TCP Delayed Window Update Mechanism for Fighting the Bufferbloat

Min Wang[1], Lingyun Yuan[1]
[1]College of Computer Science and Information Technology, Yunnan Normal University, Kunming 650500, P.R. China
[e-mail: honghewangmin@sina.com.cn, blues520@sina.com.cn]
*Corresponding author: Min Wang

Received March 21, 2016; revised July 21, 2016; accepted August 13, 2016; published October 31, 2016

Abstract

The existence of excessively large and too filled network buffers, known as bufferbloat, has recently gained attention as a major performance problem for delay-sensitive applications. Researchers have made three types of suggestions to solve the bufferbloat problem. One is End to End (E2E) congestion control, second is deployment of Active Queue Management (AQM) techniques and third is the combination of above two. However, these solutions either seem impractical or could not obtain good bandwidth utilization. In this paper, we propose a Transmission Control Protocol（TCP）delayed window update mechanism which uses a congestion detection approach to predict the congestion level of networks. When detecting the network congestion is coming, a delayed window update control strategy is adopted to maintain good protocol performance. If the network is non-congested, the mechanism stops work and congestion window is updated based on the original protocol. The simulation experiments are conducted on both high bandwidth and long delay scenario and low bandwidth and short delay scenario. Experiment results show that TCP delayed window update mechanism can effectively improve the performance of the original protocol, decreasing packet losses and queuing delay while guaranteeing transmission efficiency of the whole network. In addition, it can perform good fairness and TCP friendliness.

Keywords: Bufferbloat, Delayed Window Update, TCP, Packet loss, Queuing Delay

This work was supported by the National Natural Science Foundation of China (61561055), the Ph.D. Programs Foundation of Ministry of Education of China (20090181110053), the National Nature Science Fund Project (61262071), Key Project of Applied Basic Research Program of Yunnan Province (2016FA024) and the Ph.D Scientific Research Foundation of Yunnan Normal University (“Research of Adaptive Routing Mechanisms and Transmission Protocols in Software Defined Networking (SDN)”) .

http://dx.doi.org/10.3837/tiis.2016.10.018 **ISSN : 1976-7277**

1. Introduction

Bufferbloat is a problem in a packet-switched network which can occur due to increase in the size of buffer with increase in internet traffic [1]. This creates high latency in network which ultimately degrades the performance of the network. Though Cardozo et al. [2] present that bufferbloat might not be a significant problem in some cases by an experimental analysis, in fact, the situation got worse in the latest years due to mainly two facts: (i) TCP loss-based design, that forces the bottleneck buffer to fill before the sender reduces his rate and (ii) the fact that network traffic keeps increasing. Alfredsson et al. [3] performed extensive measurements to investigate the impact of TCP congestion control on bufferbloat in commercial 3rd Generation (3G), 3.5G and 4th Generation(4G) cellular networks. The results show that the completion time of short flows increases significantly when concurrent long flow traffic is introduced. This is caused by increased buffer occupancy from the long flows. In addition, for 3G and 3.5G the completion time is shown to be dependent significantly on the congestion control algorithms used for the background flows, with Cubic TCP [4] leading to significantly larger completion time.

Although bufferbloat problem is prevalent in current networks, the practical solutions have not been deployed. There exist some potential solutions to bufferbloat including applying AQM techniques, end-to-end congestion control solutions and the combination of both.

AQM techniques [5-11] refer to the techniques which are used to control the amount of data stored in network node buffers. They can proactively drop packets before router buffer space is exhausted. This signals incipient congestion to endpoints and avoids persistently large queues. AQM aims at reducing queuing latency and is one piece of the solution to Bufferbloat. However, AQM methods need to be implemented in the routers and so they can only be adopted in AQM networks that have not been widely deployed in short term.

End to end congestion control solutions keep the core of network princeple unaltered and their implementation is limited to the updates at the end hosts. Jiang et al [12] provide a comprehensive discussion about the potential solution spaces for bufferbloat and think TCP-based end-to-end solution is highly practical. Experiment results show TCP Vegas [13] is resistive to bufferbloat as it uses a delay-based congestion control algorithm. Low-priority congestion control techniques (LPCC) are also delay-based engineering of end-to-end flow and congestion control (CC) alternatives to best-effort TCP and specifically aim at lower than best-effort priority [14-16]. LPCC techniques are initially proposed for background transfers, i.e. they could not compete with other TCP versions for network bandwidth. Therefore, they can prevent the increased delay caused by queuing for most of other users. Unfortunately, despite this and other advantages, some researches [17][18] discover a number of issues in these delay-based protocols, including the inability to get a fair share (fairness in TCP refers that each TCP session sharing the same bottleneck link should have the same average rate) when competing with aggressive TCP Reno-style flows.

The coexistence of both AQM and LPCC-based solutions has been recently studied by Gong et al [19][20]. Ideally, both approaches should be able to coexist transparently. Nevertheless, Gong et al [20] present evidences that this coexistence may cause a problem called “reprioritization”. This problem occurs because AQM queues try to limit the excessive usage of bandwidth to protect new and short duration flows. LPCCs, on the other hand, attempt to use the available bandwidth without interfering with other flows that have low

priority. As a consequence, LPCCs under the influence of AQMs have their low priority ignored and their flows become as aggressive as normal TCP flows, reducing the benefit of LPCCs usage to avoid bufferbloat.

In this work, we aim to figure out a deployable and practical solution to the bufferbloat while maintaining good performance. For this we will handle bufferbloat with end to end solutions because compared to the AQM solutions, end to end solutions have a number of advantages:

• An end-to-end solution is more feasible and light-weight compared to solutions that modify routers.

• Considering deployment issues, an end-to-end solution is easier to deploy and has lower deployment cost.

• The solution can be either server-based or client-based or both. This potential flexibility provides a larger solution space to researchers.

End to end solutions to bufferbloat can bring a lot of benefits, while there are still some design challenges. Although delay-based TCP algorithms (e.g., TCP Vegas) perform normally in bufferbloated networks, they are proved to suffer from throughput degradation when competing with loss-based flows because of the estimate error of delay [21]. Therefore, the challenging issues for the end to end solutions include: 1) how to reliably detect network congestion to effectively prevent the bufferbloat, and 2) what strategy should be applied to maintain good performance across a broad range of the network environments, even when competing with the loss-based flows.

In this paper, a TCP delayed window update mechanism is proposed as a solution to bufferbloat. The mechanism utilities bandwidth utilization and delay jitter instead of direct delay information to predict the congestion level of the network, and then delays congestion window update or decreases moderately the congestion window according to the congestion level. If the network is non-congested, the mechanism stops work and congestion window is updated based on the original protocol. The mechanism can work only when the network congestion is detected. Therefore, it could not affect normal window update and the estimation error of delay could not result in significant throughput degradation. In addition, it can be quickly and easily deployed because only the sender side needs to be modified.

Our main contributions are as follows: 1) the congestion detection approach based on multi-bit control information is introduced in which both bandwidth utilization and delay jitter are adopted to predict the congestion level of networks, enhancing the reliability of congestion prediction, 2) when detecting the network congestion to be upcoming, the delayed window update control strategy is presented to maintain good throughput performance, 3) the proposed mechanism is designed in a modular manner, i.e., it can be implemented in current TCP versions as an independent part and it does not change the congestion control algorithm of the original protocols. Based on Cubic TCP, the preliminary application of the mechanism is investigated over wide network scenarios to verify its feasibility.

The rest of this paper is organized as follows. In Section 2, a brief overview of related work is presented. Section 3 details the proposed mechanisms. In Section 4, simulation-based experiment results are presented. Finally, Section 5 concludes the paper.

2. Related Work

In this section, we present a brief overview of the solutions to bufferbloat, including AQM techniques, end to end congestion control solutions and the combination of both.

