TCP WISE: One Initial Congestion Window Is Not Enough

Xiaohui Nie[§] Youjian Zhao[§] Guo Chen [*†] Kaixin Sui [†] Yazheng Chen[§] Dan Pei[§] Miao Zhang [‡] Jiyang Zhang [‡]
[§]Tsinghua University [†]Microsoft Research [‡]Baidu Inc., Beijing, China
[§] Tsinghua National Laboratory for Information Science and Technology (TNList)
{nxh15,cyz14}@mails.tsinghua.edu.cn, {zhaoyoujian,peidan}@tsinghua.edu.cn
{guoche,kasui}@microsoft.com, {zhangmiao02, zhangjiyang01}@baidu.com

***Abstract*—Current TCP is very inefficient for web services. Web transactions are often very short-lived. TCP flow starts with a conservative initial congestion window (IW), which causes multiple round-trip times to finish the transmission even if the end-to-end bandwidth is sufficient for the transaction to be finished in one round-trip time. Previous research efforts have been focusing on finding the overall best IW for the entire Internet or a service company. However, we observe that one-IW-fits-all is suboptimal after one year of online measurement in Baidu, one of the top global search engine companies. To reduce the TCP latency, we propose TCP WISE, which dynamically assigns suitable IWs for different user cluster at different times on the server side. The values of users' IWs are proactively learned based on the historical experience on the server-side. Our testbed experiments show that our learning algorithm can handle the network changes and converge to the best IW. We have deployed TCP WISE in one of Baidu's production data center, and results shows that the 80^{th} percentile latency of the HTTP responses has been reduced by 10.4% compared with current TCP with a fixed IW of 10.**

***Index Terms*—TCP, Initial congestion window, Web performance**

I. Introduction

Over the past few years, web service has become a major way for billions of users to access the rich resources on the Internet. As bandwidth is getting relatively large and cheap, the latency has become the crucial performance metric for web services. Even slightly increasing latency leads to noticeable decrease in revenue and affects users' experience. For example, reports suggest that increasing 400ms web search latency caused a decrease of about 0.74% in Google's search frequency [1], and Bing found that a 2s slowdown would reduce the revenue by 4.3% [2].

Currently, data transmission of most web services (*e.g.*, Microsoft [3] and Baidu [4]) are based on TCP. Prior works have put much effort on reducing the TCP transmission time, however, it is still far from ideal. One of the major bottlenecks in achieving a low TCP latency, is caused by the *flow startup problem* [5]. Specifically, when flow is just established, current TCP has no knowledge about the current available bandwidth As such, for congestion-wise safety, TCP starts from a conservative initial congestion window (IW) and probes the network with slow-start mechanism [6]. This procedure incurs long latency which requires multiple RTTs until getting converged to the actual available sending rate. This problem has been highlighted when the available bandwidth is high and RTT is relatively long in modern Internet.

Although plenty of works (*e.g.*, [7–9]) have improved a lot on the congestion control, they only help to quickly converge to the right available bandwidth during the transmission of flow. However, at *the flow startup phase*, the TCP sender has very little (if nothing at all) information on the current network condition. As such, it is very hard to decide the sending rate at the flow startup. Due to this inherent challenge, although several works have tried to improve the speed of probing the initial sending rate by advanced slow-start algorithm (*e.g.*, [10]) or with the help of feedback signal from the network (*e.g.*, [11]), the flow startup problem is only mitigated but not directly solved. This has become an open issue of TCP [5].

Revisiting the problem, we ask ourselves: *can we directly assign the flow with the right sending rate at the initial setup phase?* Luckily, after one-year of continuous measurements and experiments in Baidu, one of the world's largest web service providers, we found the answer to be *affirmative*. Our intuition is to explore the optimal initial sending rate based on historical experience. Modern web-service platform has mature and powerful log system, generating rich data which can detailedly record each flow's performance (*e.g.*, flow completion time). Servers can achieve flexible control of the sending rate. Based on that, we propose a system called TCP WISE. It runs on the web-service platform, which *explores the appropriate initial sending rate for different flows before starting transmission, through periodically learning from the historical experience*.

There are several challenges that TCP WISE needs to address. First, a flow may only access our web-service for a few times, and it is challenging to get useful information from such sparse history data. To address this challenge, TCP WISE classifies different flows (which may come in at different time) into clusters. Then we learn and set the same initial sending rate for such a cluster. Second, how to set the initial sending rate without interfering the existing well-tuned congestion control algorithms. To avoid interference, TCP WISE implements a flexible API to sets the initial sending

* Guo Chen is the corresponding author.

978-1-5090-6468-7/17/$31.00 ©2017 IEEE

rate only by setting the IW, but without any modification to the existing congestion control algorithms. Given that, the third challenge is how to get the best suitable IW from historical data. On one hand, directly deriving the best initial sending rate only from historical TCP-level performance metrics (*e.g.*, RTT, loss rate) is quite challenging and may be inaccurate. On the other hand, all the flows in the studied company only have used one IW before (which is 10, a practice similar to [12]). As such, TCP WISE leverages a self-evolved close-loop learning manner to proactively explore the best IW for each cluster. Specifically, it calculates a set of possible suitable IWs for each cluster based on their IWs used before, and serves them with a randomly chosen IW from this set. Later, it will update the IW set based on the last round of flow performance under different IWs. With such round-loop learning, TCP WISE can gradually figure out the suitable initial sending rate for each flow cluster. Note that TCP WISE is a continuous process rather than a one-time shot.

