Reducing Web Latency through Dynamically Setting TCP Initial Window with Reinforcement Learning

Xiaohui Nie[†⋆], Youjian Zhao[†⋆], Dan Pei[†⋆], Guo Chen[‡*] , Kaixin Sui[§], Jiyang Zhang[¶]
[†]Tsinghua University [‡]Hunan University [§]Microsoft Research [¶]Baidu
[⋆]Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—Latency, which directly affects the user experience and revenue of web services, is far from ideal in reality, due to the well-known TCP flow startup problem. Specifically, since TCP starts from a conservative and static initial window (*IW*, 2∼4 or 10), most of the web flows are too short to have enough time to find its best congestion window before the session ends. As a result, TCP cannot fully utilize the available bandwidth in the modern Internet. In this paper, we propose to use group-based reinforcement learning (*RL*) to enable a web server, through trial-and-error, to dynamically set a suitable *IW* for a web flow before its transmission starts. Our proposed system, *SmartIW*, collects TCP flow performance metrics (*e.g.*, transmission time, loss rate, RTT) in real-time without any client assistance. Then these metrics are aggregated into groups with similar features (*subnet, ISP, province*, *etc.*) to satisfy *RL*'s requirement. *SmartIW* has been deployed in one of the top global search engines for more than a year. Our online and testbed experiments show that, compared to the common practice of $IW = 10$, *SmartIW* can reduce the average transmission time by 23% to 29%.**

I. Introduction

Latency, which greatly affects the user experience and revenue [1, 2], has become one of the most important performance metrics for modern online web services (*e.g.*, web search, e-commerce website). However, the network transmission time in those web services is still far from ideal, and significantly increases the end-to-end latency [3, 4]. One of the key reasons is that TCP, which most online web services (*e.g.*, Microsoft [5], Google [3], Baidu [4]) are based on, cannot deal with short web flows gracefully.

Specifically, TCP starts with a conservative and static initial congestion window (*IW*, 2, 4, or 10 [6]), then 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 inadequately take multiple RTTs to probe for its best *CWND* although the flows are too short to have enough time for TCP find one starting from the conservative *IW*. As a real-world example, Table I shows that, for the mobile search service in a top global search company *B* where *IW* =10, more than 80% of the TCP flows are still in slow-start phase when the sessions end, not utilizing the available bandwidth.

The above *TCP flow startup problem* is still considered by the research community as an open research problem [7] for general TCP environment. Google proposed to increase the standard *IW* from 2 ∼ 4 to 10 [6]. But is 10 still too small for those high-speed users (*e.g.*, with fiber access) and is 10 too large for low-speed users (*e.g.*, GPRS access in remote areas)? Since the network conditions can be highly variable both spatially and temporally, it is actually infeasible to choose a static *IW* that is best for all flows. In this paper, we argue that *a web-service provider can collect TCP flow data and use an appropriate data-driven learning method to set a suitable IW for a web flow even before its transmission starts*.

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

TABLE I: The distribution of TCP states after finishing transmission. Measured in the mobile search service of *B* during one week in 2017. The average flow size is about 120KB.

Setting TCP *IW* through data-driven learning has been little studied. Since the network condition can be highly variable and the relationship between *IW* and network conditions is complex, we model the *IW* setting as a *reinforcement learning (RL)* problem (real-time exploration and exploitation) instead of a *prediction* problem (offline training and online prediction). Its basic idea is to continuously strike a balance between exploring suboptimal decisions and exploiting currently optimal decisions [8]. Our choice of *RL* is inspired and encouraged by the recent progress in applying *RL* (instead of prediction model) to Internet video QoE optimization through dynamically deciding a video session's serving frontend servers [9] or through tuning ABR algorithms [10].

In this paper, *RL* is used to continuously update the decisions of the suitable *IW* for flows, based on the most recent fresh performance data, to maximize the cumulative reward (network performance). Although applying *RL* to set TCP *IW* is a promising abstraction, there are three major practical challenges in practice:

- Challenge 1: *How to measure TCP performance data on the server side only? RL* method needs fresh TCP performance data to compute reward. However, a traditional web service server cannot directly measure the TCP transmission time without clients' collaboration.
- Challenge 2: *How to apply RL methods on highly variable and non-continuous network conditions of the Internet? RL* continuously updates the decisions based on fresh data and historical experience. Its decision is determined by a static (but unknown) distribution of the context,

* Guo Chen is the corresponding author.

978-1-5386-2542-2/18/$31.00 ©2018 IEEE

and traditional *RL* methods require the continuity of the context that affects the reward of the decision [11, 12]. In our case, the decision of *RL* is the latest suitable *IW* for a flow, and the context of *RL* is a flow's **network conditions (*i.e.*, available end-to-end bandwidth and RTT)**. However, not every user or user group can satisfy *RL*'s context continuity requirement. Even worse, for some fine-grained user groups such as *user IP*, most of them do not even have enough data samples to tell whether their network conditions are continuous or not.

- Challenge 3: *How to search for the optimal IW from a large window space quickly? RL* methods are essentially based on *trial and error*. They require clever exploration mechanisms. Brute-forcedly selecting actions results in poor performance [13], and typically only suits small and limited decision space, which can quickly converge to the best decision after a small number of trials. However, the possible *IW* space for a TCP flow can be so large that the network conditions might have already changed before brute-forcedly searching can find the optimal *IW* for the previous network conditions.

