shows the performance of *TCP-RL* for one long flow whose network conditions change during the flow's lifetime. We compared the performance of a few representative *CCs* in Table IV, *TCP-RL* and *TCP-RL w/o DNC*; *TCP-RL w/o DNC* is the version of *TCP-RL* without the model that detects changes of network conditions (§V-D). The network condition

Scheme	Mean reward ($log(Throughput/RTT)$)	Mean throughput (Mbits/s)	Mean 95th-%ile RTT (ms)	Mean loss rate
TCP-RL	14.71	58.31	12.16	0.0011
Indigo	14.28	40.26	28.58	0.0015
TCP Remy	13.94	74.64	115.7	0.0087
TCP Vegas	13.87	71.97	110.7	0.0231
Sprout	13.51	11.47	16.08	0.0034
Copa	13.42	8.008	11.5	0.0007
TCP-RL w/o DNC	13.42	8.008	11.5	0.0007
TCP BBR	13.22	94.60	124.8	0.0144
PCC experimental	13.13	41.04	67.75	0.0089
QUIC Cubic	12.78	4.899	12	0.0012
TCP Cubic	12.76	91.51	225.5	0.0522
PCC	12.55	24.98	106.4	0.0018
PCC Vivace	12.49	8.19	46.5	0.013
Fillp-Sheep	12.38	92.71	279	0.0268
Verus	12.32	39.84	217.1	0.0028
Fillp	10.33	89.55	2212	0.4764

TABLE V: *TCP-RL* versus existing congestion control algorithms.

was set to be changed every 12 steps (about 1 min, as there are 5 s in one step). The trajectory of network condition change is $(20\ Mbps, 20\ ms, 2\%) \rightarrow (50\ Mbps, 40\ ms, 1\%) \rightarrow (150\ Mbps, 20\ ms, 10\%) \rightarrow (200\ Mbps, 20\ ms, 1\%)$. The results show that *TCP-RL* can learn the appropriate *CC* with only 2 or 3 trials, and this learned *CC* is close to the optimal *CC*'s performance if not the same. Table V shows this long flow's mean performance (i.e., reward, throughput, RTT, loss rate) for *TCP-RL*, *TCP-RL w/o DNC*, and the 14 *CCs*. The results show that *TCP-RL* outperforms all the other *CC* schemes in this long flow. This is because without the detection of network changes, *TCP-RL w/o DNC* performance is worse than that of *TCP-RL*, but it is still better than that of many other *CC* schemes because of the good performance of A3C model. Although some *CC* schemes (i.e. Taova, Fillp) have a larger throughput than *TCP-RL*, they have a much higher RTT (95^{th} percentile). The reward ($log(Throughput/RTT)$) ensures *TCP-RL* achieve a high throughput while still keeping a low RTT.

In theory, there could be room to define an even better reward, *TCP-RL*'s learning method is general and can easily extend to other versions of reward. We leave this to our future work.

IX. Related work

Congestion Control: An efficient *CC* algorithm is critical for data transmission. Since the development of Jacobson's TCP Tahoe algorithm [11] in 1988, TCP *CC* over the Internet has been a hot research topic for decades. Many *CC* algorithms [8, 11, 13, 22, 26, 41–43] have been proposed.

Some *CCs* are rule based, including Tahoe, Reno [11], NewReno [41], Vegas [26], FastTCP [42], Compound TCP [43], Bic, Cubic [12], and BBR [13]. They all depend on some specific rules (e.g. slow start, fast retransmission) to determine their window size and when data are transmitted. Tahoe, Reno [11], New Reno [41], and Cubic [12] are loss-based algorithms. The sender increases or reduces its congestion window based on whether loss has happened. Vegas [26] and FastTCP [42] are delay-based algorithms. They treat RTT as the congestion signal. Compound TCP [43] combines the ideas of loss-based and delay-based algorithms, treating loss and RTT as the congestion signal. BBR [13] aims to find the balance between maximizing the throughput and minimizing the RTT. All the above *CC* schemes use a *heuristic-based* trial-and-error approach to probe for the best congestion window size *within* a TCP session *only*. Some *CCs* are based on machine learning. Remy [22] uses an offline-trained machine learning model to dynamically assign the congestion window sizes based on the latest network conditions measured *within* the TCP session *only*. Indigo [8] uses a deep learning method to train a network congestion algorithm for detailed network conditions. Some other works [6, 44–46] modify *CC* schemes and propose better loss recovery mechanisms to deal with packet losses. However, although many variants of *CC* schemes have been proposed, it is shown in Pantheon [8] that until now, there is no *CC* scheme that can outperform all the others. In this paper, instead of building a new congestion *CC* scheme, we argue that using deep *RL* could dynamically configure the right *CC* schemes for different network conditions on a per-flow level.