To the best of our knowledge, TCP WISE is the first to provide a systematic approach for directly setting the suitable initial sending rate for different web transaction flows based on historical experience[1]. At last, we both use testbed and online experiments to prove that our system works well. To summarize, our main contributions are listed as follows:

- To speed up web transmission, we firstly propose a flexible system that could use different IWs for different user clusters in dealing with the complex and heterogenous network conditions.
- We propose a novel IW learning algorithm. It uses a set of IWs during the same period. By continuously *online A/B testing*, the server learns which IW is better and wisely performs rate control to converge to the appropriate fair sending rate.
- TCP WISE can be easily deployed with only server-side modification. It has been deployed in Baidu, and our experiments show that TCP WISE can improve the 80^{th} percentile latency by 10.4% for about 250 million flows.

II. Background

A. TCP Latency

For latency-sensitive web service, the web response time directly affects user experience. Figure 1 shows the general HTTP request/response timeline between the user and the web service provider. It is a general architecture for web services. Usually, the frontend server accepts the user's request from the Internet, then proxies the connection to the internal backend servers which handle complex functionalities of the web service in the data center. After getting the response data from those backend servers, the frontend server forwards the response back to the user.

[1]It is noteworthy that some previous works also focused on setting appropriate IW for web transaction flows. According to the *experience* with network evolution, Google proposed to increase IW from 2∼4 to a *fixed* 10 in 2010 [12]. However, one initial congestion windows is not enough for largely diverse bandwidth and bottleneck buffer sizes of different user networks. While using larger IW for high-speed users can reduce the latency, for low-speed users, conservative IW is a better choice.

Generally, the web response time consists of two parts, 1) the data transmission time from frontend servers to user and 2) server processing time and backend intra-datacenter transmission latency. In this paper, we focus on the first part, data transmission time from frontend servers to user. We define this time period as *TCP latency*, as shown in figure 1. Specifically, *TCP latency* can be calculated as $(T_{end} - T_{start})$. T_{start} is the moment when the frontend server begins to send data to the user, and T_{end} is the moment when the frontend server receives the last ACK of this HTTP response [2].

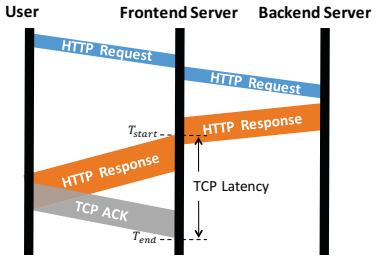

Fig. 1. The detail timeline of the HTTP request/response.

B. One Initial Congestion Window is Not Enough

Initial congestion window (IW) determines the initial sending rate of the transmission. It indicates how many bytes of data can be sent at the beginning. One single small IW value (2∼4) for all TCP flows has remained unchanged since 2002 [13], which is not efficient for nowadays complex network conditions. Improper IW values will result in high latency. For example, a conservative value will lead to poor bandwidth utilization in high-bandwidth network environments, while an aggressive value for low-bandwidth network will cause congestion loss and suffer expensive retransmission timeout.

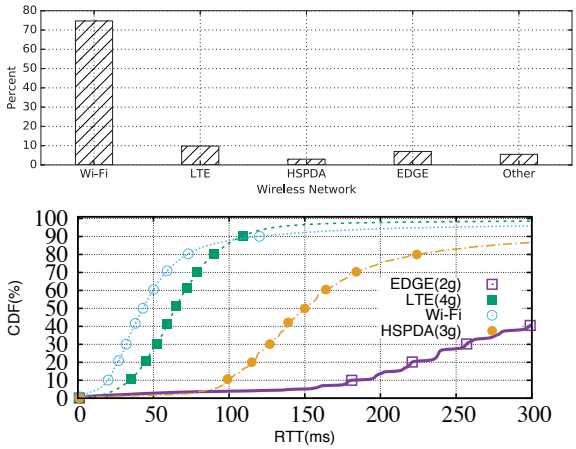

Fig. 2. The percentage of flows from different wireless accessing network (including 2G, 3G, 4G and Wi-Fi), and the distribution of their RTT.

Obviously, it is suboptimal to set one single IW for different flows which experience various network conditions. For example, different accessing network has different network conditions. Taking the wireless network as the example, we have done a large scale measurement during two months,

[2]The last ACK's transmission time is closely approximated to the time of receiving HTTP request

sampling about 300 million flows in one web service. Fig. 2(a) and Fig. 2(b) show the percentage of flows and the round-trip time of different wireless accessing network, including 2G (EDGE), 3G (HSPDA), 4G (LTE) and Wi-Fi. This data is collected from client side with Baidu App. It indicates that different flows served by the same web servers may have hugely different network conditions, and they should be set with different IW. For example, 2G network's RTT is 300∼1000 ms, and 4G network's RTT is 10∼100 ms. As such, assuming MSS = 1440B, the suitable IW that exactly utilize the bandwidth of 2G and 4G network should be 3∼36 and 1∼456, respectively [3].