In this paper, we propose a system called *SmartIW* that can set TCP *IW* at server side smartly using group-based reinforcement learning, which addresses all the above challenges. The contributions of this paper are summarized as follows:

- To address Challenge 1, we modify Linux kernel and Nginx software to enable web servers to collect and store billions of TCP flow performance records (*e.g.*, transmission time, loss rate, RTT) in real-time without any client assistance (§V). Such a server-side TCP measurement system can be used beyond *SmartIW*, *e.g.*, for TCP performance troubleshooting.
- To address Challenge 2, our intuition is that flows from users sharing the same **network features (*e.g.*, *subnet, ISP, province*)** typically have similar network performance [9]. Thus, we propose a bottom-up approach to group flows from users with the same network features to find the most fine-grained user groups that both have enough samples and satisfy the *RL* context continuity requirements. Then we apply *RL* methods at the group granularity. (§IV-B)
- To address Challenge 3, we improve *RL* with a fast decision space searching algorithm. Based on the common perception of the relationship between TCP performance and the *IW*, we propose a sliding-decision-space method that can quickly converge to the best *IW* (§IV-A).
- *SmartIW* has been deployed in one of the top global search engines for more than a year. Our online and testbed experiments show that, compared to the common practice of $IW = 10$, *SmartIW* can reduce the average transmission time by 23% to 29%.

II. Background

A. TCP response time in web services

Fig. 1 shows a typical infrastructure of web services [3, 6], and illustrates how users, frontend servers, and backend

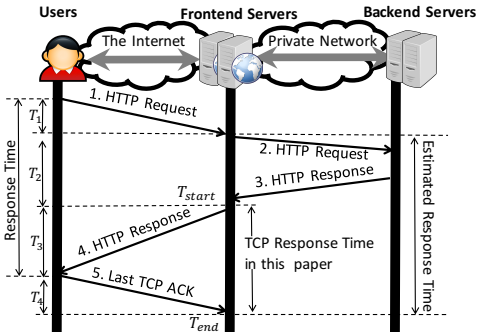


Fig. 1: Typical infrastructure of web services, HTTP interactions, and TCP response time.

servers interact to complete a web request. Frontend servers parse users' requests from the Internet (step 1), and redirect them to the backend servers who handle the concrete tasks (step 2). Then they receive the responses (HTML, images, css, js files) from backend servers (step 3) and send them to users through the Internet (step 4). The Web *response time* (or informally *Web latency*) [3] consists of $(T_1 + T_2 + T_3)$, the transmission time between frontend servers and users through the Internet, and T_2, the time between frontend and backend servers through the dedicated high-speed private network and the processing time at the backend servers. Unfortunately, neither T_1 nor T_3 can be directly measured at server-side without client assistance. However, the frontend server can measure T_{start} and T_{end}, where T_{start} is the moment when the frontend server begins to send data to the user (step 4), and T_{end} is the moment when the frontend server receives the last ACK of the response which indicates the end of this network transmission (step 5). Thus, T_4 is the transmission time of the response's last TCP ACK (a small packet), which can be used to closely approximate T_1 (the time to transmit the HTTP request, also a small packet). Therefore, it is a common practice [3] to use $T_2 + T_3 + T_4$ (estimated response time) to estimate $T_1 + T_2 + T_3$ (response time).

Because T_2 can be considered by the users as an overall processing time of the web service provider, and $T_1 + T_3$ is affected by the TCP flow start problem the most, in this paper we focus on $(T_1 + T_3)$ only, and call it **TCP Response Time** hereinafter. It is calculated as $(T_{end} - T_{start})$.

B. IW greatly affects the network transmission time

The common perception of IW: *IW* determines the initial sending rate at the flow startup phase. Too small an *IW* suffers from more RTTs than necessary to finish the transmission; too large an *IW* results in congestion or even expensive TCP timeout, which results in high TCP response time [6]. In summary, different *IWs* can greatly affect TCP response time.

Fig. 2 shows *IWs*' effects in two example network conditions. In Fig. 2(a) the network condition is relatively good where the link can support *CWND* =3. So if server's *IW* =3, the TCP response time is 1 RTT. But when the *IW* = 1, the response time is 2 RTTs. In Fig. 2(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 greatly reduce the network transmission time.

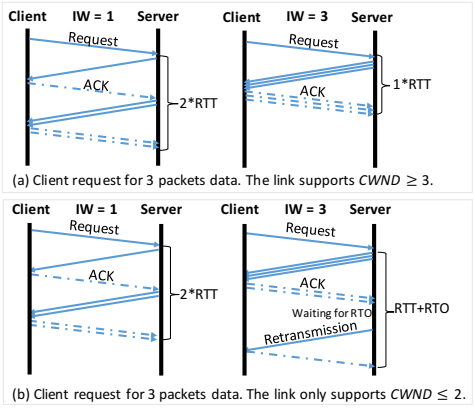


Fig. 2: An illustrative example to show the effect of *IW*.