***IW* Improvement**: [9] proposed to simply increase the standard *IW* to 10 for all flows, and we have shown that *TCP-RL* outperforms this approach by 23%. Halfback [20] always starts with a large *IW* and then applies pacing and redundancy technologies to deal with the loss (caused by the aggressive startup) *within* the flow without using history session information; by contrast we have shown that *TCP-RL* outperforms the approach that blindly sets a large *IW* (e.g., 200) which can cause significant congestion. In an early work, for repeated flows between the same client and server, [21] uses the last session's TCP parameters for a fast startup, but it needs router support, and there might not be many repeated flows between the same client and server. These approaches use information from within a TCP session only, and they do not utilize the valuable information from previous sessions. In comparison, *TCP-RL* utilizes much richer history information from the user group, is much more applicable, and only needs to modify the TCP servers.

Cases of *RL* in Internet Video QoE Optimization: Pytheas [15] applies *RL* to video QoE optimization, by dynamically deciding a session's serving frontend server. It differs from *TCP-RL* because of the following domain differences: First,

IW uses the sliding-decision-space approach to deal with the large *IW* decision space, which is much larger than the frontend server selection in Pytheas. [15] only shows the testbed evaluation while *TCP-RL* has been deployed in real data centers for more than a year. Third, the user grouping methods are different in *TCP-RL* (for general TCP performance) and Pytheas (tailored for video QoE). Pensieve [16] improves video QoE by applying Deep *RL* to generate the ABR algorithms for each client session given the measurement data from *within* the session. In order to improve TCP transmission performance for long flows, *TCP-RL* uses deep *RL* to generate a policy for dynamically configuring *CC* schemes.

X. Conclusion

In this paper, to improve the performance of TCP transmission, we propose a system called *TCP-RL* to dynamically configure a suitable *IW* for short flows through group-based *RL*, and to dynamically configure a suitable *CC* scheme for each long flow through deep *RL*. *TCP-RL* is incrementally deployable at the server side without any client or router support. *TCP-RL*'s dynamic *IW* configuration does not change and is compatible with existing TCP *CC* algorithms. *TCP-RL*'s dynamic *CC* configuration can work with any *CC* schemes. *TCP-RL* has been deployed in one of the top global search engines for more than a year. Our online and testbed experiments show that for services dominated by short flows, compared with the common initial window size of 10, *TCP-RL* can reduce the TCP response time by 23% to 29%. For long flows, compared with the performance of 14 *CC* schemes, *TCP-RL*'s performance ranks top 5 for about 85% of the 288 given static network conditions, whereas for about 90% of the conditions, its performance drops by less than 12% compared with that of the best-performing *CC* schemes for the same network condition. Even when network conditions change dynamically during a flow's lifetime, *TCP-RL* can converge to the appropriate *CC* according to the latest network conditions after only 2-3 trials. This flow-level automatic and dynamic *CC* configuration is a significant improvement over service-level static and manual *CC* configuration, given the temporal and spatial (across different users) dynamics of network conditions.

We believe that *TCP-RL* is an important step toward applying advanced machine learning techniques to solve hard and open network research problems. In the future, we could use *RL* to control other TCP parameters (such as RTO), or use machine learning techniques to improve the network protocol in the data center, satellite, and other network scenarios.

Acknowledgment

We thank the anonymous reviewers and our shepherd Rolf Stadler for their valuable feedbacks. We acknowledge Qinghua Ding and Jiachang Liu for their survey of *RL* and coding in Nginx. This work has been supported by the National Natural Science Foundation of China (NSFC) under grants 61472214 and 61472210, Beijing National Research Center for Information Science and Technology (BNRist) key projects, the Global Talent Recruitment (Youth) Program, and Okawa Foundation Research Grant.