2.1 AQM Techniques

AQM techniques affect the scheduling and discard packets in the buffer differently from a traditional FIFO discipline, thereby decreasing queue length and queuing delay. AQM is not a new research field, with numerous techniques proposed over the years such as Random Early Detection (RED) [5], CHOKe [6] and Shortest Queue First (SQF) [7]. RED detects incipient congestion by computing the average queue size. When the average queue size exceeds a preset threshold, the gateway drops or marks each arriving packet with a certain probability, where the exact probability is a function of the average queue size. Unfortunately, due to the fact that RED needs careful tuning of its parameters for various network conditions, most network operators do not turn RED on.

To fight bufferbloat, some novel AQM solutions are proposed [8-11]. CoDel (for Controlled Delay) queue management algorithm [8-9] essentially tries to detect bad queues, which are those that grow harmfully without signs of deflation. In face of a bad queue, CoDel intentionally drops packets to induce the activation of the TCP congestion control. CoDel is self-configurable, which is an advantage over traditional AQM solutions. PIE (Proportional Integral controller Enhanced)[10], which combines the benefits of both RED and CoDel, is proposed as a lightweight algorithm. Similar to RED, PIE randomly drops a packet at the onset of the congestion. The congestion detection, however, is based on the queueing latency like CoDel instead of the queue length like conventional AQM schemes such as RED. Furthermore, PIE also uses the latency moving trends: latency increasing or decreasing, to help determine congestion levels. In order to make the computer generate an AQM for the scenario specified by users, a program called Kemy is developed based on off-line machine learning technologies in Reference [11]. The Kemy-generated AQM is evaluated in various scenarios and achieves the goals of solving bufferbloat problem. Compared to some representative human-designed AQMs, Kemy-generated AQM performs even better in some cases.

AQM can effectively control queue latency thereby solving the bufferbloat. However, AQM has not been widely deployed because of implementation difficulties and general misunderstanding about Internet packet loss and queue dynamics. Therefore, it could not be widely used in the Internet in short term.

2.2 End to End Congestion Control Solutions

End to end congestion control solutions to bufferbloat use delay-based congestion control mechanism to detect congestion earlier, such as LPCC Protocols and TCP Vegas.

LPCC protocols are end-to-end flow and congestion control (CC) alternatives to best-effort TCP and specifically aim at lower than best-effort priority. They are initially proposed for background transfers. TCP Nice [14] modifies TCP congestion control to be more sensitive to congestion than traditional protocols such as TCP Reno [22] or TCP Vegas by detecting congestion earlier, reacting to it more aggressively, and allowing much smaller effective minimum congestion windows. Low Extra Delay Background Transport (LEDBAT) [15] is one of the most popular LPCCs, which is a TCP alternative protocol originally created for Bittorrent P2P networks. It is governed by a linear controller designed to infer the occurrence of network congestion earlier than TCP. Its congestion control algorithm is based on the estimation of one-way delay: queuing delay is calculated as the difference between the instantaneous delay and a base delay, taken as the minimum delay over the previous observations. Whenever a growing one-way delay is detected by the sender, it infers that queue is building up and reacts by decreasing its sending rate. Chirichella and Rossi [16] show that LEDBAT prevents the increased delay caused by queuing for most of Bittorrent users. LPCC protocols cannot compete with other TCP versions for network bandwidth because they

are proposed for background transfers, resulting in bandwidth unfairness. Thus, it is impractical to widely deploy the protocols in current networks.

Some traditional delay-based congestion control protocols can be the potential solutions to bufferbloat [12]. TCP Vegas makes an estimate of the used buffer size at the bottleneck router based on Round Trip Time (RTT) measurements. The minimal RTT value observed during the connection lifetime is considered a baseline measurement indicating a congestion-free network state. In other words, a larger RTT is due to increased queuing in the transmission path. TCP Vegas tries to quantify an absolute number of packets at the bottleneck router as a function of the expected and actual transmission rate. FAST TCP [23] is a typical delay-based high-speed TCP variant derived from TCP Vegas. The protocol maintains queue occupancy at routers for a small but not zero value so as to make the network around full bandwidth utilization and achieve a higher average throughput then TCP Vegas.

In addition, Jiang et al [24] reveals the severity of bufferbloat in current cellular networks with extensive measurements over the 3G/4G networks and proposes a dynamic receive window adjustment (DRWA) scheme which requires only receiver-side modification and can be easily deploy via over-the-air (OTA) updates. DRWA is similar in spirit to delay-based congestion control algorithms but runs on the receiver side. It modifies the existing receive window adjustment algorithm of TCP to indirectly control the sending rate. Im et al [25] address the bufferbloat problem in resource-competitive environments such as Wi-Fi, and propose a receiver-oriented scheme, named Receiver-side TCP Adaptive queue Control (RTAC), to tackle the downstream bufferbloat problem. The receiver adjusts its advertised receive window (rwnd) to control the transmission rate. The receiver calculates the loss probability and advertises rwnd according to measured RTT, minimum RTT and estimated queue length. The queue length is estimated directly with RTT and minimum RTT. So RTAC is still a delay-based scheme.

The delay-based protocols can achieve more throughput than LPCC protocols, but they may suffer from some inherent weakness. Firstly, delay-based flows could suffer from significant performance degradation when competing with loss-based flows [21]. Secondly, the performance of these delay-based protocols deteriorates if the delay measurements are noisy. They may suffer from performance degradation on congested links because a large number of packet losses could result in wrong delay estimation.

2.3 The Combination of AQM and End to End Solutions

Gong et al [19] analyze the AQM vs. LPCC reprioritization issue via a fluid model which describes system dynamics of heterogeneous congestion control protocols (namely, TCP and LEDBAT) competing on a bottleneck link governed by AQM (namely, RED) and propose a system level solution able to reinstate priorities among protocols. As an extension of Reference [19], Gong et al [20] show a potentially fateful interplay between AQM and LPCC again: namely, AQM resets the relative level of priority between best-effort and low-priority congestion control protocols and validate the generality of our findings by an extended set of experiments with packet-level ns2 simulation. Nevertheless, Gong et al [20] present evidences that this coexistence may cause a problem called “reprioritization”. As a consequence, LPCCs under the influence of AQMs have their low priority ignored and their flows become as aggressive as normal TCP flows, reducing the benefit of LPCCs usage to avoid bufferbloat.

In short, AQM techniques can directly control latency, but they cannot be easily deployed in current networks. End to end solutions including LPCC and delay-based protocols are deployable, but they cannot achieve good performance when competing with other TCP versions.

3. TCP Delayed Window Update Mechanism

3.1 Overview

TCP's congestion avoidance mechanism could result in self-similarity of TCP flows [26]. In other words, TCP could increase the sending rate till packet losses occur. When a large amount of TCP flows enter the network, the bottleneck link becomes congested and all TCP flows through the bottleneck link can perceive the packet losses at almost the same time. Nevertheless, it is too late because the packets sent in this RTT would maybe have been lost. If the network congestion (or upcoming packet losses) can be perceived as soon as possible, packet losses could be decreased and then bandwidth utilization could become higher by moderately decreasing congestion window or remaining current congestion window unchanged.

In this section, we briefly describe the overall design of TCP delayed window update mechanism. The mechanism is composed of two components: network congestion predication algorithm and window update control strategy. **Fig. 1** presents the architecture of TCP delayed window update mechanism. The network congestion predication algorithm periodically predicts the network congestion level according to network bandwidth utilization and delay jitter. If detecting the network congested, the window update control strategy would update the congestion window according to the congestion level. If detecting the network non-congested, the congestion window is updated as usual. In other words, the window update control strategy works only if the network is detected congested.