III. TCP WISE Key Idea

TCP WISE learns the suitable IW for each flow using the following two steps in figure 3:

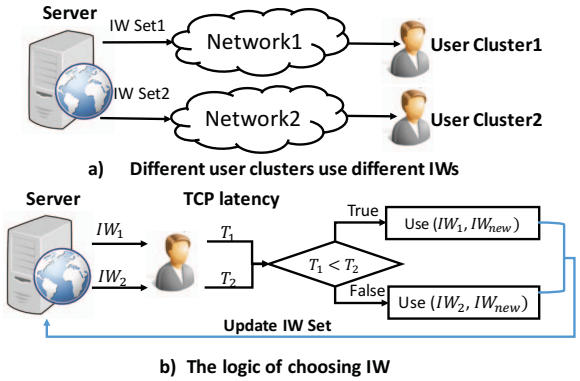

Fig. 3. The key idea of TCP WISE

1) As figure 3 a) shows, it uses different IWs (IW set) for different user clusters. One-IW-fits-all is suboptimal. Its goal is to achieve a fine-grained control and performance improvement.

2) For each user cluster, TCP WISE continuously performs *A/B testing* and adjusts the IW based on the performance objective (*TCP latency*). Specifically, the server randomly selects one IW from the candidate set (IW_1, IW_2) during one fixed time interval. After that, it directly measures the *TCP latency* T_1 and T_2 of IW_1, IW_2. If T_1 is smaller (meaning IW_1 is better than IW_2), then we keep using IW_1 and vice versa. Besides, a new IW IW_{new} is added for the next round of *A/B testing*. The detailed algorithm is described in section IV. Its basic procedure is like gradient descent, continuously updating each user cluster's IWs based on historical experience to find the optimal IW.

IV. System Detail

In this section, we introduce the detailed design of our system in figure 4, which consists of three components called *Connection Manager*, *Data Collector* and *Performance Oriented Learning*.

1) *Connection Manager* is a module implemented in the web proxy (*e.g.* Nginx [15]) which is deployed at the frontend

[3]Calculated using the equation bandwidth = CWND*MSS/RTT. Typically, 4G network's bandwidth is 1∼50 Mbit/s and 2G network's bandwidth is 100∼400 Kbit/s [14].

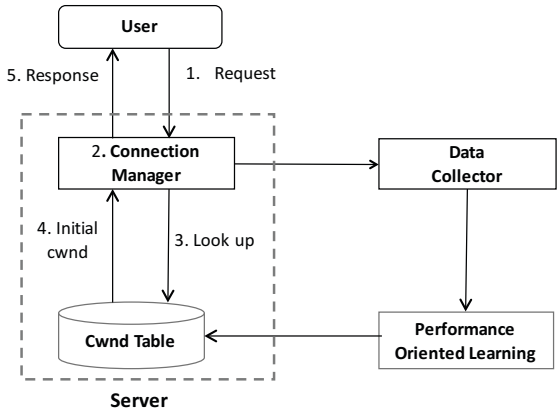

Fig. 4. System architecture

server. When the server establishes a TCP connection with a user, it looks up the *Cwnd Table* to find its appropriate IW, and then it immediately sets the IW for the TCP session to accelerate the transmission. *Cwnd Table* is a configuration file which describes the user clusters and their IWs.

2) *Data Collector* collects and stores all the performance data. It provides historical data for *Performance Oriented Learning*.

3) *Performance Oriented Learning* learns each user cluster's IW from the historical performance data and updates the *Cwnd Table* for *Connection Manager* periodically.

In this way, TCP WISE builds a close-loop scheme to achieve consistent high performance and evolves with network development.

A. Connection Manager

The jobs of *Connection Manager* are real-time identifying the user of the flow and setting IW before the flow transmission begins. When the frontend server establishes a TCP connection with a user, it firstly obtains user's features (*e.g.*, IP) and queries the *Cwnd Table* for a set of IW. Then the server randomly chooses one IW and finishes the IW setting on this TCP session immediately. All the procedures are quickly finished before the data transmission. Note that *Cwnd Table* is the most important configuration of TCP WISE, it records user clusters and their IWs. The number of IWs for one user cluster is more than one. Because of our basic idea is constantly performing A/B testing to explore the suitable IW.

Figure 5 is a simple example to explain the basic workflow of *Connection Manager*. Assuming we treat each IP as a user cluster. *Connection Manager* obtains the IP(192.168.1.1) when TCP connection is established, and then it looks up the 192.168.1.1's IWs from the *Cwnd Table*. The *Cwnd Table* is a hash map with IP as the key and a set of IWs as the value. The result is the set [10, 15]. Then TCP WISE randomly chooses one value from the set with equal probability, assuming it is 10. It calls a socket API *setsocketopt(IW=10)* to change the session's IW before sending response data to 192.168.1.1. The socket API *setsocketopt(IW)* is a new API implemented in Linux kernel by us. To be a robust and flexible system, the system kernel's job is only changing the value of initial congestion window, and it does not change the TCP congestion

control behavior. Hence, TCP WISE is compatible with any congestion control algorithm with slow start.

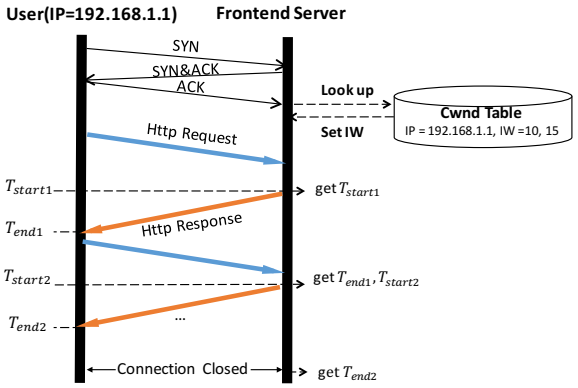

Fig. 5. A simple example of TCP WISE's online workflow, including setting IW and collecting data procedure.