III. Core Ideas and System Overview

As mentioned in §I, the best *IW* is determined by the client-server end-to-end link's network conditions (*e.g.*, available bandwidth and RTT), but the network conditions are highly variable, both temporally and spatially. To deal with the variability of the network conditions, we propose a bottom-up approach to group flows from 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* context continuity requirements.

On the one hand, using *RL* method can naturally deal with the temporal variability of the network conditions. *i.e.*, *RL* aims to *find the best IW 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 client-server end-to-end link's network conditions. If we run *RL* for each user group within which network conditions are similar, the performance of each group can be improved.

A. Why reinforcement learning

Choosing a proper *IW* for a TCP flow is not an easy job. Using data-driven method is a promising direction, but even if we have detailedly logged the networking conditions, it is still hard to decide a proper *IW* since it is highly related to multiple complex factors such as network bandwidth, RTT, router buffer size, flow size, *etc.*, along the end-to-end path between the user and the server. Moreover, all these factors can frequently change over time, which means the proper *IW* changes over time. Reinforcement learning (*RL*), inspired by human behaviorist psychology [8], is a popular technique in machine learning community and is very suitable to cope with

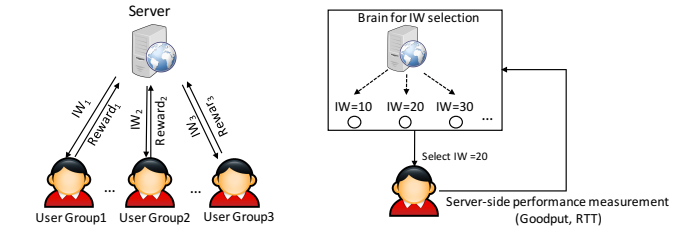


Fig. 3: The key idea of *SmartIW*.

above situation. Basically, it continuously makes decisions based on the environment feedback. Once the optimization goal (called as *reward function* in *RL*) is properly defined, *RL* can gradually find the best decision based on a *trial and error* manner, 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 setting *IW*. With a properly defined reward function, the *RL* algorithm can help automatically find a good *IW*, without worrying about the complex network factors and their variation.

B. How to define the reward function

A naive way would be directly using the TCP response time, which is our optimization goal in this paper. However, the size of web responses may vary largely for different requests, so their network transmission time cannot be directly compared. We note that, *RL* method needs to aggregate TCP flows data together to learn history and choose *IW* by comparing the rewards of different decisions. Therefore, we need a reward function that can accurately reflects the effect of *IW* on the TCP response time, regardless of the response size.

Goodput, which is the number of bytes transmitted per unit of time, is a good candidate that satisfies above requirements. Increasing *IW* to get the best goodput of flows that deploy our approach (called *SmartIW flows*) can hurt the performance of *non-SmartIW flows* that share some network resource with *SmartIW* flows. Therefore, to maintain fairness, we constrain our *RL* optimization goal to be *getting as higher goodput as possible for SmartIW flows while trying best not to hurt non-SmartIW flows*. As such, we take into consideration both *goodput* and *RTT* in the reward function, where *RTT* is a good indicator to the network congestion that affects the performance of non-*SmartIW* flows [14]. Here we do not add *loss* to the reward function, because many *RTT*-based congestion control algorithms [15, 16] work well without considering *loss*.

C. Overview of SmartIW

When the *SmartIW*, shown in Fig. 3, the frontend server receives an HTTP request from a user, it establishes a session with the user and identifies which user group it belongs to. Then the frontend server obtains the most up-to-date decision of *IW* from the per-group *RL*'s result and set this *IW* for the session quickly before sending the response to the user. After the response's transmission is finished, the frontend server

outputs the TCP performance data and reports it to the server cluster which runs per-group *RL* with fresh measurement data and acts as the brain for learning each user groups' *IW*. Besides, the brain runs the user grouping algorithm with the historical data. The brain continuously sends the *IW* decision of each user group to the frontend servers at a timescale of minutes. In this way, it controls all the sessions' behavior. Note that, all the procedures are done at server side without any client or middleware (*e.g.* router, switch, link) modification or assistance, and our method only changes the *IWs* without modifying the TCP congestion control algorithm.

IV. Core Algorithms

In this section, we present two core algorithms of *SmartIW*: *RL* algorithm for learning per-group *IW* given a user group (§IV-A) , and user grouping algorithm (§IV-B).

A. *Reinforcement Learning*

In this paper, we formulate the *IW* learning problem as a non-stationary multi-armed bandit problem, a reinforcement learning problem. It focuses on the online performance by striking a balance between exploration (uncharted actions) and exploitation (current optimal actions). Many algorithms for this problem have been proposed [8].

For the stationary multi-armed bandit problem, the basic UCB algorithm [17] has been shown to be optimal [12]. Its assumption is that the unknown distribution of the context 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 adopt discounted UCB algorithm [12] which was proposed to solve the non-stationary bandit problem. The basic procedure is as shown in Algorithm 1. At each time t, the player chooses an arm $I_t \in 1, ..., K$ (an 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.

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

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

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

In order to use discounted UCB in *IW* learning, the keys are the definitions of the reward function and the arms.