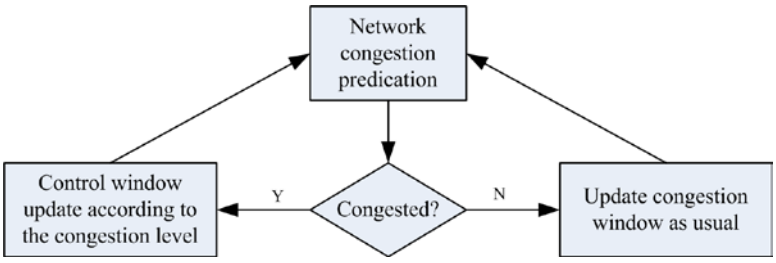


Fig. 1. Architecture of TCP delayed window update mechanism

Fig. 2 depicts the congestion window evolution behavior when the mechanism is adopted in Cubic TCP. At startup, new protocol integrated with the mechanism detects available bandwidth with estimated network bandwidth utilization and delay jitter. When detecting serious congestion, the mechanism would moderately decrease the congestion window. When detecting mild congestion, the mechanism would suspend the window update, i.e., remain current window unchanged for a while, till the network is non-congested. If the network is non-congested, the congestion window is updated according to the original protocol, and the mechanism continues to detect the congestion. When packets are lost, the protocol conducts fast retransmission and fast recovery as usual. After that, the mechanism resets the corresponding variants and restarts to detect the network congestion.

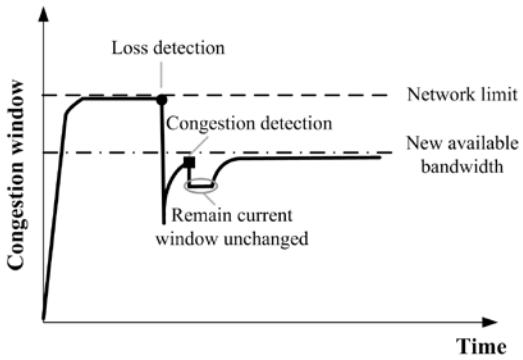


Fig. 2. Congestion window behavior (take Cubic as an example)

3.2 Network Congestion Predication Algorithm

In order to make more reliable predictions on the network congestion, the mechanism uses multi-bits information, i.e. bandwidth utilization and delay jitter, to predict the congestion. The network congestion predication algorithm estimates the congestion level every once in a while (denoted as *period*), and calculates the bandwidth utilization and delay jitter since last estimation. The bandwidth utilization and delay jitter are calculated as follows.

(1) Bandwidth utilization

As mentioned in section 2, delay-based congestion control algorithms (e.g. TCP Vegas) detect the network congestion by estimating short-term queueing delay (i.e. directly use current RTT and minimum RTT on the link to update the congestion window). Although these algorithms can perceive the network congestion early, sometimes they cannot work well due to the estimation errors of delay.

The network congestion predication algorithm would not detect the congestion according to the short-term change of queueing delay, but obtain the overall trend of queueing delay over a period of time by calculating the bandwidth utilization, thereby reducing the impact of individual estimation errors on the predication of the network congestion. The bandwidth utilization U is calculated according to Eq. (1).

$$U = 1 - \frac{noncongested\_num}{total\_num} \tag{1}$$

Where *noncongested_num* is the number of packets experience RTT that equals to RTT_{min} (the minimum RTT), *total_num* is the total number of packets sent by senders.

(2) Delay jitter

Delay jitter reflects the changes of the delay on the link. The network congestion predication algorithm calculates average RTT that all packets experience during the period. Average RTT of adjacent period are denoted as *ave_rtt* and *last_ave_rtt* respectively. Delay jitter is calculated according to Eq. (2).

$$Jitter = ave\_rtt - last\_ave\_rtt \tag{2}$$

In order to achieve reliable congestion predictation, the network congestion predication algorithm predicts the congestion level of the network with both bandwidth utilization U and delay jitter *Jitter* as follows.

$$\text{The network is } \begin{cases} \text{mild congested}, & 0.99 < U < 1 \text{ and } Jitter > 0 \\ \text{serious congested}, & U = 1 \\ \text{non-congested}, & \text{Others} \end{cases} \tag{3}$$

3.3 Window Update Control Strategy

TCP delayed update mechanism adopts different strategies to update congestion window. When the network is thought mild congested, TCP delayed update mechanism would start a timer and delay the window update, i.e. keep current congestion window unchanged for a while. Once the timer is timeout, window update control strategy of the original protocol would be used again. When the network is thought serious congested, TCP delayed update mechanism would suitably decrease the congestion window. When the network is thought non-congested, TCP delayed update mechanism would use window update control strategy in the original protocol.

3.4 Detailed Design of TCP Delayed Update Mechanism

TCP delayed update mechanism can be used in current TCP versions, especially in the loss-based protocols. **Fig. 3** is the place where TCP delayed update mechanism is deployed in TCP versions and **Fig. 4** is the flow chart of TCP delayed update mechanism.

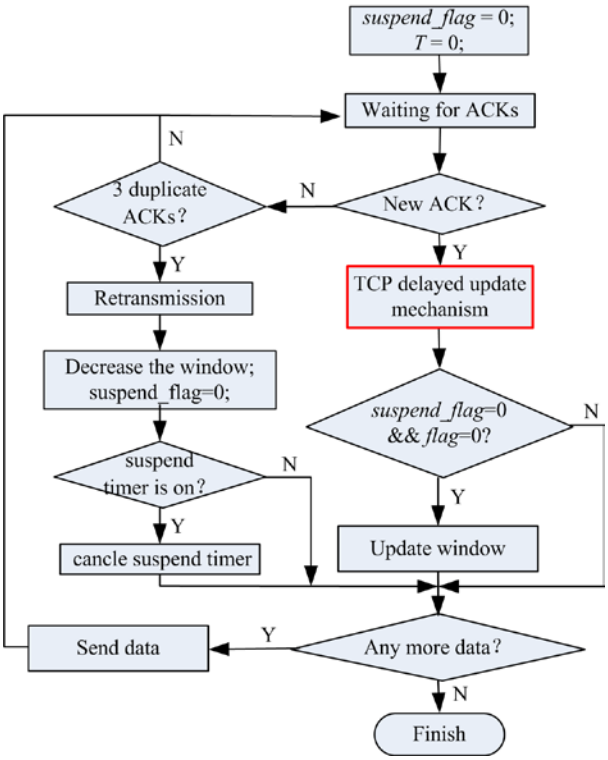


Fig. 3. The place where TCP delayed update mechanism is deployed

It can be seen from **Fig. 3** that TCP delayed update mechanism (in red border) can be implemented in TCP versions as a relative independent part. It can be seen from **Fig. 4** that TCP delayed update mechanism detects the congestion level with a period, denoted as *period*. Then suspend_flag and flag are used to identify whether current network is congested, and suspend_flag is set according to bandwidth utilization U and jitter *Jitter*. When $U > 0.99$ and *Jitter* > 0, the network is thought congested and suspend_flag is set as 1. Specially, when $U = 1$ and *Jitter* > 0, the network is thought serious congested. Although no packet loss is detected, the congestion window is slightly decreased by $1-\theta$ and the suspend timer starts. When

$U \leq 0.99$ and $|Jitter| < threshold$, the network is thought non-congested and suspend_flag is set as 0. In addition, the value of flag is set according to the average RTT (i.e., *ave_rtt*), the maximum RTT (i.e., *max_rtt*) and the minimum RTT (i.e., *min_rtt*). When $ave\_rtt \leq (min\_rtt + max\_rtt)/2$, the flag is set as 0, otherwise is set as 1. Only when both suspend_flag and flag are 0, the network is thought non-congested and the congestion window is updated according to the original protocol. Otherwise, the window update is suspended and current window is remained unchanged.