B. Data Collector

Our IW learning algorithm is based the TCP performance. For each HTTP response, we obtain the TCP data including user's IP, data size, *TCP latency*, round-trip time (RTT), TCP retransmission rate, TCP timeout event and *etc.* Here we mainly introduce *TCP latency*. It is the response time described in figure 1 and is the key performance indicator of the web system. Generally, till now there is no method to obtain *TCP latency* at server side. Here we use a sophisticate method to record the latency with only server modification. The key point is collecting the timestamp of T_{start} and T_{end} (see Fig. 5). When the web proxy begins to send data to the user or terminate the connection, it calls a socket API called *getsockopt(tcp data)* implemented by us, and it is similar to the API to obtain the *tcp_info* data structure; besides, it labels the T_{start} of this response, and it also records the T_{end} of previous response. When it is called, the previous response should have been delivered and T_{end} is the timestamp of last TCP ACK from the user.

After the TCP session finishes, the web proxy outputs the performance data for each response. At last, all the performance data will be gathered to a centralized data storage platform. *Performance Oriented Learning* takes the data as the basic input.

C. Performance Oriented Learning

In this section, we mainly introduce how to learn the *Cwnd Table* used in *Connection Manager*.

1) User Clustering: Ideally, TCP WISE would be able to group user at the granularity of an IP address. However, this does not scales because too fine-grained clustering will suffer dimensionality and it also would be lack of samples. The choice of user cluster size must maintain the balance between scale and accuracy. In our paper, we adopt the prefix (24/ IP prefix) as the user clustering method. The users in the same 24/ IP prefix generally belong to the same ISP and region. Conceptually, this is the simple way that the network operators build the network. A /24 IP prefix is also the longest prefix advertised in global BGP routing tables. Clients from same /24 IP prefix will have similar web load time [16]. We acknowledge /24 IP prefix is not the best user clustering scheme. Although users from the same /24 IP prefix could still have different end-to-end performance, it may sacrifice the accuracy. Compared with one IW for all the users, /24 IP prefix could achieve better accuracy. Our learning algorithm is maximize the performance for overall cluster, not single flow.

2) Performance Objective: Our goal is to find the best IW for each user cluster, so what is the "best" IW. In our scenario, we use 80th percentile latency as the performance objective. The 80th percentile is a standard metric used for network-related performance across many Baidu teams. **Here we define the "best" IW is the smallest IW which could minimize the 80th percentile *TCP latency*.** The best IW should fully utilize the bandwidth without overfilling the pipe.

3) Learning Algorithm: Figure 3 shows the basic idea of our learning algorithm. Recall that small IW leads to low utilization and large IW causes congestion loss. According to TCP domain-specific knowledge, learning the best IW is actually similar to find the optimal value in a quadratic function, as shown in figure 6. The learning algorithm consistently discovers each user cluster's IW and updates fresh *Cwnd Table* for online optimization. The basic procedure is as the **algorithm 1** shows. For each user cluster, their IWs is a set

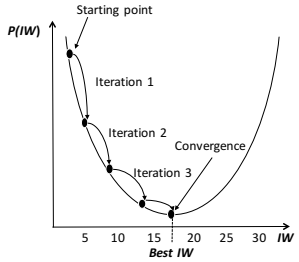

Fig. 6. The basic idea of our learning algorithm, it is similar to gradient descent. P(IW) is the performance objective.

in the initial state. For example, we use two IWs: basic IW α and probe IW β, which are the control and variation in the *A/B testing*. Here we define Δ as the minimum window size for increasing or decreasing IW. In the initial state, $\alpha = 10$ (as [12] proposed) and $\beta = \alpha + \Delta$. The algorithm is triggered periodically. Here we define a learning interval (θ), it is a fixed size of time window. TCP WISE tests α and β's performance during a θ in order to collect the evidence for decision making. After that, TCP WISE enters decision making state. Note that the size of initial IW set could be more that two for fast converging, and during the next learning intervals, two IW is enough.

In the decision making state, per-cluster logic is comparing the performance of different IWs. TCP WISE figures out which IW is better and updates the IW set for the next θ. If the performance of β is better than α, $\alpha = \alpha + \Delta$, $\beta = \beta + \Delta$. Otherwise, $\alpha = \alpha - \Delta$, $\beta = \beta - \Delta$. Based on the comparison result, it updates the new IW set in the next θ. TCP WISE randomly chooses the IW with equal probability, in order to control other factors (such as access time, response size

Algorithm 1 The algorithm of performance oriented learning.

Input: IW set (α, β), step size Δ, learning interval θ, S_{min}
Output: ***Cwnd table*** (each user cluster's IWs in the next learning iteration.)

function PERFORMANCE ORIENTED LEARNING
    1. Startup phase: init each user's IW set (α, β).
    2. Online Testing: testing IW set during a θ online.
    3. Decision Making:
    for each user cluster $user_i$ **do**
        $IW\ Set_i$ = DecisionMaking($user_i$, $IW\ Set_i$)
        Output $user_i$ and $IW\ Set_i$ to ***Cwnd table***
    end for
    Go to 2. Online Testing
end function

function DECISIONMAKING($user_i$, $IW\ Set_i$)
    if $user_i$'s data sample number $< S_{min}$ **then**
        Keep collecting $user_i$'s data in the next θ.
        return $IW\ Set_i$
    else
        find the IW X with largest $P(X)$
        if X is the largest IW in $IW\ Set_i$ **then**
            return X, $X + \Delta$
        else
            return X, $X - \Delta$
        end if
    end if
end function

distribution and *etc.*) similar, which makes the *A/B testing* reliable. Besides, the network condition changes over time, so we also learn the best IW in 24 hours individually. Here we define a minimize sample number S_{min}. Testing with a sample number may not be statistically significant. In each θ, for each IW, the user's responses sample number must large than the S_{min}; if not, TCP WISE can not figure out which IW is better, and it will keep the same IW set in the next θ in order to collect more data evidence.