Reward function definition: Our goal is to set the ideal *IW* which can fully utilize the client-server end-to-end link's bandwidth without causing congestion. As mentioned in §III-B, we consider RTT as the signal of the network congestion. The reward in Equation 4 reflects our goal of maximizing the goodput and minimizing the RTT. $Goodput_s(i)$ is arm i's instantaneous goodput at time s, $RTT_s(i)$ is arm i's RTT at time s. $Goodput_{max}$ is maximum goodput in the history measurement, and RTT_{min} is the minimum RTT in the history measurement. α is the parameter which strikes the balance between the goodput and RTT. Small α favors low RTT, which may make the algorithm be conservative with a small *IW*. Large α favors high goodput, which may make the algorithm be aggressive with a large *IW*.

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

Arms definition: The list of arms in 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 the large decision space quickly. Brutally searching the whole 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 the *IW*. At first, we start with a short list of *IWs* as the arms, and the value in the arm list is dynamic based on the arms' performance.

The basic procedure is shown in Fig. 4. 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 to update the arm list. The basic idea is to check whether the largest arm IW_{large} or smallest arm IW_{small} is currently the best arm. If yes, we update the arm list; else the arm list keeps the same. The best arm is the arm having largest reward and smallest value of padding function in Equation 3. 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), too large and too small *IW* are both 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 *IW* space. If the current best *IW* is not the largest or smallest, the arm list keeps the same.

B. *User grouping*

In reality, the users' network conditions have large diversity because users have different network features (*i.e.*, *subnets,*

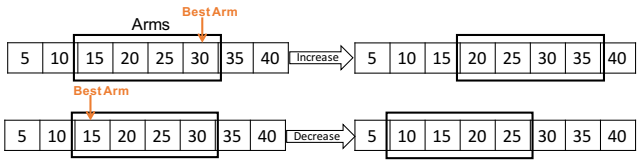


Fig. 4: The procedure of the sliding-decision-space method.

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 the *B* company's geolocation database.

ISP, province). For the users coming from different provinces (*e.g.*, *Beijing, Shanghai*) and ISP (*e.g.*, *CHINANET, CMNET, UNICOM*), both 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 per-link. However, each link hardly has enough samples for *RL* to learn *IW*. Thus, we argue that grouping users with similar network features to share their samples is a promising solution. However, grouping users is challenging due to the following dilemma. 1) Too fine-grained a user group (*e.g.*, *IP*) typically lacks enough samples to monitor its network performance continuously, which cannot satisfy the requirement of *RL*; 2) Too coarse-grained a 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., keep continuity in network conditions)*. The basic idea is using a bottom-up (thus finest-to-coarsest) searching technique 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 goodput and RTT.

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 *B* company's geolocation database by IP, which is similar to the geolocation database [18], we can infer the other network features such as *subnet, ISP, province etc.* Table II shows an example of the geolocation database. Note that an IP only belongs to one record of the features in the table, and all the records are mutually exclusive in IP space. Thus the structure of user grouping result forms a 4-layer (subnet, ISP, Province, All) tree. Fig. 5 shows an example.

We say a user group has enough samples when it has at

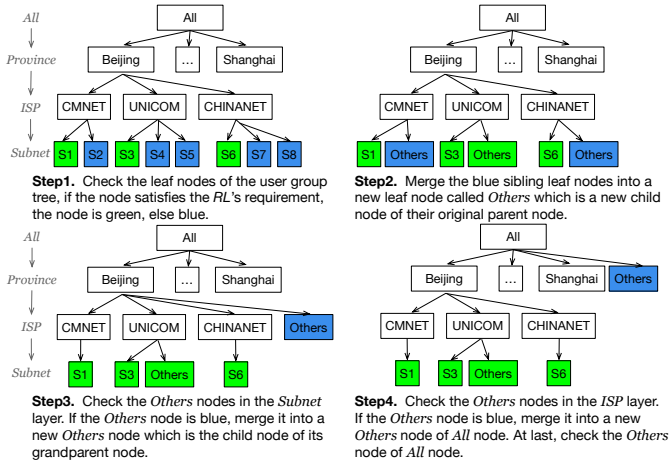


Fig. 5: The procedure of the user grouping algorithm.

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 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 the network conditions. n denotes the number of time bins, and X_s is the reward at time bin s. Since *IW* affects the reward, when computing J, the *IW* should remain the same in each time bin. Note that a small J means the change of network condition is small. To apply *RL* method to a given user group, the smaller J, the better. Here we choose a threshold T, if 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, assuming the finest users' network feature is the *subnet* and the coarsest feature is *All*. *Beijing* has three ISPs: *CMNET*, *UNICOM* and *CHINANET*, they have 8 subnets *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, S6* (in green color) satisfy the *RL*'s requirement and *S2, S4, S5, S7, S8* (in blue color) do not, so *S1, S3, S6* are three finest user groups that can use *RL* to learn *IW*.
- Step 2: the sibling leaf nodes which 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, *S1* is turned into the *Others*, a new child node of *CMNET*. *S4* and *S5* are merged into the *Others*, a new child node of *UNICOM*. *S7* and *S8* are merged into the *Others*, a new child node of *CHINANET*.
- Step 3: we check whether *Others* nodes satisfy the *RL*'s requirement. If the *Others* node doesn't meet the requirement, it needs to be merged with its parent (ISP)'s sibling's *Others* nodes, 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 the *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 the leaf nodes (except root's child *Others* node) satisfy the *RL*'s requirement. Finally, if the *Others* of *All* doesn't meet the requirement, we use the standard *IW* [6] 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*.