Once suspend timer is timeout, suspend_flag is set as 0 so as to detect timely available bandwidth with the original protocol when the network load becomes light. If the timer has started when packets are lost, the timer would be canceled and suspend_flag would be set as 0. The value of the suspend timer is critical for the performance of TCP delayed update mechanism. If the timer is too long, the mechanism could not timely detect the change of the bandwidth and the congestion window could not be updated when available bandwidth increases. This would result in the decrease of bandwidth utilization. If the timer is too short, the congestion window could not be effectively controlled. Thus it could not relieve the network congestion. In our mechanism, a random number is used to set the value of the suspend timer and it is ensured that the timer is not more than the minimum RTT observed so far. The suspend timer is calculated according to Eq.(4).

$$suspend\ \ timer = min\_rtt \times rand,\ \ rand \sim U(0,1] \tag{4}$$

where *rand* is a random number with uniform distribution.

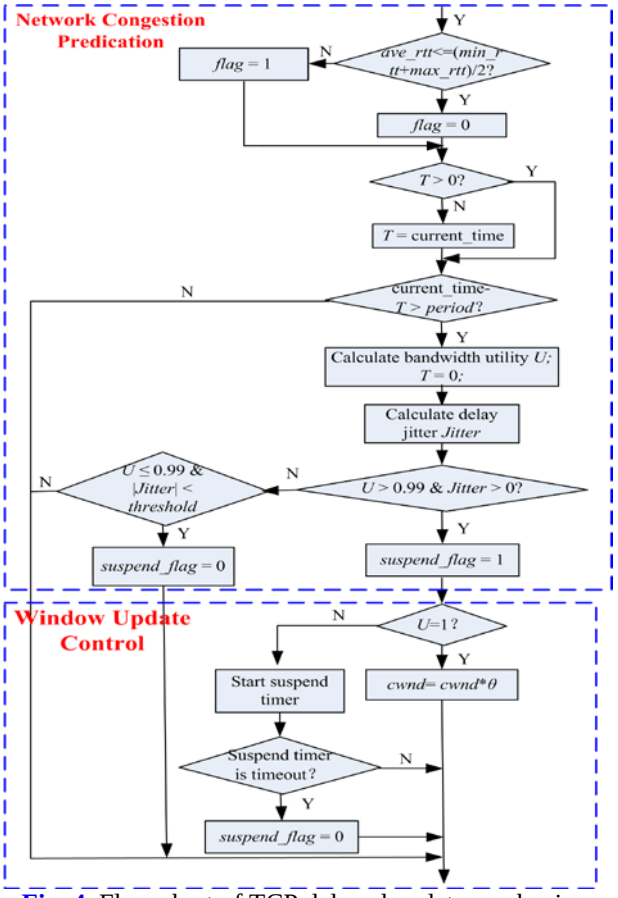


Fig. 4. Flow chart of TCP delayed update mechanism

4. Simulation Experiment Results

We conduct extensive experiments based on network simulators, and the experimental results are presented in this section.

TCP delayed window update mechanism can be used in different end-to-end congestion control protocols. In order to verify its validity, the mechanism is implemented in Cubic TCP which is the default congestion control technique in current Linux operating system [27], denoted as Cubic+. Then we conduct simulation experiments to evaluate Cubic+ using OPNET Modeler. Our experiments consist of two aspects: 1) performance comparison of Cubic+ and other protocols such as Cubic TCP and TCP Vegas in term of bandwidth utilization, packet loss rate, and 2) the effect of TCP delayed window update mechanism on the performance of the original protocol (i.e., Cubic TCP in this paper) when background flows are deployed with them, and the effect on fairness and TCP friendliness of the original protocol in different scenarios and different buffer size.

4.1 Experimental Topology

As shown in **Fig. 5**, a dumb-bell network topology with single bottleneck link shared by one or multiple users is considered in the simulation. Two scenarios, including high bandwidth and long delay scenario and low bandwidth and short delay scenario, are set in our experiments. The parameters in the two scenarios are shown in **Table 1**, where "Scenario I" represents low bandwidth and short delay scenario and "Scenario II" represents high bandwidth and long delay scenario.

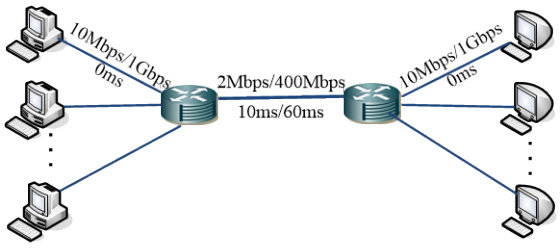


Fig. 5. The network topology with simple bottle link

Table 1. Simulation parameters

Scenario	**I**	**II**
Bottleneck link capacity	2Mbps	400Mbps
RTT	20ms	120ms
Access link capacity	10Mbps	1Gbps
Buffer Size	500KB/50KB	1500KB/300KB
Period	100ms	200ms
threshold	30ms	0.6ms
θ	0.9	
Data packet size	1500B	
Queue type	FIFO	
Simulation time	200s	

4.2 Performance Comparison of Cubic+ and Other Protocols

4.2.1 Single Traffic Flow

We first evaluate the performance of Cubic+ in term of throughput, packet loss rate, buffer usage and average queueing delay when there is a single traffic flow in the network. **Table 2** shows that in both scenarios, Cubic+ and TCP Vegas achieve less packet loss rate and average queueing delay than Cubic. For different buffer size, although the average throughput of Cubic+ is slightly lower than that of Cubic, Cubic+ still achieves good bandwidth utilization while TCP Vegas achieve much less throughput than Cubic+ and Cubic TCP in Scenario II.

In low bandwidth and short delay scenario, the packet loss rate of Cubic reaches up to 3.27% while Cubic+ and TCP Vegas have no packet loss when the buffer size is 500KB. The buffer usage of Cubic is about 81.86% while that of Cubic+ and TCP Vegas is only 1.74% and 0.31% respectively. Average queueing delay of Cubic+ is reduced by about 98.14% compared with Cubic and that of TCP Vegas is much less. This is because Cubic uses packet loss as the only congestion signal while Cubic+ uses both bandwidth utilization and delay jitter to detect the network congestion level in time. Cubic would decrease sending rate only when perceiving packet loss. At this time, lots of packets would fill the buffer and a large amount of packets maybe have been lost, thereby increasing the queueing delay and aggravating the bufferbloat. While Cubic+ can estimate congestion level in time according to the bandwidth utilization and delay jitter. Although packet loss events have not been detected, the mechanism can use different window update strategies, including decreasing window or suspending window update for a while, according to different congestion level, thereby significantly decreasing packet loss and queueing delay. When buffer size is 50KB, Cubic can detect packet loss earlier, resulting in a few of packet losses. Cubic+ and TCP Vegas still have no packet loss. At this time, the buffer usage of Cubic is lower than that when buffer size is 500KB and the average queueing delay of Cubic is significantly reduced. However, the buffer usage of Cubic is still higher than half of the available buffer size. While the buffer usage of Cubic+ is only about one third of the available buffer size and the average queueing delay of Cubic is more than triple that of Cubic+. In this scenario, the buffer usage, the packet loss rate and average queueing delay of TCP Vegas are least among the three protocols because it uses the delay as the congestion signal. TCP Vegas can detect network congestion earlier before packets are lost and decrease sending rate, thereby reducing the rate that packets enter the router buffer and decreasing packet losses.

The bandwidth of bottleneck links increases in high bandwidth and long delay scenario. Hence Cubic and Cubic+ do not result in significant bufferbloat when there is only one flow in the network. When the buffer size is large, the average throughput of Cubic+ is slightly lower than that of Cubic because the window increase of Cubic+ is less aggressive than that of Cubic. However, the bandwidth utilization of Cubic+ still reaches about 87%. TCP Vegas achieves higher performance than other two because it can effectively detect network congestion and decrease packet losses. When the buffer size is small, the packet loss rate of Cubic+ is less than a quarter of that of Cubic and the average queueing delay of Cubic+ is less than one third of that of Cubic. Higher packet loss rate results in frequent decrease of window of Cubic, thus the average throughput of Cubic is lower than that of Cubic+. The packet loss rate and average queuing delay of TCP Vegas are slightly less than that Cubic+. Yet TCP Vegas uses the delay as the only congestion signal and there may be some estimation error of delay when the network is congested, which could result in significant throughput degradation. So the bandwidth utilization of TCP Vegas only reaches about 42% which is significantly lower than that of Cubic+ (about 71%).