The smallest value of Δ is 1. It is obvious that large Δ can increase the speed for searching best IW, but it is not accurate and maybe hard to find the best IW. Small Δ could search more accurately but its searching speed is low. In our experiment, we preliminarily use $\Delta = 5$ (almost 7KB), and it is usually the smallest step size that the operators conduct *A/B testing* when searching the best IW for entire web service.

In this way, after several learning intervals, different users' IWs would be different because of their diverse network conditions. The ideal ending is that all the users reach the converging point.

V. EVALUATION

In this section, we both use testbed experiment and online experiment on *Mobile Search* service of Baidu to show the following key points. 1) Our testbed experiment shows TCP WISE can converge to best IW over time and handle the network changes (bandwidth, RTT, loss). 2) Our real experiment shows TCP WISE could reduce the 80^{th} percentile latency by 10.4% with little impact on TCP retransmission rate.

A. Testbed Experiment

*1) **Testbed setup**:* We build a small 1Gbps server-client testbed in Baidu data center. It consists of two servers located in the same rack with 10 Intel Xeon 2.30GHz CPU, 128GB RAM and 10Gbps NIC, both servers run Linux 2.6.32 kernel with Cubic[7] as the congestion control algorithm. We deployed TCP WISE in one server with web service, which acts as the frontend server. The other server acts as the user to send requests to the fronted server. Besides, we use traffic control tools [17] to emulate different network conditions (Bandwitdh, RTT, loss rate). The learning interval of TCP WISE is one minute, user server sends about 100 requests to frontend server in one minute. To evaluate whether TCP WISE could converge to best IW and handle the changes, we need the ground truth (best IW) for each condition. Here we perform the experiments with 100KB responses and some certain network conditions. The ground truth is built by brutally searching the best IW from 1 to 100 in our testbed in advance. The initial IW set (α, β) is equal to (5, 10), the step size Δ is 5.

*2) **Algorithm convergence and network changes**:* In this section, we mainly describe TCP WISE's behavior under these certain network conditions (Bandwidth = 10Mbps, 20Mbps, RTT = 10ms, 20ms, loss rate = 0, 10%). We mainly perform six kinds of network changes to prove that learning algorithm can handle network changes and realize convergence, including bandwidth decrease and increase, RTT decrease and increase, loss decrease and increase.

Bandwidth changes: Bandwidth can change without the path changing, *e.g.* when a link changes its rate. Figure 7 shows each learning iteration's IW and *TCP latency* compared with the ground truth. From this we can see, during the 1$\sim$20 learning iterations, TCP WISE increases its IWs and converges to the IW best (ground truth). The *TCP latency* decreases from 123ms to 61ms, bringing about 50% improvement. After decreasing or increasing the bandwidth, TCP WISE can also sense the changes and converge to the best IW. Note that when decreasing bandwidth in 21 learning iteration, TCP WISE's latency is still equal to ground truth, because the reason is that TCP WISE's IWs are larger than the best IW, and it fully utilizes the bandwidth without resulting bufferbloat, but its bad effect is high RTT because packets are filling the pipe and causing long queuing delay [18]. That's why TCP WISE decreases its IW.

RTT changes: RTT can change (for example, on a route change) but still have the same bottleneck bandwidth. Figure 8 shows TCP WISE can also handle the RTT changes (20ms$\Rightarrow$ 10ms $\Rightarrow$ 20ms) and converges to the corresponding best IW.

Loss rate changes: Here we only emulate the random loss by using *netem* tools. Congestion loss can be reflected in bandwidth and RTT changes. Loss rate can change because of malfunctioning hardware failure, *etc.* Ideally, random loss does not affect the size of the best IW. Figure 9 shows TCP WISE can also handle the changes of random loss.

B. Online Experiment

*1) **Experiment Setup**:* We evaluate our system in one representative production data centers, which are located in Beijing. Beijing DC mainly serves the *Mobile Search* service (one of most popular Chinese search engine in the world), its users are mainly from the north region in China, and its daily response number is more than 1 billion. Users'

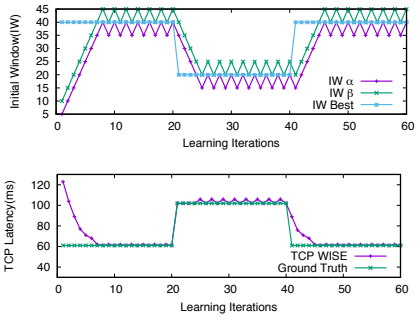

Fig. 7. Bandwidth changes. During 1∼20 and 41∼60 learning iterations, the network condition is (bandwidth = 20Mbps, RTT=20ms, loss = 0). During 21∼40 the network condition changes to (bandwidth = 10Mbps, RTT=20ms, loss = 0).