V. Design and Implementation

In this section, we present the system design and implementation of *SmartIW* as shown in Fig. 6. It has three keys components. 1) *Connection Manager* is a module implemented in the web proxy (*e.g.*, *Nginx* [19]), which is deployed at frontend servers. Its basic functions are setting a suitable *IW* for each TCP session and output the performance data for each TCP session. It has a configuration file called *IW Table*, which stores each user group's *IWs*. 2) *Data Collector* collects and stores all the performance data of the frontend servers. It provides the fresh data for *Reinforcement Learning*. (Note that these two components can be used beyond *SmartIW*, *e.g.*, for TCP performance troubleshooting.) 3) *Reinforcement Learning* runs *user grouping* and *RL* algorithms in §IV based on the fresh data and updates the *IW Table* for *Connection Manager* periodically. It is the controller of the *SmartIW* system.

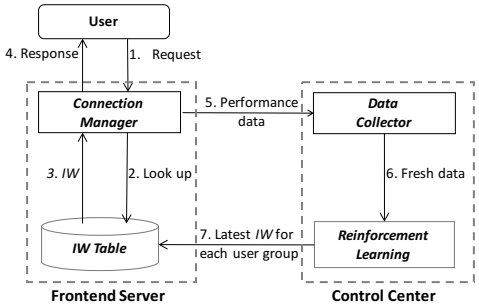


Fig. 6: System Design

A. Connection Manager

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

To realize the functions of *Connection Manager*, we implemented a new module in a web proxy (*e.g.*, *Nginx*) and modified Linux kernel. To be a robust and easily controllable system, most of the jobs are done in application level in the web proxy, such as getting user's IP after TCP three-way handshake, looking up the *IW* from the *IW Table*, *etc.* The

TABLE III: The performance data in our system

TCP Metrics	Description
Size	The data size of the HTTP response.
TCP Response Time	The transmission time defined in Fig. 1.
MSS	Maximum segment size.
Goodput	$\frac{Size}{TcpResponseTime}$
RTT	The 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.
Retrans	The retransmission rate of the HTTP response.
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.

modified Linux kernel's job is just exporting two new APIs to change the value of *IW* and output the TCP performance data. The web proxy cooperates with the modified kernel by calling these two APIs. The first API is *SetIW(fd, iw)*, which is implemented in the *setsockopt* function in Linux, *fd* is the file descriptor of the TCP socket, *iw* is the value of *IW*. When it is called, the socket's *IW* is 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* is the configuration file, providing per-group *IWs* for different user groups. Since the value of *IW* for each user group is based on *RL*'s decision and can change over time, the web proxy also reloads *IW Table* on demand or periodically.

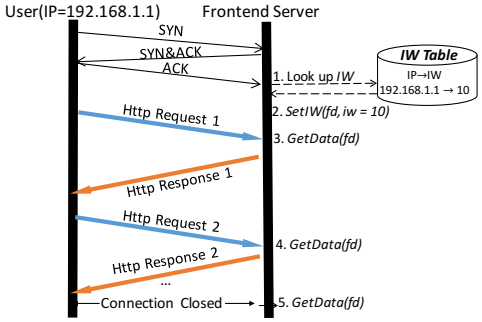


Fig. 7: An example of the connection manager's workflow.

Fig. 7 is a simple example that illustrates the basic workflow of *Connection Manager*. It obtains the use's IP (*e.g.*, 192.168.1.1) when a TCP connection is established, and then looks up the user's *IW* from the *IW Table*. The *IW Table* is stored in a *trie* [20] structure with IP as the key and *IW* the value, which provides longest prefix IP lookup function. The time consumption of query is O(N), where the worst case is N = 32. In this example, the query result is IW = 10. Then it calls the API *SetIW(fd, iw=10)* to set the session's *IW* before sending the response data to 192.168.1.1.

B. Data Collector

The *TCP Response Time* described in Fig. 1 is the key metric for evaluating TCP performance. 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 collecting the timestamp of T_{start} and T_{end} (see Fig. 1). When the web proxy begins to send data to the user or terminate the connection, it calls the *GetData(fd)*, which labels the T_{start} of this HTTP response, and it also records the T_{end} of the previous HTTP response. When it is called, the transmission of the previous HTTP response should be finished and T_{end} is the timestamp of the last TCP ACK.

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

C. Reinforcement Learning

After collecting all the performance data, *SmartIW* runs user grouping and reinforcement learning algorithm in §IV. The user grouping algorithm runs at a long timescale such as day or week. *RL* algorithm runs at a timescale of minutes to continuously learn the suitable *IW* for each user group. The result is the *IW Table* which contains the user groups' *IWs* at the next learning iteration. This module controls all the frontend servers' behavior by updating their *IW Tables*. It is implemented with Golang [21] and Python in *Control Center*.