Table 2. A single traffic flow

Scenario	Assessment Criteria	Experimental Results					
I	Buffer size	500KB			50KB		
	Protocols	Cubic	Cubic+	Vegas	Cubic	Cubic+	Vegas
	Average throughput (Mbps)	1.9992	1.9988	1.9992	1.9992	1.9988	1.9992
	Packet loss rate (%)	3.27	0	0	0.52	0	0
	Buffer usage (%)	81.68	1.74	0.31	67.14	17.38	3.10
	Average queuing delay(s)	1.61	0.03	0.0064	0.13	0.03	0.0064
II	Buffer size	1500KB			300KB		
	Protocols	Cubic	Cubic+	Vegas	Cubic	Cubic+	Vegas
	Average throughput (Mbps)	368.67	348.87	375.66	203.93	285.68	166.83
	Packet loss rate (%)	0.0284	0.0298	0.0021	0.2187	0.0537	0.0046
	Buffer usage (%)	0.11	0.11	0.11	1.81	0.56	0.11
	Average queuing delay(ms)	0.13	0.13	0.102	0.42	0.13	0.027

4.2.2 Two Traffic Flows

Then we evaluate the performance of Cubic+ when there are two traffic flows in the network and the two flows enter the network at the same time.

4.2.2.1 The Same Flows

In this section, we evaluate the performance of Cubic+ when there are two traffic flows deployed with the same protocols. **Table 3 (a)** and **(b)** show the average throughput, packet loss rate, buffer usage and average queueing delay of the two flows in both scenarios.

It can be seen from **Table 3 (a)** that the total throughput of Cubic+ is lower than that of Cubic in the low bandwidth and short delay scenario because Cubic+ could stop window update or even decrease window once the network congestion is detected (even if packet loss events have not been detected). Therefore, Cubic+ would not “grab” aggressively the bandwidth like Cubic when other traffic flows coexist in the network. Meanwhile, Cubic+ would not make the network excessively congested and would not cause too much packet losses. When buffer size is 50KB, packet loss events occur in the two flows of Cubic and Cubic+ because of small buffer size. However, the packet loss rate of Cubic+ is much lower than that of Cubic and the buffer usage of Cubic+ is less than one third of that of Cubic. Thus, the average queueing delay of Cubic+ is only about one third of that of Cubic. When buffer size is 500KB, Cubic would detect the network congestion later than that when buffer size is small, i.e. the window size would be bigger when detecting packet losses, thereby causing more packet losses. Moreover, the increase of buffer size would cause the increase of queueing delay of Cubic. Instead, the packet loss rate and queueing delay of Cubic+ do not increase with the increase of buffer size. Specifically, the total packet loss rate of Cubic reaches up 8.12% but the packet loss rate of the two flows of Cubic+ is both zero. The buffer usage of Cubic reaches up 82.08% but that of Cubic+ is only 17.41%, resulting that the average queueing delay of Cubic is more than 40 times of that of Cubic+. In both buffer sizes, TCP Vegas achieves lowest packet loss rate and queuing delay meanwhile maintaining higher throughput. This is still because that TCP Vegas can detect network congestion earlier before packets are lost and decrease sending rate, thereby reducing the rate that packets enter the router buffer and decreasing packet losses and queuing delay.

It can be seen from **Table 3(b)** that the total performance of Cubic+ is still higher than that of Cubic though the throughput of Cubic+ is slightly less than that of Cubic. The packet loss rate and queueing delay of Cubic and Cubic+ both increase with the increase of the buffer size. However, Cubic+ always achieves lower packet loss rate and queueing delay. When buffer size is 300KB, the average queueing delay of Cubic+ is only 7.14% of Cubic. When buffer size is 1500KB, the average queueing delay of Cubic is about 20 times of that of Cubic+. This is still because of reliable congestion detection feature and delay update mechanism in Cubic+. In this scenario, the packet loss rate of TCP Vegas is higher than that of Cubic+. This is because when the link bandwidth is high, the difference of the delay is small and thus TCP Vegas cannot detect the network congestion as quickly as possible. The measurement errors of the delay result in some packet losses. Cubic+ uses bandwidth utilization and delay jitter instead of direct delay information to predict the network congestion. Therefore, the measurement error of delay could not significantly affect the performance of Cubic+. It can be seen that the bandwidth utilization of Cubic+ is higher than that of TCP Vegas, especially when the buffer size is 300KB.

Table 3. Two same flows enter the network at the same time

(a) Scenario I

Buffer Size	Assessment Criteria	Cubic1	Cubic2	Cubic+ 1	Cubic+ 2	Vegas 1	Vegas 2
50KB	Average throughput (Mbps)	1.0291	0.9805	0.7900	1.1800	1.0246	0.9752
	Packet loss rate (%)	1.01	0.87	0.02	0.04	0	0
	Average queueing delay (s)	0.13		0.04		0.003741	
500KB	Average throughput (Mbps)	1.0631	0.9269	0.9335	0.9632	1.0246	0.9752
	Packet loss rate (%)	3.97	4.15	0	0	0	0
	Average queueing delay (s)	1.6300		0.0410		0.0037	

(b) Scenario II

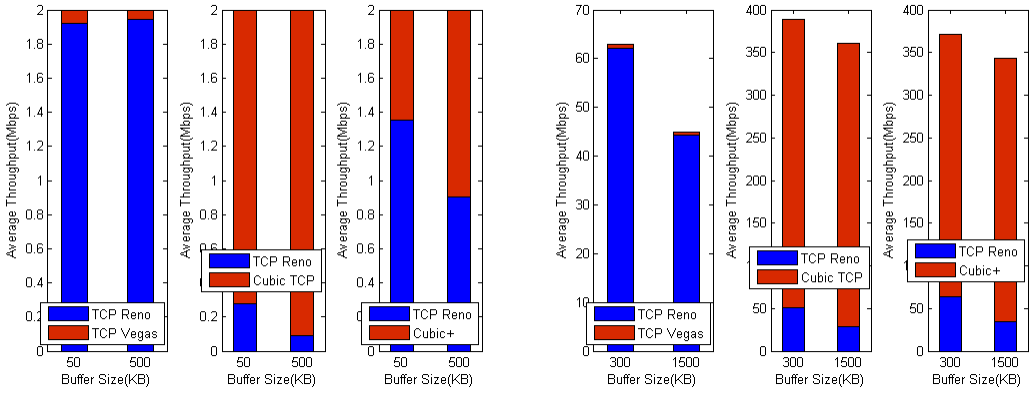
Buffer Size	Assessment Criteria	Cubic1	Cubic2	Cubic+ 1	Cubic+ 2	Vegas 1	Vegas 2
300KB	Average throughput (Mbps)	186.85	186.69	169.53	174.28	130.42	159.05
	Packet loss rate (%)	0.0091	0.0107	0.0053	0.0020	0.0084	0.0038
	Average queueing delay (ms)	0.98		0.07		0.036	
1500KB	Average throughput (Mbps)	195.78	191.52	195.07	183.21	173.81	202.33
	Packet loss rate (%)	0.0243	0.0221	0.0104	0.0097	0.0146	0.0147
	Average queueing delay (ms)	2.36		0.12		0.12	

4.2.2.2 Different Flows

In this section, we evaluate the performance of Cubic+ when it and TCP Reno exist at the same time in the network, i.e., there are two traffic flows deployed with different protocols. In the two flows, one uses TCP Reno and the other uses one of TCP Vegas, Cubic+ and Cubic TCP. The two sources start sending data and terminate simultaneously. **Fig. 6 (a)** and **(b)** show the average throughput of the two flows in both scenarios.