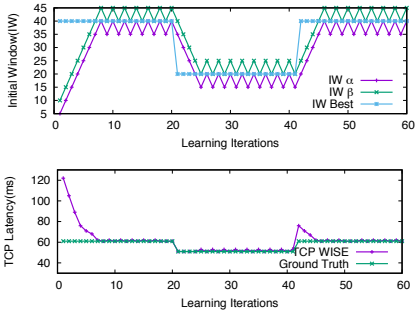

Fig. 8. RTT changes. During 1∼20 and 41∼60 learning iterations, the network condition is (bandwidth = 20Mbps, RTT=20ms, loss = 0). During 21∼40 the network condition changes to (bandwidth = 20Mbps, RTT=10ms, loss = 0).

network access technology is almost wireless network (Wi-Fi or cellular network).

The HTTP transactions are randomly load-balanced according to users' IP addresses and ports. The users in the same subnet are served by the same data center. The frontend servers run TCP Cubic [7] with default setting.

Typical A/B testing scheme is adopted in our evaluation. Before being deployed with TCP WISE, the 24-hour measurement results show that **different frontend servers in the same DC is similar in latency distribution (less than 0.5% difference in each percentiles).** Hence, we choose 4 $\sim$ 8 of servers in each DC, enable half of them with TCP WISE

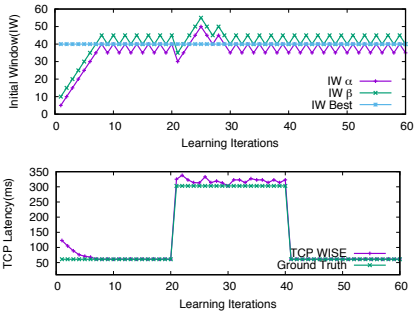

Fig. 9. Loss rate changes. During 1∼20 and 41∼60 learning iterations, the network condition is (bandwidth = 20Mbps, RTT=20ms, loss = 0). During 21∼40 the network condition changes to (bandwidth = 20Mbps, RTT=20ms, loss = 10%)

(setting IWs for different user cluster in different hours), and keep the rest of the servers using TCP-10 (TCP with IW = 10 according to [12]) as the baseline. We conducted this A/B testing experiment for about one month. The minimal sample size S_{min} = 100.

As for TCP WISE's input parameters, the learning interval (θ) is one week in our experiments. Because one week data can reflect the periodicity of network changes. The IW learnt in last week could be used in next week. We preliminarily define $\Delta = 5$ and the minimal IW is 3.

*2) **Mobile Search**:* The web services characteristics including response size distribution, RTT distribution (frontend server to user), retransmission rate *etc.*, are the main factors that influence the web latency. Figure 10 shows the search query response size and RTT distribution of *Mobile Search*. Theoretically, if the bandwidth is unlimited, about 89% response need at least two RTTs if the initial cwnd equals to 10. The average and medium RTT are about 500ms, 60ms. The average retransmission rate (retransmitting data size/total data size) is about 3%. From these we can see, the latency of *Mobile Search* is far from ideal (One RTT), and even reducing one RTT could help a lot.

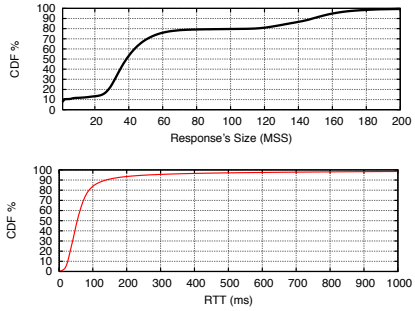

Fig. 10. The CDF of responses' size and round-trip time in *Mobile Search* service. Here the unit of size is MSS (Maximum Segment Size), generally it is 1440B in our data sets. The size means how many TCP windows size are needed for transmission

*3) **IW Learning**:* The IW learning procedure is as shown in the algorithm 1. By the time of this submission, we have performed experiment for about 4 learning iterations (about one month). Here, in order to quickly converge, a fast startup scheme is utilized in our experiment. At the first iterations, all users randomly use 5 IWs (10, 15, 20, 25 ,30) at the same time. In the following three learning iterations, TCP WISE consistently updates the IW set for each user cluster.

At last, 3928 user clusters have fulfilled the learning algorithms requirements (response' sample number $> S_{min}$). About 81.2% of this DC's flows come from these 3928 user clusters. For other user cluster without enough samples, we classify them into a user cluster called "others". At last, TCP WISE learns user cluster's IWs in each hour individually. Here we firstly take the result in 20^{th} hour as the example, and figure 11 is the IW distribution of each user cluster.

From this we can see: **(1) One initial congestion window is not enough. IW =10 is not the best choice. (2) Most user cluster have a larger IW (IW $>$ 10). IW = 30 is the most popular configuration, which means many users' IWs**

converge to 30 at last. Besides, a small group users's IWs (about 2.5%) have decreased bellow 10, which means IW = 10 would hurt their performance. (3) TCP WISE functions well, after 4 learning iterations, some user clusters increase their IWs and some user clusters decrease their IWs.