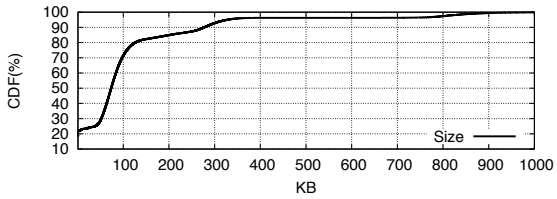
VI. Online Evaluation

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

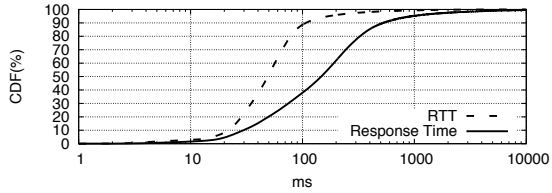
A. Experiment Setup

Fig. 9 has shown 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 [3] for short flows). Besides, as shown previously in Table I, more than 80% of the TCP flows are still in slow-start phase when the sessions end, not 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) 2.40GHz CPUs, 62GB RAM and

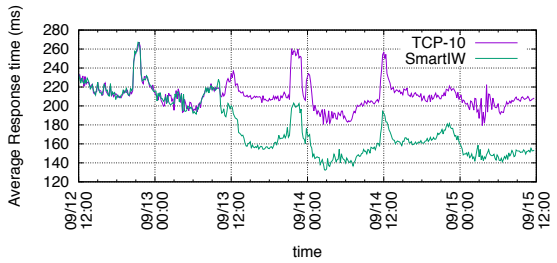


(a) The distribution of response size (KB).

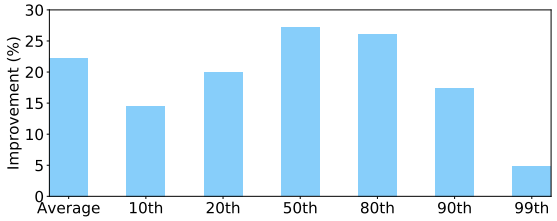


(b) The distribution of RTT and response time (ms). The x-axis is logscale.

Fig. 8: The characteristics of mobile search service in *B*. Response time is the *TCP response time* introduced in §II-A.



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



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

Fig. 9: TCP response time of *SmartIW*.

10Gbps NIC. The Linux kernel version is 2.6.32 and the congestion control algorithm is Cubic [22]. 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.
- *SmartIW*: Our method with group-based *RL*. The learning iteration interval is 10min, which means *SmartIW* will recalculate and update the *IWs* for all user groups in every 10min. The user grouping method takes *Subnet, ISP, Province* as users' network features. The parameters of user grouping is $S_{min} = 100$, $\theta = 0.1$ and the size of time bin = 10min. The *RL*'s $\alpha = 0.8, \triangle = 5$, and $\xi = 0.1$. The initial arm list is *IW* = [5, 10, 15, 20]

B. Overall Performance

Fig. 9a shows an example of the average TCP response time in the two groups of frontend servers. Before 2017.09.13

10:00:00, both groups of servers use *IW* = 10 [6], and their performance proved to be the same. After that, *SmartIW* is started in one group. The TCP response time of *SmartIW* decreases dramatically and continuously outperforms the other group *TCP-10* with about 23% improvement. Besides, Fig. 9b shows that *SmartIW* can make improvement in each percentile of the response time. The 50th and 80th percentiles have been improved by about 25%. We observe that 99th percentile has the smallest improvement, and the main reason is *SmartIW* does not explicitly help with the loss that are part of the reasons that causes the 99th percentile long tail response time [3]. Compared with *TCP-10*, *SmartIW* may appear to be more aggressive in *IW*, but the results show that even in 99th percentile, *SmartIW* also has about 5% improvement over TCP-10. From this we can see that *SmartIW* is also quite cautious when increasing *IW*.

C. User Group's performance

SmartIW uses *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*) which can use *RL* to improve their performance. Fig. 10 shows the jitter J of each these 19 groups. For the user groups *Shanghai, Guangdong, 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 jitter is larger than 0.1 or number of samples are smaller than S_{min}). This is because the requests of these users should be routed to data centers, but are by accident routed to the studied data center in Beijing. Fig. 11 shows that all the use groups' response time have been improved (by about 15% $\sim$ 31%).

VII. Testbed Evaluation

In this section, we use a trace-driven method to systematically evaluate *SmartIW*. We built a testbed which 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 *B*. The testbed experiments validate the following:

1) Both key techniques, *user grouping* and *reinforcement learning*, can help improve the network performance.
2) The flows using a aggressive *IW* can cause network congestion and even hurt their own performance.
3) *SmartIW*'s performance is the closest to the optimal performance.

A. 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 1000Mbps NICs, 64GB RAM and 64 CPUs (2.4 GHz). One machine acts as the server with *SmartIW* deployed, and each of the other 9 machines acts as one user group. All the machines use TCP *cubic* with default configuration.

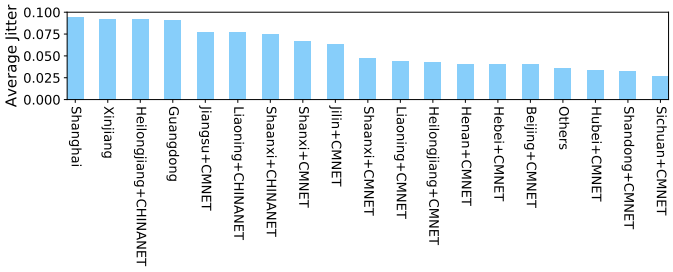


Fig. 10: The average network jitter of each user group.