It can be seen from **Fig. 6(a)** that the two flows can obtain full utilization of bandwidth in Scenario I. Cubic+ can share the link resources with TCP Reno more fairly than the other two.

However, TCP Vegas cannot compete with the loss-based flows (e.g., TCP Reno) because TCP Vegas would decrease congestion window (i.e., sending rate) when the delay increases while TCP Reno would not decrease congestion window until packet loss events occur. It can be observed from **Fig. 6(b)** that as the link bandwidth increases, the oscillation of the throughput would become more severe and thus the two flows deployed different protocols cannot reach full bandwidth. Especially, the total throughput of TCP Vegas and TCP Reno is less than one sixth of the total link bandwidth (i.e., 400Mbps) and the average throughput of TCP Vegas is less than 1Mbps in Scenario II. The delayed window update control strategy of Cubic+ would delay congestion window update or decreases moderately the congestion window according to the congestion level, which makes it obtain good throughput performance. In addition, Cubic+ can obtain more throughput than TCP Reno because it uses the window update strategy of Cubic TCP when detecting the network non-congested. Even so, Cubic+ can still share the link resources with TCP Reno more fairly than the other two.



(a) Scenario I (b) Scenario II

Fig. 6. Average throughput of two different flows

4.3 The Effects on Performance of the Original Protocol

In this section, we will evaluate the effects of TCP delayed window update mechanism on the performance of the original protocol when background flows are deployed the original protocol and the effect on the fairness of the original protocol.

4.3.1 Background Flows

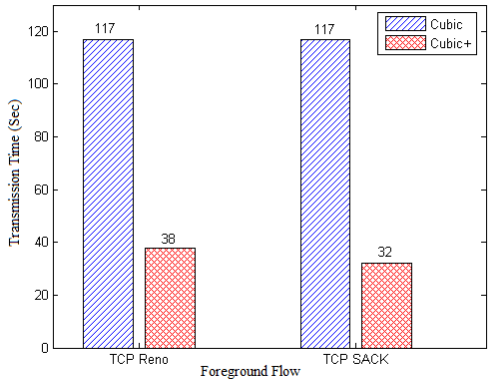
Firstly, we evaluate the effects of Cubic and Cubic+ on the performance of TCP Reno and TCP SACK when the background flows are deployed with Cubic and Cubic+. Two flows enter the network at different time in the simulation. The server deployed with Cubic/Cubic+ first sends data and then the one with TCP Reno/TCP SACK starts to transmit the data after 15 seconds. In the low bandwidth and short delay scenario, the foreground flow transmits 5MB data when the buffer size is 50KB and 500KB when the buffer size is 500KB. In the high bandwidth and long delay scenario, the foreground flow transmits 300MB data when the buffer size is whether 300KB or 1500KB. **Fig. 7** illustrates the completion time of the foreground flows in both scenarios and **Fig. 8** shows the packet loss rate of the flows in Scenario I. It can be seen from **Fig. 7(I)** that the transmission time of the flows deployed with TCP Reno and TCP SACK is both 117 seconds when the background flows use Cubic in the case of small buffer size while that is respectively 38 seconds and 32 seconds when the

background flows use Cubic+. The completion time of the foreground flows reduces 67.52% and 72.65% respectively. When the buffer size is 500KB, the data transmitted by the foreground flows reduces to 500KB. However, the completion time of the flows deployed with TCP Reno and TCP SACK does not significantly reduce when the background flow uses Cubic. There are two reasons for this: (1) the aggressive window increase would make Cubic to "grab" the available bandwidth which should be fairly shared with TCP Reno and TCP SACK; (2) Cubic, TCP Reno and TCP SACK are all loss-based protocols which would result in full buffer and long queueing delay, thereby increasing the completion time of the foreground flows. Cubic+ calculates the bandwidth utilization according to the delay to perceive entering of new flows and then updates the window. Therefore, it would not occupy excessive buffer and would not bring excessive queueing delay. In fact, the average buffer usage of Cubic is about 408KB and that of Cubic+ is only 91KB when buffer size is 500KB. The average queueing delay of Cubic is 1.62 seconds and that of Cubic+ is only 0.36 seconds. For this, the transmission time of the flows deployed with TCP Reno and TCP SACK reduces 87.74% and 75.47% respectively when the background flows use Cubic+ than that when the background flows use Cubic.

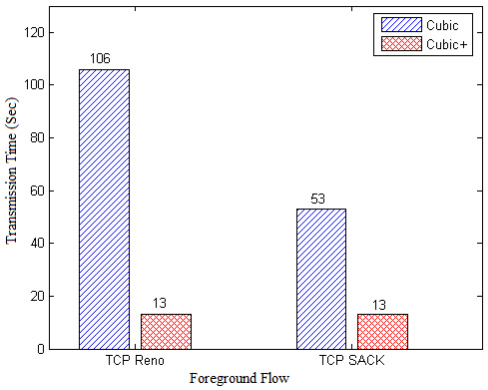
Fig. 7(II) presents the experimental results in the high bandwidth and long delay scenario. The transmission time of the flows deployed TCP Reno and TCP SACK is 123 seconds and 44 seconds respectively when the background flows use Cubic in the case of small buffer size while that is respectively 69 seconds and 38 seconds when the background flows use Cubic+. The completion time of the foreground flows reduces 43.90% and 13.64% respectively. This is mainly because the total packet loss rate when the background flow uses Cubic is higher than that when the background flow is Cubic+ and more packets need be retransmitted. When the buffer size is 1500KB, the total packet loss rate in both cases is very close. The completion time of all the foreground flows is shorter when the buffer size is 1500KB than that when the buffer size is 300KB because of the increase of the average throughput. The queueing delay when the background flows use Cubic+ is lower than that when the background flows use Cubic. Therefore，the completion time of the foreground flows when the background flows use Cubic+ is still the shortest.

Fig. 8 presents the packet loss rate of the foreground flows in low bandwidth and short delay scenario. It can be seen that the total packet loss rate when the background flows use Cubic+ is far lower than that when the background flows use Cubic. Especially, no any packets are lost when the background flows use Cubic+ in the case of large buffer size. The packet loss rate when the background flows use Cubic reaches up to 3.08% (when the foreground flows use TCP SACK). This is because Cubic uses aggressive window update which exacerbates the network congestion, resulting in much more packet losses and degrading the transmission performance of other flows. Cubic+ could decrease sending rate in advance before packet losses occur so that the network congestion would not be exacerbated and packet losses could be decreased. Thus, Cubic+ can reduce the impact on other coexisting flows and can efficiently utilization the network bandwidth.

In a word, TCP delayed window update mechanism can help the original protocol (i.e., Cubic in this paper) reliably detect the network congestion and effectively control queuing delay and packet loss rate, thereby reducing the impact on other coexisting flows.