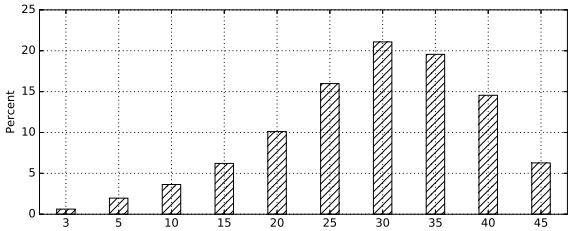

Fig. 11. The distribution of each cluster's IW. X-axis is the IW and y-axis presents the percentage of its user clusters.

*4) **Performance Result**:* Table I shows the latency over one week period. TCP WISE outperforms TCP-10 obviously. TCP WISE has 10.4% in 80^{th} percentile latency. For the the 99^{th} and 99.9^{th} percentile (the tail) latency, TCP WISE also has a 3.3% and 1.4% latency reduction. It proves that TCP WISE is fast and cautious. Although most of TCP WISE's windows is larger than 10 (see figure 11), it does not cause a long tail.

Figure 12 shows the performance result in 24×hour by evaluating one week data. Obviously, TCP WISE consistently has a better performance than TCP-10. As our learning algorithm's performance objective is 80^{th} percentile latency, we mainly introduce the 80th percentile latency here. In most hours, the latency reduction ratio is larger than 10%. In 12^{th} hour, TCP WISE has the best performance (12.3% reduction ratio and 61ms absolute reduction.). In 23^{th} hour, TCP WISE also has 7.9% latency reduction ratio and 36 ms reduction. In fact, the other percentiles (AVG, $20^{th}, 50^{th}, 90^{th}, 99^{th}$) latency also have the similar result. Figure 13 shows the detailed latency reduction ratio of the other percentiles in one week.

Compared with previous works [12, 19], our latency reduction is significant. Google increases IW from 3 to 10 [12], which has a 10% reduction ratio of average latency. Now we take TCP-10 as the baseline, we improve the average latency by 7.9%. Reactive [19] improves the average latency for about 6% by fast loss recovery. TCP WISE only focuses on setting IW, which is compatible with Reactive [19].

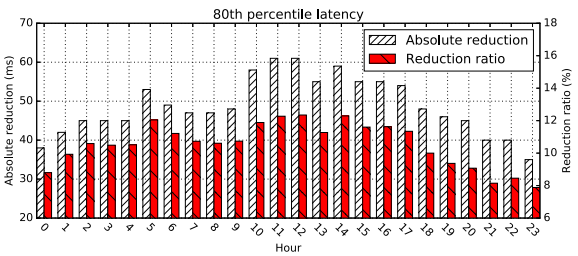

Fig. 12. The 80^{th} percentile latency of TCP WISE compared with TCP 10. The x-axis presents the hour, and the left y-axis presents the absolute reduction of latency and the right y-axis presents the reduction ratio of latency.

C. Negative Impact

Table II summarizes TCP WISE's effect on the average retransmission rate and timeout ratio (#responses whose transmission occurred TCP timeout/#responses). Note that TCP WISE has a slight increase or no increase in retransmission rate and timeout ratio. If packets loss happens, TCP will suffer package retransmission or expensive timeout event. Increasing IW may increase congestion loss, but from the result we can prove TCP WISE is cautious.

TABLE I
COMPARISON OF TCP LATENCY (MS) BETWEEN TCP WISE AND TCP-10 IN *Mobile Search* SERVICE.

Percentile	TCP WISE	TCP-10	Diff [%]
AVG	417	453	36 [7.9]
10^{th}	101	115	14 [12.2]
20^{th}	140	158	18 [11.4]
50^{th}	236	266	30 [11.3]
80^{th}	422	471	49 [10.4]
90^{th}	628	702	74 [10.5]
99^{th}	3617	3736	119 [3.3]
99.9^{th}	15813	16030	217 [1.4]

VI. DISCUSSION

Algorithm limitation: In this paper, we mainly introduce how to discover a appropriate IW for each user cluster (/24 IP prefix) in 24 × hours. The appropriate IW could be used for a long term, that is why we choose learning interval as one week. Figure 13 shows that our IWs could consistently improve the *TCP latency*. As we acknowledged, our user clustering (24/ IP prefix + 24 hour) is not the best scheme. Continually, we will work on better clustering schemes in the future, and believe better clustering scheme will bring better performance.

The limitation of client receive window: TCP's first sending window is equal to min(IW, client initial rwnd, flow size). Client initial rwnd is the initial advertised receive window of the client. Small client initial receive window hinders the potential performance improved by increasing IW. Our experiment result shows that most users' network condition is good enough to have a much larger IW than current 10. We have measured the client initial rwnd from the first HTTP request of the TCP connection in the frontend servers for 24 hours in one DC. About 94.7% IOS, 90% Windows users and 60% Android users have a larger initial rwnd than 40.

VII. RELATED WORK

Reducing the latency of short flow transmission is a hot topic [12, 19, 20]. These related works fall into several broad categories: speeding up startup phase, fast loss recovery.

Speeding up startup phase: For short flow, TCP's slow start is a relative slow mechanism. Many works have been proposed to accelerate the slow start of TCP. [20] uses TCP's recent parameters for fast start and needs router support. Google proposed IW to 10 [12] for easy deployment. RC3 [21]

TABLE II
THE AVERAGE RETRANSMISSION RATE AND TIMEOUT RATIO OF *Mobile Search*.