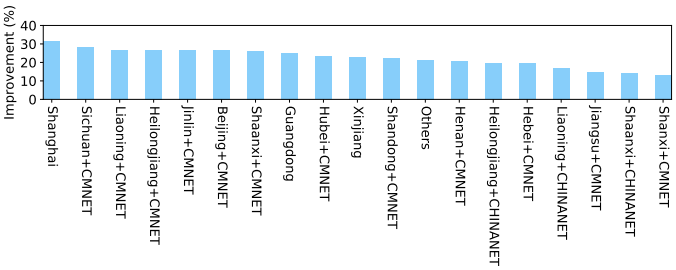


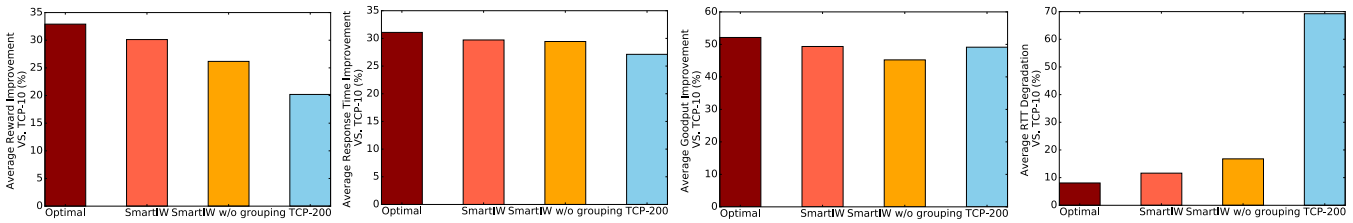
Fig. 11: The average TCP response time's improvement of each user group.

Given the HTTP traffic typically has a daily pattern, we selected one day of data traces for 9 user groups from the online experiments in §VI. 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) using the *90th-percentile goodput* (*average RTT*) from the data traces of this user group and hour. We then simulate the network conditions by using the Linux *TC tool* [23] (with *HTB* and *netem* queue) to shape the traffics.

To systematically evaluate the performance of *SmartIW*, we compare the performance of the following techniques:

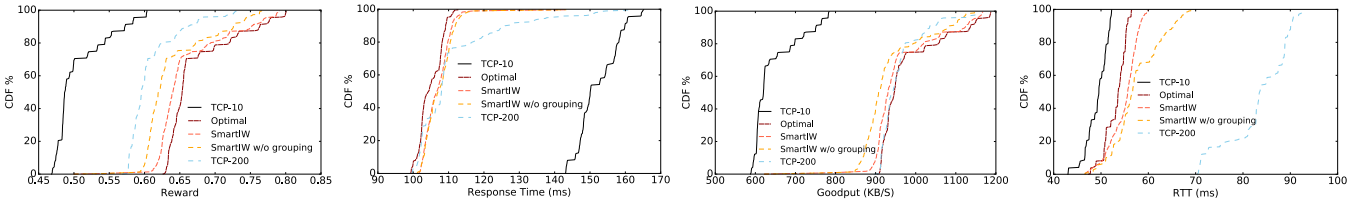
1) *TCP-10*: It is the baseline [6], which uses one static *IW* = 10 for all the flows. The data size of *IW* = 10 is about 14KB ($MSS = 1448$).
2) *TCP-200*: It uses a aggressive *IW* = 200 for all the flows[1]. The data size of *IW* = 200 is about 280KB.
3) *SmartIW*: The learning iteration interval is 10min. The *RL*'s $\alpha = 0.8, \triangle = 1$, and $\xi = 0.1$. The initial arm list is *IW* = [5, 10, 15, 20].
4) *SmartIW without grouping*: Compared with *SmartIW*, it only uses RL to learn the *IW* for all the flows.
5) *Optimal*: It is the best possible performance in the testbed experiment setting, obtained by exhaustively searching

[1]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) Goodput (d) RTT

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



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

Fig. 13: The distribution of each technique's reward, TCP response time, goodput and RTT.

the *IW* space for the best *IW* for each network condition of each user group.

B. 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, goodput, and RTT). We define the technique t's improvement in reward, goodput, and response time as $(reward_t - reward_{tcp-10})/reward_{tcp-10}$, $(goodput_t - goodput_{tcp-10})/goodput_{tcp-10}$, $(responsetime_{tcp-10} - responsetime_t)/responsetime_{tcp-10}$. The degradation of RTT is defined as $(RTT_t - RTT_{tcp-10})/RTT_{tcp-10}$. We can see that *SmartIW* is closest to the optimal, and 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 goodput.

C. Contributions of user grouping and RL

In this experiment, we use reward as the metric since it capture the effects of both goodput and RTT. Firstly, to evaluate the performance of *RL*, we compare the reward of *SmartIW without grouping* with *TCP-10*. Fig. 12a shows that *SmartIW without grouping* has 25% improvement. Fig. 13a also shows that *SmartIW without grouping* greatly outperforms *TCP-10*. Secondly, to evaluate the performance of *user grouping*, we compare the reward of *SmartIW* with *SmartIW without grouping*. Fig. 13a shows *SmartIW* can bring more improvement (29%) than *SmartIW without grouping* (25%). Besides, *SmartIW* also outperfoms *SmartIW without grouping* in goodput (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 *reinforcement learning* can help improve the TCP response time.