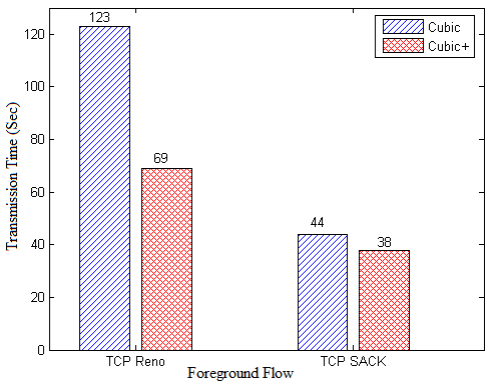


(a) Buffer size is 50KB

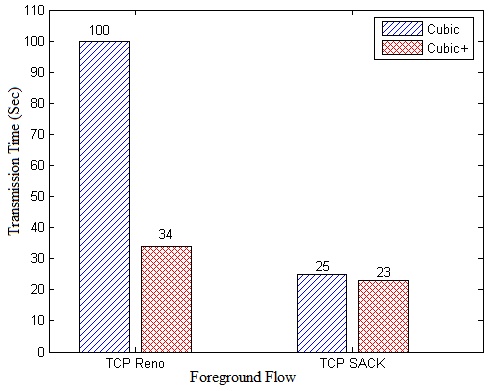


(b) Buffer size is 500KB

(I) Scenario I



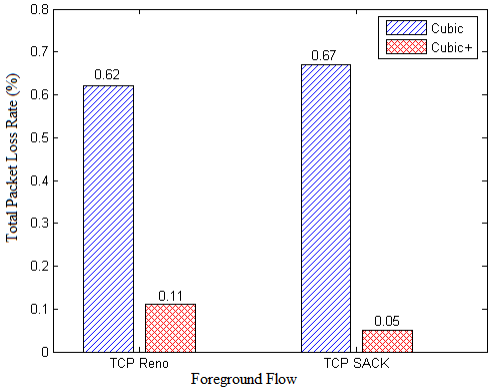
(a) Buffer size is 300KB



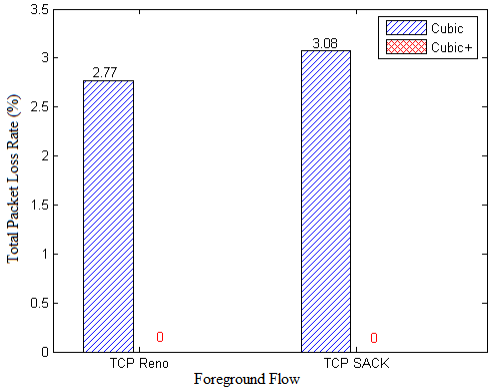
(b) Buffer size is 1500KB

(II) Scenario II

Fig. 7. Flow completion time of the foreground flows



(a) Buffer size is 50KB



(b) Buffer size is 500KB

Fig. 8. Packet loss rate of the foreground flows in Scenario I

4.3.2 Fairness

Then, we evaluate the fairness performance of Cubic+. We consider two different scenarios, including homogeneous RTT and heterogeneous RTT, and use the Jain's fairness index (FI) [28] to quantitatively evaluate the fairness performance of the protocol.

For homogeneous RTT scenarios, two TCP flows pass through the same bottleneck path and the RTT is the same for all users. The two sources start sending data at 10s and terminate at 210s simultaneously. According to the average throughput in **Table 3**, we can obtain the fairness index of Cubic and Cubic+ presented in **Table 4**. It can be seen that Cubic+ achieves the same fairness with Cubic in both scenarios, i.e., the TCP delayed window update mechanism does not change the fairness of Cubic.

Table 4. Fairness for homogeneous RTT scenarios

	Scenario I		**Scenario II**	
Buffer size (KB)	50	500	300	1500
Cubic	0.9996	0.9953	0.9999	0.9999
Cubic+	0.9623	0.9997	0.9998	0.9990

For heterogeneous RTT scenarios where two TCP flows with different RTT ratios share the bottleneck link, Cubic+ still can achieve good fairness in both scenarios. The RTT of one of the two flows is fixed to 20ms and that of the other flow is one of 40ms and 60ms, thus the RTT ratios of the two flows are 2 and 3 respectively. **Table 5** presents the results in the low bandwidth and short delay scenario. The efficiency index, EI, is the amount of the concurrent flows' throughput and FI is fairness index. It can be seen from **Table 5(a)** that all protocols can achieve good fairness when buffer size is 50KB. When the buffer size increases, the flow with short RTT can occupy more buffers and thereby increasing the unfairness. It can be seen from **Table 5(b)** that the fairness performance of Cubic and Cubic+ is both degraded in some degree when buffer size is 500KB. However, Cubic+ can still achieve better fairness performance than Cubic because it can control the buffer size occupied by the flows well, i.e., the TCP delayed window update mechanism could keep and even improve the fairness performance of Cubic.

Table 5. Simulation results of EI and FI in Scenario I

(a) Buffer size is 50KB

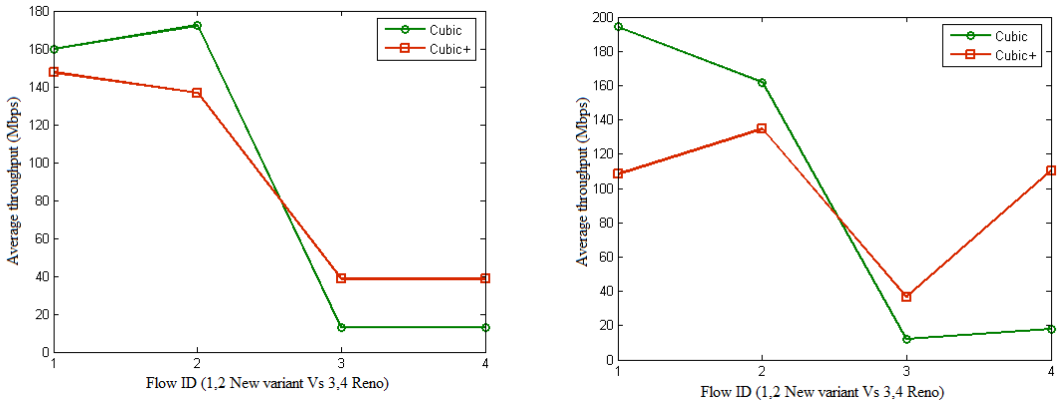
RTT Ratio	**2**				**3**			
Protocols	**T1**	**T2**	***EI***	***FI***	**T1**	**T2**	***EI***	***FI***
TCP Reno	0.96	1.02	1.98	**0.9989**	0.86	1.10	1.97	**0.9854**
Cubic	0.99	1.01	2.00	**0.9999**	0.87	1.13	2.00	**0.9838**
Cubic+	0.94	1.06	2.00	**0.9963**	0.97	1.03	2.00	**0.9990**

(b) Buffer size is 500KB

RTT Ratio	**2**				**3**			
Protocols	**T1**	**T2**	***EI***	***FI***	**T1**	**T2**	***EI***	***FI***
TCP Reno	0.96	1.04	2.00	**0.9982**	0.94	1.06	2.00	**0.9959**
Cubic	0.71	1.29	2.00	**0.9215**	0.73	1.27	2.00	**0.9325**
Cubic+	0.99	1.01	2.00	**0.9998**	0.87	1.13	2.00	**0.9830**

4.3.3 TCP-friendliness

TCP-friendliness is referred to the fairness between new protocol and TCP Reno. For the evaluation of TCP-friendliness performance, we conduct the simulations that two sources are specified to run TCP Reno while two sources implement the TCP variants (Cubic or Cubic+) in a homogeneous RTT scenario. The experimental results show that Cubic+ performs good TCP-friendliness in different scenarios. **Fig. 9** presents the results in the high bandwidth and long delay scenario. They show the average throughput of the four flows with different buffer sizes in which the number 1 and 2 denote the flows using Cubic or Cubic+ and the number 3 and 4 denote the flows using TCP Reno. From these figures, it can be seen that TCP-friendliness of Cubic and Cubic+ when buffer size is large is better than that when buffer size is small. Cubic overall performs unfair and significantly "grab" the bandwidth of TCP Reno and thereby reducing the average throughput of TCP Reno flows. Cubic+ does not always suppress the concomitant TCP Reno flows and achieves better TCP-friendliness performance than Cubic. Cubic+ is not like Cubic which increases the congestion window till packet losses occur. It uses the bandwidth utilization and delay jitter to detect the congestion and moderately decreases the congestion window. Therefore, Cubic+ flows would not occupy too much buffer at the routers, and the redundant buffer is available for the TCP Reno flows, which results that the TCP Reno flows could share the bandwidth with the Cubic+ flows. That is, TCP delayed window update mechanism can also improve TCP-friendliness performance of the original protocol.