Metrics	Retransmission Rate (%)	Timeout Ratio (%)
TCP WISE	2.53	5.3
TCP-10	1.93	5.0
Diff	0.6	0.3

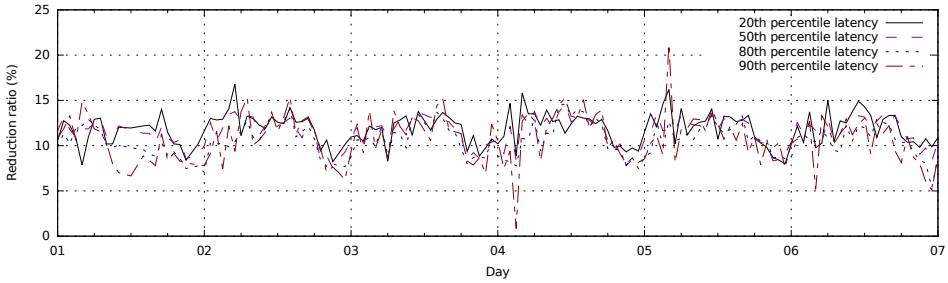

Fig. 13. TCP latency reduction ratio compared with TCP-10 in *Mobile Search* service.

uses a aggressive initial sending rate and requires routers to support priority queues to eliminate the negative effect.

Fast loss recovery: Loss also plays a big role in the slowing down of the short flow transmissions. When loss happens, traditional TCP can only detect it by fast recovery or by a timeout. There have many works dealing with fast loss recovery. FACK [22] handles multiple packet loss in a window. Early retransmit [23] explicitly reduces the fast retransmit threshold when the congestion window is small, in order to rapidly recover from single packet losses. [19] has proposed Reactive, Proactive, and Corrective schemes. These mechanisms try to reduce package loss and detect loss as early as possible. S-RTO [24] is an extension of TCP that helps mitigate timeout retransmission stalls. FUSO [25] leverages the inherent multi-path diversity for transport loss recovery.

VIII. Conclusion

In this paper, we present a system called TCP WISE, which reduces the web latency by setting different initial congestion windows (IW) for different user clusters. The IW for different user clusters is learned from historical experience with a sophisticated method. Besides, our system can be easily deployed with only at server-side modification. It does not change TCP congestion control and is compatible with other congestion control algorithms with slow start.

Our testbed experiments prove that TCP WISE can handle network changes and converge to the best IW. Now it has been deployed in one production data centers in Baidu, one of top global search engine companies. After one month of real deployment and evaluation, our result shows TCP WISE can reduce the 80^{th} latency of the HTTP responses by 10.4%

IX. Acknowledgment

This work was supported by the State Key Program of National Science of China under grant No. 61233007, the National Natural Science Foundation of China (NSFC) under grant NO. 61472214, 61472210, the National High Technology Development Program of China (863 program) under grant No. 2013AA013302, the National Key Basic Research Program of China (973 program) under grant No. 2013CB329105, the Tsinghua National Laboratory for Information Science and Technology key projects, the Global Talent Recruitment (Youth) Program, and the Cross-disciplinary Collaborative Teams Program for Science & Technology & Innovation of Chinese Academy of Sciences-Network and system technologies for security monitoring and information interaction in smart grid.

References

[1] Jake Brutlag. Speed matters for google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, June 22, 2009.
[2] Steve Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, July 1, 2009.
[3] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. *ACM SIGCOMM Computer Communication Review*, 43(4):243–254, 2013.
[4] Dapeng Liu, Youjian Zhao, and etc. Focus: Shedding light on the high search response time in the wild. In *INFOCOM*, pages 1–9. IEEE, 2016.
[5] Open research issues in internet congestion control. https://tools.ietf.org/html/rfc6077, 2011.
[6] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18. ACM, 1988.
[7] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS*, 42(5):64–74, 2008.
[8] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*, 2006.
[9] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50:20–50:53, October 2016.
[10] Sally Floyd. Limited slow-start for tcp with large congestion windows. *RFC 3742*, 2004.
[11] Sally Floyd, Mark Allman, Amit Jain, and Pasi Sarolahti. Quick-start for tcp and ip. *RFC 4782*, 2007.
[12] Nandita Dukkipati, Tiziana Refice, and etc. An argument for increasing tcp's initial congestion window. *Computer Communication Review*, 40(3):26–33, 2010.
[13] Increasing tcp's initial window. https://tools.ietf.org/html/rfc3390, 2002.
[14] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013.
[15] NGINX Inc. A free, open-source, high-performance http server. https://www.nginx.com/, 2017. [accessed 18-May-2017].
[16] Hongqiang Harry Liu and etc. Efficiently delivering online services over integrated infrastructure. In *USENIX NSDI*, 2016.
[17] Martin A. Brown. Traffic control howto. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html, 2006.
[18] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50, 2016.
[19] Tobias Flach, Nandita Dukkipati, et al. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 159–170. ACM, 2013.
[20] Randy H Katz and Vankata N Padmanabhan. Tcp fast start: A technique for speeding up web transfers. In *IEEE Globecom*, volume 34, 1998.
[21] Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Recursively cautious congestion control. In *Proc. of the USENIX Symposium on NSDI*, pages 373–385, 2014.
[22] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining tcp congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 281–291. ACM, 1996.
[23] M Allman, K Avrachenkov, U Ayesta, J Blanton, and P Hurtig. Early retransmit for tcp and stream control transmission protocol (sctp). Technical report, 2010.
[24] Jianer Zhou and etc. Demystifying and mitigating tcp stalls at the server side. In *CoNEXT*, 2015.
[25] Guo Chen et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC*, 2016.