D. Aggressive IW's effect:

According to the *common perception of IW* (see §II-B), neither too small an *IW* nor too large a *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 a highest average RTT, which is a good indicator for network congestion. On the other hand, *SmartIW*'s RTT is closet to that of *Optimal*'s, and is better than that of *SmartIW without grouping*. From this we can see, *SmartIW* is quite cautious in avoiding congestions. For average response time and goodput, *TCP-200* outperforms *TCP-10*, but it cannot beat *SmartIW*. 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 *SmartIW* because it suffers from packet loss, which causes costly TCP timeout.

VIII. Related Work

TCP Optimization from within a TCP session: There is a rich body of literature in TCP optimizations which utilize information from within a TCP session only, and do not utilize the valuable information from previous sessions. For example, TCP congestion control algorithms (such as [15, 16, 24, 25]) use a *heuristic-based* trial-and-error approach to probe for the best *CWND within* a TCP session *only*. Remy [26] uses offline-trained machine learning model to dynamically assign the congestion window sizes based on the latest network conditions measured *within* the TCP session *only*. [3, 27, 28] modify TCP protocol to achieve fast loss recovery based on the data collected within a TCP session *only*. None of these above approaches try to improve *IW* or utilize information from history sessions, thus *SmartIW* is complementary to and compatible with these approaches.

***IW* Improvement**: [6] proposed to simply increase the standard *IW* to 10 for all flows, and we have shown that *SmartIW* outperforms this approach by 23%. Halfback [29] always starts with a large *IW*, then applies pacing and redundancy technologies to deal with the loss (caused by the aggressive startup) *within* the flow without using history session information, while we have shown *SmartIW* 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, [30] 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. In comparison, *SmartIW* utilizes much richer history information from the user group, is much more applicable, and only needs to modify the TCP servers.

Cases of Reinforcement Learning in Internet Video QoE Optimization: Pytheas [9] applies *RL* to the video QoE optimization, through dynamically deciding a session's serving frontend server, and differ with *SmartIW* due to domain difference as follows. First, *IW* uses the sliding-decision-space approach to deal with the large *IW* decision space, which is much larger than frontend server selection in Pytheas. [9] only shows testbed evaluation while *SmartIW* has been deployed in real data centers for more than a year. Third, the user grouping methods are different in *SmartIW* (for general TCP performance) and Pytheas (tailored for video QoE). Pensieve [10] improves video QoE through applying Deep *RL* to generate ABR algorithms for each client session given the measurement data from *within* the session, without utilizing history session data.

IX. Conclusion

In this paper, we propose a system, called *SmartIW*, to use group-based reinforcement learning to enable a web server to dynamically set a suitable TCP initial congestion window for a web flow before its transmission starts. *SmartIW* is incrementally deployable at the server side without any client or router support. It doesn't change and is compatible with existing TCP congestion control algorithms. *SmartIW* has been deployed in one of the top global search engines for more than a year. Our online and testbed experiments show that, compared to the common initial window size of 10, *SmartIW* can reduce the TCP response time by 23% to 29%.

We believe that *SmartIW* is an important step towards applying advanced machine learning techniques to solving the hard and open network research problems.

X. Acknowledgment

This work was supported by Baidu Front End team and National Natural Science Foundation of China (NSFC) under grant NO. 61472214, 61472210. We appreciate Qinghua Ding and Jiachang Liu for their survey of *RL* and coding in Nginx.

References

[1] "Latency Is Everywhere And It Costs You Sales," https://goo.gl/bRi5Xs, Accessed: 2018-04-18.

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

[3] 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 *SIGCOMM*, vol. 43, no. 4. ACM, 2013, pp. 159–170.

[4] 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 *INFOCOM*. IEEE, 2016, pp. 1–9.

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

[6] 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," *Computer Communication Review*, vol. 40, no. 3, pp. 26–33, 2010.

[7] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe, "Open Research Issues in Internet Congestion Control," Tech. Rep., 2011.

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

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

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

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

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

[13] 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.

[14] S. Sundaresan, M. Allman, A. Dhamdhere, and K. Claffy, "TCP Congestion Signatures," in *IMC*. ACM, 2017, pp. 64–77.

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

[16] 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 *SIGCOMM*, vol. 45, no. 4. ACM, 2015, pp. 537–550.

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

[18] "DB-IP," https://db-ip.com/, 2018, Accessed: 2018-04-18.

[19] N. Inc., "A Free, Open-Source, High-Performance HTTP Server," https://www.nginx.com/, Accessed: 2018-04-18.

[20] P. E. Black, "Dictionary of Algorithms and Data Structures," Tech. Rep., 1998.

[21] "The Go Programming Language," https://golang.org, Accessed: 2018-04-18.

[22] 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.

[23] "Linux Traffic Control," http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html, 2018.

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

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

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

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

[28] G. Chen *et al.*, "Fast and Cautious: Leveraging Multi-Path Diversity for Transport Loss Recovery in Data Centers," in *USENIX ATC*, 2016.

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

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