(a) Buffer size is 300KB (b) Buffer size is 1500KB

Fig. 9. TCP-friendliness in Scenario II

5. Conclusions

In this paper, we propose a TCP delayed window update mechanism as the solution to bufferbloat and implement it in Cubic TCP. The mechanism utilities bandwidth utilization and delay jitter to predict the congestion level of networks, and then delays congestion window update or decreases moderately the congestion window according to the congestion level. The simulation results show that Cubic+ achieves significant performance improvements in term of packet loss and queueing delay over Cubic TCP and TCP Vegas while maintaining good throughput in different scenarios. In addition, the experiment results also show that TCP delayed window update mechanism can keep and even improve the fairness and TCP-friendly performance of the original protocol.

TCP delayed window update mechanism can be adopted in various TCP versions. Currently, we only implement it in Cubic. As future work, we intent to implement the mechanism in more

TCP versions and investigate its performance in a wide range of the network environment.

References

[1] Vint Cerf, Van Jacobson, Nick Weaver, Jim Gettys, "Bufferbloat: what's wrong with the internet? ," *Commun ACM*, vol. 55, no. 2, pp. 40-47, 2012. Article (CrossRef Link).

[2] Cardozo T B, da Silva A P C, Vieira A B, et al., "Bufferbloat systematic analysis," in *Proc. of 2014 International Telecommunications Symposium (ITS)*, pp. 1-5, Aug 17-20, 2014. Article (CrossRef Link).

[3] Alfredsson S, Del Giudice G, Garcia J, et al., "Impact of TCP congestion control on bufferbloat in cellular networks," in *Proc. of 2013 IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1-7, June 4-7, 2013. Article (CrossRef Link).

[4] Ha S, Rhee I, Xu L, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64-74, 2008. Article (CrossRef Link).

[5] Floyd S, Jacobson V, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on networking*, vol. 1, no. 4, pp. 397-413, 1993. Article (CrossRef Link).

[6] Pan R, Prabhakar B, Psounis K, "CHOKe-a stateless active queue management scheme for approximating fair bandwidth allocation," in *Proc. of. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, pp. 942-951, March 26-30, 2000. Article (CrossRef Link).

[7] G. Carofiglio, L. Muscariello, "On the Impact of TCP and Perflow Scheduling on Internet Performance," in *Proc. of 2010 IEEE INFOCOM*, pp. 1-9, Mach 15-19, 2010. Article (CrossRef Link).

[8] Nichols K, Jacobson V, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42-50, 2012. Article (CrossRef Link).

[9] Nichols K, Jacobson V, McGregor A, et al., "Controlled Delay Active Queue Management," *draft-ietf-aqm-codel-00*, *Internet Engineering Task Force*, March 10, 2014. Article (CrossRef Link).

[10] Pan R, Natarajan P, Piglione C, et al., "PIE: A lightweight control scheme to address the bufferbloat problem," in *Proc. of 2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, pp. 148-155, July 8-11, 2013. Article (CrossRef Link).

[11] Lin X A, Zhang D, "Kemy: An AQM generator based on machine learning," in *Proc. of 2015 10th International Conference on Communications and Networking in China (ChinaCom)*, pp. 556-561, August 15-17, 2015. Article (CrossRef Link).

[12] Jiang H, Liu Z, Wang Y, et al., "Understanding bufferbloat in cellular networks," in *Proc. of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pp. 1-6, August 13-17, 2012. Article (CrossRef Link).

[13] L.S. Brakmo and L.L. Perterson, "TCP Vegas: End-to-End Congestion Avoidance on a Global Internet," *IEEE Journal on Selected Areas in Communication*, vol. 13, no. 8, pp. 1465-1480, 1995. Article (CrossRef Link).

[14] Venkataramani A, Kokku R, Dahlin M, "TCP Nice: A mechanism for background transfers," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 329-343, 2002. Article (CrossRef Link).

[15] M. Kuehlewind, G. Hazel, S. Shalunov, J. Iyengar, "Low Extra Delay Background Transport (LEDBAT)," *IETF RFC6817*, Dec. 2012. Article (CrossRef Link).

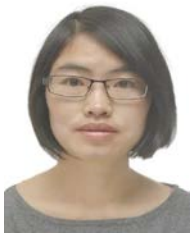
[16] C. Chirichella and D. Rossi, "To the Moon and back: are Internet bufferbloat delays really that large?," in *Proc. of IEEE INFOCOM Workshop on Traffic Measurement and Analysis*, pp. 417-422, April 14-19, 2013. Article (CrossRef Link).

[17] G. Hasegawa, K. Kurata, and M. Murata, "Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet," in *Proc. of IEEE ICNP*, pp. 177-186, November 14-17, 2000. Article (CrossRef Link).

[18] K. Srijith, L. Jacob, and A. Ananda, "TCP Vegas-A: Improving the performance of TCP Vegas," *Computer Communications*, vol. 28, no. 4, pp. 429-440, 2005. Article (CrossRef Link).

[19] Y. Gong, D. Rossi, E. Leonardi, "Modeling the Interdependency of Low-priority Congestion control and Active Queue Management," in *Proc. of International Teletraffic Congress (ITC)*, pp. 1–9, September 10-12, 2013. Article (CrossRef Link).
[20] Y. Gong, D. Rossi, C. Testa, S. Valenti, et al., "Fighting the bufferbloat: on the coexistence of AQM and low priority congestion control," *Computer Networks*, vol. 65, pp. 255-267, 2014. Article (CrossRef Link).
[21] S. Liu et al, "TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks," in *Proc. of First International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS)*, pp. 417-440, October 11-13, 2006. Article (CrossRef Link).
[22] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM Computer Communication Review*, vol. 18, no. 4, pp. 314-329, 1988. Article (CrossRef Link).
[23] D.X. Wei, C. Jin, S.H. Low, S. Hegde, "FAST TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246-1259, 2006. Article (CrossRef Link).
[24] Jiang H, Wang Y, Lee K, et al., "Tackling bufferbloat in 3G/4G networks," in *Proc. of the 2012 ACM conference on Internet measurement conference*, ACM, pp. 329-342, November 14-16, 2012. Article (CrossRef Link).
[25] Im H, Joo C, Lee T, et al., "Receiver-side TCP Countermeasure to Bufferbloat in Wireless Access Networks," *IEEE Transactions on Mobile Computing*, vol. 15, no. 8, pp. 2080-2093, 2015. Article (CrossRef Link).
[26] Sikdar B, Vastola K S, "The effect of TCP on the self-similarity of network traffic," in *Proc. of 35th Annual Conference on Information Science and Systems(CISS)*, pp. 1-6, March 21-23, 2001. Article (CrossRef Link).
[27] Ahmad U, Ngadi M A B, Isnin I F B, "Fairness Evaluation and Comparison of Current Congestion Control Techniques," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 1, no. 1, pp. 176-181, 2016. Article (CrossRef Link).
[28] Jain R, Chiu D M, Hawe W, "A quantitative measure of fairness and discrimination for resource allocation in shared systems," *DEC TR-301, Littleton*, MA: Digital Equipment Corporation, 1984. Article (CrossRef Link).

Min Wang received the Ph.D. degree in Computer Science from Sichuan University of China, Chengdu in 2015. Now she is with the College of Computer Science and Technology, Yunnan Normal University. Her research interests cover a wide variety of topics in wireless and satellite networks, with emphasis on design of transport layer protocols for wireless networks.



Lingyun Yuan is with the College of Computer Science and Technology, Yunnan Normal University. Her major is Internet of Things (IOT) and Wireless Sensor Network.