References

[1] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing Web Latency through Dynamically Setting TCP Initial Window with Reinforcement Learning," in *IEEE IWQoS*, 2018, pp. 1–10.

[2] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang, "A Provider-Side View of Web Search Response Time," in *ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 243–254.

[3] D. Liu, Y. Zhao, K. Sui, L. Zou, D. Pei, Q. Tao, X. Chen, and D. Tan, "FOCUS: Shedding Light on the High Search Response Time in the Wild," in *IEEE INFOCOM*, 2016, pp. 1–9.

[4] "Latency Is Everywhere And It Costs You Sales," https://goo.gl/bRi5Xs, Accessed: 2018-12-14.

[5] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of Response Latency on User Behavior in Web Search," in *ACM SIGIR*, 2014, pp. 103–112.

[6] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing Web Latency: The Virtue of Gentle Aggression," in *ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 159–170.

[7] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, "PCC Vivace: Online-Learning Congestion Control," in *USENIX NSDI*, 2018, pp. 343–356.

[8] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein, "Pantheon: the training ground for Internet congestion-control research," in *USENIX ATC*, 2018, pp. 731–743.

[9] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An Argument for Increasing TCP's Initial Congestion Window," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 3, pp. 26–33, 2010.

[10] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe, "Open Research Issues in Internet Congestion Control," *RFC 6077*, 2011.

[11] V. Jacobson, "Congestion Avoidance and Control," in *ACM SIGCOMM*, vol. 18, no. 4, 1988, pp. 314–329.

[12] S. Ha, I. Rhee, and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," *ACM SIGOPS*, vol. 42, no. 5, pp. 64–74, 2008.

[13] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, p. 50, 2016.

[14] V. Arun and H. Balakrishnan, "Copa: Practical Delay-Based Congestion Control for the Internet," in *USENIX NSDI*, 2018, pp. 329–342.

[15] J. Jiang, S. Sun, V. Sekar, and H. Zhang, "Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation," in *USENIX NSDI*, 2017, pp. 393–406.

[16] H. Mao, R. Netravali, and M. Alizadeh, "Neural Adaptive Video Streaming With Pensieve," in *ACM SIGCOMM*, 2017, pp. 197–210.

[17] A. Slivkins, "Contextual Bandits With Similarity Information," in *The Journal of Machine Learning Research*, vol. 15, 2014, pp. 2533–2568.

[18] A. Mahajan and D. Teneketzis, "Multi-Armed Bandit Problems," *Foundations and Applications of Sensor Management*, pp. 121–151, 2008.

[19] M. Tokic and G. Palm, "Value-Difference Based Exploration: Adaptive Control Between Epsilon-Greedy and Softmax," in *Annual Conference on Artificial Intelligence*. Springer, 2011, pp. 335–346.

[20] Q. Li, M. Dong, and P. B. Godfrey, "Halfback: Running Short Flows Quickly and Safely," in *ACM CoNEXT*, 2015.

[21] R. H. Katz and V. N. Padmanabhan, "TCP Fast Start: A Technique for Speeding up Web Transfers," in *IEEE Globecom*, vol. 34, 1998.

[22] K. Winstein and H. Balakrishnan, "TCP ex Machina: Computer-Generated Congestion Control," in *ACM SIGCOMM*, vol. 43, no. 4, 2013, pp. 123–134.

[23] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[24] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-Time Analysis of the Multiarmed Bandit Problem," *Springer Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[25] A. Garivier and E. Moulines, "On Upper-Confidence Bound Policies for Non-Stationary Bandit Problems," *arXiv preprint arXiv:0805.3415*, 2008.

[26] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance," in *ACM SIGCOMM*, vol. 24, no. 4, 1994.

[27] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats *et al.*, "TIMELY: RTT-based Congestion Control for the Datacenter," in *ACM SIGCOMM*, vol. 45, no. 4, 2015, pp. 537–550.

[28] "DB-IP," https://db-ip.com/, 2018, Accessed: 2018-12-14.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[30] L. Kleinrock, "Internet congestion control using the power metric: Keep the pipe just full, but no fuller," *Elsevier Ad Hoc Networks*, vol. 80, pp. 142–157, 2018.

[31] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in neural information processing systems*, 2000, pp. 1057–1063.

[32] N. Inc., "A Free, Open-Source, High-Performance HTTP Server," https://www.nginx.com/, Accessed: 2018-12-14.

[33] "The Go Programming Language," https://golang.org, Accessed: 2018-12-14.

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *USENIX OSDI*, vol. 16, 2016, pp. 265–283.

[35] "Linux Traffic Control," http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, Accessed: 2018-12-14.

[36] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, "PCC: Re-architecting Congestion Control for Consistent High Performance." in *USENIX NSDI*, vol. 1, no. 2.3, 2015, p. 2.

[37] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *ACM SIGCOMM*, 2017, pp. 183–196.

[38] K. Winstein, A. Sivaraman, H. Balakrishnan *et al.*, "Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks." in *USENIX NSDI*, vol. 1, no. 1, 2013, pp. 2–3.

[39] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg, "Adaptive congestion control for unpredictable cellular networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 509–522.

[40] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for http." in *USENIX ATC*, 2015, pp. 417–429.

[41] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for TCP," in *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4, 1996, pp. 270–280.

[42] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, 2006.

[43] K. T. J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance networks," in *IEEE INFOCOM*, 2006.

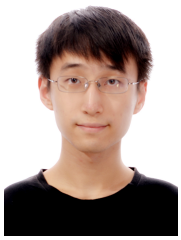
[44] J. Zhou and etc., "Demystifying and Mitigating TCP Stalls at the Server Side," in *ACM CoNEXT*, 2015.

[45] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang *et al.*, "Fast and Cautious: Leveraging Multi-Path Diversity for Transport Loss Recovery in Data Centers," in *USENIX ATC*, 2016.

[46] C. P. Fu and S. C. Liew, "TCP Veno: TCP enhancement for transmission over wireless access networks," *IEEE Journal on selected areas in communications*, vol. 21, no. 2, pp. 216–228, 2003.

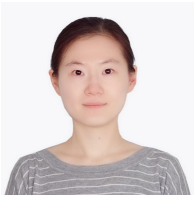
Xiaohui Nie received his B.S. degree from the in computer science and technology department of Jilin University, Changchun, China in 2013. He is currently a Ph.D. candidate in computer science in Tsinghua University, Beijing, China. His current research interests include intelligent TCP, service monitoring and management.

Youjian Zhao received his B.S. degree from Tsinghua University in 1991, his M.S. degree from the Shenyang Institute of Computing Technology, Chinese Academy of Sciences, in 1995, and his Ph.D. degree in computer science from Northeastern University, China, in 1999. He is currently a professor with the CS Department of Tsinghua University. His research mainly focuses on high-speed Internet architecture, switching and routing, and high-speed network equipment.



Zhihan Li received his B.S. degree from Xidian University, Xi'an, China, in 2017. He is currently a Ph.D. candidate in the computer science and technology department in Tsinghua University, Beijing, China. His current research interests include machine learning and its applications in network management.

Guo Chen received his B.S. degree from Wuhan University in 2011 and his Ph.D. degree from Tsinghua University in 2016. He was an associate researcher with Microsoft Research Asia from 2016 to 2018. He is currently an associate rrofessor with Hunan University. His current research interests include networked systems, with a special focus on data center networking.



Kaixin Sui is an associate researcher in Microsoft Research Asia (MSRA). Before joining MSRA, she was a software engineer in Microsoft Azure. She got her PhD from Tsinghua University in 2016. Her present research focuses on capacity management and cloud service intelligence.

Jiyang Zhang received his B.S. degree from Zhengzhou University in 2013. He is currently a senior engineer in Baidu Inc. His research interests include high-performance web proxy, TCP optimization.

Zijie Ye was admitted to Tsinghua University, Beijing, China in 2016. He is currently an undergraduate student in the computer science and technology department

Dan Pei received his B.S. and M.S. degrees from Tsinghua University, Beijing, China in 1997 and 2000, respectively, and his Ph.D. degree from the University of California, Los Angeles, CA, USA, in 2005. He is currently an associate professor with Tsinghua University. His current research interests include AIOps, which is at the intersection of AI, business, and IT operations.